Abstract

Functional Programming with Logical Frameworks

Adam Brett Poswolsky

2008

Logical frameworks are languages used to represent information. In this dissertation we present the Delphin programming language, which is a functional programming language with a logical framework supporting both higher-order abstract syntax and dependent types. Higher-order abstract syntax, or HOAS, refers to the technique of representing variables of an object language using variables of a meta-language, which leads to more concise and elegant encodings than first-order alternatives. Dependent types allow one to represent complex data (such as *derivations* in various logics) and enforce more properties of programs than possible using only simple types.

Delphin is not only a useful programming language but also a useful system for formalizing proofs as total functions express proofs that the input entails the output. Just as representations using HOAS free the programmer from concerns of handling explicit contexts and substitutions, our system permits computation and reasoning over such encodings without making these constructs explicit, leading to concise and elegant programs. Delphin is a two-level system distinguishing functions used for representation from functions expressing computation. The ability to perform computation over higher-order data is driven by a construct allowing the dynamic introduction of scoped constants, which we call parameters.

We motivate our system with examples of translating derivations between logics and converting between higher-order and first-order representations of data. The Delphin website can be reached at `http://www.delphin.logosphere.org/`.

Functional Programming with Logical Frameworks

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Adam Brett Poswolsky

Dissertation Director: Carsten Schürmann

December 2008

Contents

List of Figures

Acknowledgments

My graduate studies have been challenging and rewarding. My research goals were to combine theory with practice, which has been achieved in my formalization and development of the Delphin programming language.

Foremost, I thank my advisor Carsten Schürmann who introduced me to, and guided me through, my research endeavors. I've learned a great deal from you and value your advice, support, and friendship. Your dedication was inspiring and evidenced by our many discussions and heated debates. Your feedback has been deep, profound, and immensely detailed. Thank you for checking over the mammoth appendix of this dissertation.

Second, I thank Jeffrey Sarnat for the hours spent discussing the current formulation, and the many superseded preliminary formulations, of Delphin.

My studies spanned both sides of the Atlantic. My first years were spent at Yale University. I found the classes rewarding and the faculty readily approachable to answer my questions. My teaching responsibilities were enjoyable and enlightening. I've looked up to many faculty members as academic role models. Most notably, in alphabetical order: Michael J. Fischer, Paul Hudak, Arvind Krishnamurthy, Drew V. McDermott, Carsten Schürmann, Zhong Shao, and Yang Richard Yang.

My final years of study took place in Denmark, where I concluded my research in the PLS group at the IT University of Copenhagen. Thank you all for both welcoming

me and treating me as one of your own. I thank Anders Schack-Nielsen for your input, especially with regard to implementation concerns. I thank Kristian Støvring for your interest and our discussions. I thank Rasmus Lerchedahl Petersen for your encouragement, support, and feedback. I am grateful to Andrzej Filinski, from the University of Copenhagen, for reading a draft of my dissertation and providing detailed feedback.

I am fortunate to have had a supportive thesis committee consisting of Paul Hudak, Greg Morrisett, Carsten Schürmann, and Zhong Shao. Thanks to Greg Morrisett, from Harvard University, for serving as the external member of my committee, showing earnest interest, and providing insightful questions and feedback.

Special thanks to the administrative staff of the Department of Computer Science at Yale University. In particular, Linda Dobb, who made sure I felt as part of Yale while abroad.

Aside from academics, I owe thanks to all my friends, old and new, home and abroad, for your support and the many wonderful experiences that have not only provided an escape from work, but have also shaped me personally and professionally.

Finally, I thank my family. First, to my parents Linda and Mel for your boundless guidance, love, and support; I owe my successes to you. To my siblings Dan, Kim, and Matt; I thank you for your love, support, and for your companionship both near and far.

Chapter 1

Introduction

The *Merriam-Webster Online Dictionary* defines a computer as "a programmable usually electronic device that can store, retrieve, and process data."[1] However, what is data? Modern computers, at their core, simply manipulate a collection of bits, where a bit is a binary digit (whose two possible values are typically written as 0 and 1). Before one may manipulate data, the first challenge is in *representing* such data. For example, we will consider representing the numbers $1, 2, 3, 4, 5$. The ancient Romans would represent such numbers as $\boldsymbol{i, ii, iii, iv, v}$. The typical representation in a computer is its binary equivalent $\mathbf{1, 10, 11, 100, 101}$. Characters (letters) are typically represented following the ASCII standard which provides a mapping to binary numbers. Similarly, all other kinds of data are eventually mapped to binary numbers. Although all data is indistinguishable at this base level, type systems distinguish data intended to represent different things (as having different types), as well as statically ensure various properties. For example, if we were to write a function to multiply two numbers, then the type system can guarantee that the function is only called on data which was intended to represent numbers.

[1] Retrieved May 8, 2008

One can distinguish different programming languages not only by the tools they afford one in representing and manipulating data, but also the static guarantees afforded by their type systems. The more powerful the type system, the fewer programs you can write (since the system restricts you), but the more properties you can guarantee.

Invariably, before one can manipulate data, one must first represent said data in the computer. In this dissertation we emphasize the importance of rich tools for representing data and introduce the functional language Delphin, which is designed to allow one to manipulate these rich encodings. The representational tools we provide allow the programmer to elegantly represent languages with bindings as well as deductive systems. In other words, the system for representation is rich enough to express judgments such as the derivability of formulas in arbitrary logics, typing rules, operational semantics, and so on. As a side effect, if functions are total, they can also be interpreted as proofs. For example, if we can write total functions that translate, in both directions, between derivations in a natural deduction calculus and a Hilbert-style calculus, then we have a proof that both logics are equivalent with respect to provability, which will be an illustrated example in this dissertation.

1.1 Representing Data

Imperative systems such as C (Kernighan and Ritchie 1988) offer the user an a priori collection of types (representations) as well as some limited support for defining ones' own types. The C programming language gives the programmer great freedom in specifying programs but at the price of weak static guarantees. Programmers have invariably encountered *segmentation faults*, indicating that the program ran into something unexpected (e.g. dereferencing a null pointer). The problem with such

errors is that they are only discovered during runtime, when the program is executed. Other languages are designed with a strong type system to statically guarantee (at compile time) that such problems will never be encountered.

One such strongly-typed language is Java (Lindholm and Yellin 1996)[2] which allows users to declare their own types following the object-oriented paradigm. However, the imperative nature of programming has proved to be quite an obstacle in statically guaranteeing properties of programs.

Imperative languages such as C and Java treat computation as a recipe where the programmer writes a *procedure* describing changes in state. Alternatively, functional languages such as Standard ML (Milner et al. 1997) (also called SML) and Haskell (Thompson 1999; Hudak 2000) emphasize a mathematical notion of applying functions to persistent data. Therefore, in a functional setting, adding an element to the middle of a list may require the creation of a new list duplicating all the values in the old one. However, the persistence of data and the mathematical design affords an ease in proving properties about the programs we write, which is why we will also adopt a functional paradigm.

Functional languages have also pioneered an enhancement in representing data. Besides providing standard a priori types, the programmer can define their own types, or datatypes. For example, natural numbers $\mathbb{N} ::= 0, 1, 2, 3, \ldots$ can be represented in ML as:

```
datatype nat
    = s of nat
    | z
```

This encoding exploits that every natural number is either 0 or the successor of

[2]We consider Java to be strongly-typed even though there is a bug in the subtyping of arrays, which has persisted to provide backwards compatibility.

another number. For example, `0` is mapped to **z**, and `2` is mapped to **s (s z)**.

One application of datatypes is to represent other languages, which will be a common theme throughout this dissertation. The most complicated use today for such representations is usually in constructing compilers, but we will show how significant reasonings over languages can be conducted once one is provided with the proper tools for representing and manipulating data.

One significant stride forward in mainstream functional languages has been in the advent of *Generalized Algebraic Datatypes*, or GADTs (Kennedy and Russo 2005; Jones et al. 2006). We can use GADTs to statically enforce more properties of a program. For example, the typical `hd` function returns the first element of a list and is undefined (or raises an exception) when applied to an empty list. Using GADTs in Haskell, we can write a version of this function which is guaranteed to always be called on a nonempty list:[3]

```
data Empty
data NonEmpty
data List x y where
     Nil :: List a Empty
     Cons:: a -> List a b ->  List a NonEmpty

hd:: List x NonEmpty -> x
hd (Cons a b) = a
```

GADTs provide a form of *indexed types*. For example, **List** is not itself a type, but **List int NonEmpty** is a type. The ability to write a better `hd` function illustrates the usefulness of more powerful type systems. However, it also exhibits the double-edged sword of using enriched datatypes. Namely, we get more guarantees but the programmer must do more work. Perhaps the concept of handling exceptions is

[3] taken from Generalised algebraic datatype – HaskellWiki, May 13, 2008, `http://www.haskell.org/haskellwiki/GADT`

easier to grasp, but there is no doubt that the above encoding adds static guarantees that are often desired.

1.1.1 Dependent Types and Representing Judgments

We now turn to the representation of judgments. GADTs can be thought of as providing some form of *dependent types* where types are indexed by other types. Dependencies are very useful in both capturing properties of programs as well as in representing judgments. For example, consider the following Hilbert-style calculus, which is an *axiomatic* deductive system, also known as a system of combinators:

$$\frac{}{\vdash A \supset B \supset A}\,\mathsf{K} \qquad \frac{}{\vdash (A \supset B \supset C) \supset (A \supset B) \supset A \supset C}\,\mathsf{S} \qquad \frac{\vdash A \supset B \quad \vdash A}{\vdash B}\,\mathsf{MP}$$

We can represent derivations $\vdash A$ using GADTs in Haskell as:[4]

```
data Comb x where
    K :: Comb (a -> b -> a)
    S :: Comb ((a -> b -> c)  -> (a -> b) -> a -> c)
    MP :: Comb (a -> b) -> (Comb a) -> Comb b
```

Notice that we can prove $\vdash A \supset A$ as follows:

$$\dfrac{\dfrac{\dfrac{}{\vdash (A \supset (A \supset A) \supset A) \supset (A \supset A \supset A) \supset A \supset A}\,\mathsf{S} \quad \dfrac{}{\vdash A \supset (A \supset A) \supset A}\,\mathsf{K}}{\vdash (A \supset A \supset A) \supset A \supset A}\,\mathsf{MP} \quad \dfrac{}{\vdash A \supset A \supset A}\,\mathsf{K}}{\vdash A \supset A}\,\mathsf{MP}$$

This *derivation* trace is represented in Haskell as **MP (MP S K) K** which has the type **Comb (a -> a)**. This exhibits what is known as the *judgments-as-types* paradigm. Terms of type **comb *X*** represent derivation trees of $\vdash X$.

[4]based on Generalised algebraic datatype – HaskellWiki, May 13, 2008, `http://www.haskell.org/haskellwiki/GADT`

1.1.2 Higher-Order Abstract Syntax

We have seen that dependencies are quite useful in representing data, but most languages and logics have a notion of variable bindings. In fact, we will show how to convert between a Hilbert-style calculus and a natural deduction calculus (or equivalently, derivations in the simply-typed λ-calculus). This begs the question of how to represent languages with bindings. For example, consider expressions of the untyped λ-calculus, which we call λ-expressions e:

$$e ::= x \mid \lambda x.\ e \mid e_1\ e_2$$

A first-order representation of this calculus can easily be encoded in Haskell:

```
data term where
    var :: string -> term
    lam :: string -> term -> term
    app :: term -> term -> term
```

As an example, we can use this datatype to represent the λ-expression $\lambda x.\ \lambda y.\ x$ as `lam "x" (lam "y" (var "x"))`. Here we have chosen to represent the variable x as a string `"x"`. The problem with this representation is that the programmer is hassled with tedious concerns involving variable renamings in the construction of capture-avoiding substitutions. One alternative is to represent variables with something better than strings. There has been recent development with programming languages with freshness (Gabbay and Pitts 2001), such as FreshML (Pitts and Gabbay 2000), which utilizes Fraenkel-Mostowski (FM) set theory to provide a built-in α-equivalence relation for first-order encodings. In other words, these languages represent variables with something special instead of strings, which alleviates the programmer of concerns with α-renaming (variable renamings). Therefore, the

programmer still needs to write out functions to do the substitutions, but they are easier to write.

Alternatively, we advocate the use of *higher-order abstract syntax*, or HOAS. Rather than representing variables x as strings, we will represent them as Haskell variables. More precisely, HOAS refers to representing variables of the object language being encoded as meta-level variables in the system where it is being encoded. This results in a deceptively simple idea of representation – we will represent functions in the untyped λ-calculus using Haskell functions:

```
data exp where
    lam :: (exp -> exp) -> exp
    app :: exp -> exp -> exp
```

Now the λ-expression $\lambda x.\, \lambda y.\, x$ is represented as **lam (\x -> (lam (\y -> x)))**. Notice that the variable x is represented as a Haskell variable **x** of type **exp**. The key benefit of this encoding is that the Haskell developers have already dealt with handling Haskell variables and we can reap the benefits in our object-level encoding. Substitutions in the object language now corresponds to Haskell function application. For example, if we were to write an evaluator, we could reasonably expect the term $(\lambda x.\, e)\, m$ to step to $e[m/x]$, which is the term resulting from substituting all occurrences of x in e with the term m. In Haskell, the λ-expression $(\lambda x.\, e)\, m$ can be represented as **app (lam F) M** where **F** is more precisely **(\x -> E)**. Now notice that **E[M/x]** corresponds simply to Haskell (meta-level) application **F M**. Therefore, β-reduction (function application) on the object language (untyped λ-calculus) corresponds to β-reduction on the meta-level (Haskell).

Unfortunately, not all is well when using HOAS in Haskell. It is easy to see that every object-level λ-expression e can be represented as a Haskell term **E** of type

exp, but it is not the case that every typeable term of type **exp** corresponds to something in the object language. For example, we can write the term **lam (\x -> case x of ...)** which has type **exp**, but it is not clear what λ-expression e it is in correspondence with. This is an example of an *exotic* term (Despeyroux et al. 1995; Schürmann et al. 2001), i.e. typeable terms that do not correspond to any concrete term in the object language being encoded. The existence of such terms may render our encoding meaningless if we would ever want to view functions as proofs. After all, we are no longer proving something about the untyped λ-calculus, but we are proving something about the untyped λ-calculus with extra stuff. For example, integers can be viewed as natural numbers with extra stuff (negative numbers), but not all properties of integers also hold for natural numbers (e.g. the existence of a predecessor). Therefore, when interpreting functions as proofs, it is important that the encoding corresponds to the object language being encoded.

We say an encoding is *adequate* if there exists a suitable correspondence between terms in the object language being represented and typeable values in the encoding. In other words, we need to be able to provide a map between object-level terms and their encoding, and vice versa.

1.2 Contributions

We have seen that both dependent types and higher-order abstract syntax (HOAS) are powerful tools to represent data. HOAS refers to the technique of representing variables of an object language as variables in a meta-logic. However, we have already seen the first challenge: if the meta-level used for representation is too powerful (e.g. includes case analysis or recursion), then it renders HOAS an unattractive option due to the existence of exotic terms.

The second challenge is supporting recursion over higher-order encodings. In the first-order representation **`term`**, it is straightforward to write a recursive function, but it is not so in the higher-order version **`exp`**. Both versions are intended to represent λ-expressions e which are made up of three things: variables, applications, and abstractions. Therefore, a recursive function over λ-expressions would be expected to handle three cases. In the first-order version, the three cases are clear, albeit messy to deal with. In the higher-order version we have two questions we need to ask ourselves: (1) What happened to the variable case? and (2) How do we recurse under a function? Recall that in the higher-order encoding we may encounter a term **lam `F`** where **`F`** has type **exp `->` exp**. If we are writing a recursive function over terms of type **exp**, it is not clear how to continue once we encounter a functional type.

In this dissertation we tackle both these issues and provide a novel way to manipulate and reason about complex data through the programming language we develop, called Delphin. To solve the first issue, we will make Delphin a two-level system distinguishing functions used for representation from functions expressing computation. The representation level is a relatively weak system whose sole purpose is to support the adequate encodings of data, languages, logics, and judgments that utilize dependent types and HOAS. The computation level supports recursion and addresses the second challenge by providing a construct to dynamically introduce scoped constants, which we call parameters.

The Delphin programming language has been implemented and is available for download at `http://www.delphin.logosphere.org/`. We will dicuss implementation concerns as well as provide a user manual with examples. Besides offering the first implemented functional programming language combining dependent types and HOAS, we will discuss the benefits of interpreting total functions in this paradigm

as proofs. Not only can Delphin be used as a replacement for writing proofs in Twelf (Pfenning and Schürmann 1998), but it could also replace the underlying meta-logic of Twelf shedding light into what it means to be a proof in Twelf – there would exist a total function in Delphin of the desired type.

1.3 Outline

Our thesis is that a programming language supporting a higher-order dependently-typed logical framework provides a foundational way to manipulate and reason about data. To this end we have developed the Delphin programming language. We will begin this dissertation, Chapter 2, by continuing our discussion of representation. Namely, the representation level of Delphin is the Edinburgh Logical Framework LF (Harper et al. 1993), without modification. LF has proved to be a successful system for representing data, but the ability to reason over such encodings has been elusive. The purpose of Delphin is to provide tools to manipulate LF encodings as if they were strings, integers, lists, and so on. Chapter 2 introduces all of our datatypes, including encodings of a Hilbert-style calculus (combinators) and a simple system of natural deduction. The conversion between derivations in these two systems will be a running example of this dissertation. We will also present encodings of the untyped λ-calculus using HOAS and *de Bruijn indices*. De Bruijn indices (de Bruijn 1972) are a first-order technique of representing bindings where variables are represented as numbers pointing to where the variable is bound. We will use Delphin to convert between a HOAS and a de Bruijn encoding of λ-expressions.

As the most important contribution of this work is in dealing with HOAS, we will first discuss a simplified version of Delphin in Chapter 3 considering only simple types. This will be extended to the full system over LF in Chapter 4. We will prove

type safety in Chapter 5. The main issue of type safety is showing that a list of cases (in a function definition) is exhaustive, a property also known as coverage. The issue of coverage in ML and Haskell is straightforward, but issues of dependent types and HOAS make it worthy of special attention in Delphin. Coverage checking is especially important in Delphin not only to ensure that programs do not get stuck (*Match Non-Exhaustive Error*) but also to be justified in interpreting functions as proofs.

We will discuss the implementation and give the user manual for the actual Delphin system in Chapter 6. We will demonstrate examples introduced in preceding sections and discuss additional and advanced examples.

We will provide a discussion of related work in Chapter 7 including our work on alternative systems. We will briefly discuss the logical programming alternative provided by Twelf, and discuss ways to enhance the usability of Twelf by a process known as *factoring*. Finally, we will discuss future work and conclude in Chapter 8 with a discussion of our contributions.

Chapter 2

Logical Frameworks

Logical frameworks are meta-languages used to represent information. They are used in virtually every program, every programming language, and every proof assistant. Languages such as ML and Haskell have a rich logical framework allowing one to express custom recursive datatypes. In these systems the function space used for representation is the same as the one used for computation. However, we have seen in Chapter 1 that these systems are not well-suited for utilizing higher-order abstract syntax (HOAS) as such attempts lead to exotic terms, i.e. typeable terms that do not correspond to any concrete term in the object language being encoded.

Therefore, the Delphin system is designed as a two level system cleanly separating representation from computation. The function space used for representation is inherently weak allowing us to reap the benefits of HOAS wile maintaining adequate encodings (i.e. no exotic terms). The computation-level function space is much stronger permitting case analysis and recursion over higher-order data.

This chapter discusses the representation level, which is the Edinburgh Logical Framework LF (Harper et al. 1993). The usefulness of LF as a means of representation has been evidenced by the success of the Twelf system (Pfenning and Schürmann

1998) which provides a logic-programming paradigm to reason about LF encodings. Besides using Twelf to execute logic programs, it provides the ability to assign *modes* to relations indicating input and output behavior. If a relation can be interpreted as a total function, it can then also be considered a proof. Alternatively, Delphin provides a functional paradigm where the user can directly write the function they intend.

In this chapter we will ease into the introduction of LF by first providing a simply-typed version, SimpleLF, which will also be the representation level used in our simply-typed Delphin (Chapter 3). We will then extend SimpleLF into full LF while giving example representations that will be used throughout this dissertation. It is important to note that our representation level is exactly LF without any modifications. Therefore, for a very thorough and detailed explanation of LF (including comparisons with other logical frameworks), we refer the reader to Pfenning (2001).

2.1 Simply-Typed Logical Framework (SimpleLF)

Church's simply-typed λ-calculus (Church 1940) is arguably the first logical framework that supports higher-order encodings, where the binding constructs of the object language (the information modeled) are expressed in terms of the binding constructs of the λ-calculus. In other words, it permits the representation of functions as functions. This deceptively simple idea allows for encodings of complex data structures without having to worry about the representation of variables, renamings, or substitutions that are prevalent in logic derivations, typing derivations, operational semantics, and more. However, it also requires programmers to internalize that the arguments passed to functions can be higher-order objects that are constructed according to the rules of the framework.

$$
\begin{array}{lllll}
\text{Types} & \boldsymbol{A}, \boldsymbol{B} & ::= & \boldsymbol{a} \mid \boldsymbol{A} \rightarrow \boldsymbol{B} \\
\text{Objects} & \boldsymbol{M}, \boldsymbol{N} & ::= & \boldsymbol{c} \mid \boldsymbol{x} \mid \boldsymbol{M}\ \boldsymbol{N} \mid \boldsymbol{\lambda x{:}A.}\ \boldsymbol{N} \\
\text{Signature} & \boldsymbol{\Sigma} & ::= & \cdot \mid \boldsymbol{\Sigma, a{:}type} \mid \boldsymbol{\Sigma, c{:}A} \\
\text{Context} & \boldsymbol{\Gamma} & ::= & \cdot \mid \boldsymbol{\Gamma, x{:}A}
\end{array}
$$

Figure 2.1: SimpleLF

$$
\frac{(\boldsymbol{c{:}A})\ \textbf{in}\ \boldsymbol{\Sigma}}{\boldsymbol{\Gamma} \vdash^{\text{lf}} \boldsymbol{c} : \boldsymbol{A}} \qquad \frac{(\boldsymbol{x{:}A})\ \textbf{in}\ \boldsymbol{\Gamma}}{\boldsymbol{\Gamma} \vdash^{\text{lf}} \boldsymbol{x} : \boldsymbol{A}}
$$

$$
\frac{\boldsymbol{\Gamma, x{:}A} \vdash^{\text{lf}} \boldsymbol{M} : \boldsymbol{B}}{\boldsymbol{\Gamma} \vdash^{\text{lf}} \boldsymbol{\lambda x{:}A.}\ \boldsymbol{M} : \boldsymbol{A} \rightarrow \boldsymbol{B}} \qquad \frac{\boldsymbol{\Gamma} \vdash^{\text{lf}} \boldsymbol{M} : \boldsymbol{A} \rightarrow \boldsymbol{B} \quad \boldsymbol{\Gamma} \vdash^{\text{lf}} \boldsymbol{N} : \boldsymbol{A}}{\boldsymbol{\Gamma} \vdash^{\text{lf}} \boldsymbol{M}\ \boldsymbol{N} : \boldsymbol{B}}
$$

Figure 2.2: SimpleLF Typing Rules

SimpleLF is Church's simply-typed λ-calculus extended with a *signature* $\boldsymbol{\Sigma}$. The functional programmer may interpret the signature as the collection of datatype declarations. The signature stores type and object constants (or constructors), which we write as $\boldsymbol{a}$ and $\boldsymbol{c}$, respectively. We present the syntactic categories of SimpleLF in Figure 2.1. Function types are written as $\boldsymbol{A} \rightarrow \boldsymbol{B}$. We write $\boldsymbol{x}$ for variables. Besides datatype constructors $\boldsymbol{c}$, the objects are just those of the simply-typed λ-calculus: variables $\boldsymbol{x}$, applications $\boldsymbol{M}\ \boldsymbol{N}$, and abstractions $\boldsymbol{\lambda x{:}A.}\ \boldsymbol{N}$. The typing rules are standard and presented in Figure 2.2.

Notice that SimpleLF contains no meaningful computational constructs such as recursion or case analysis. We take $\equiv_{\boldsymbol{\alpha\beta\eta}}$ (equality modulo α, β, and η) as the underlying notion of definitional equality between objects, which is defined by the following rewriting rules:

$$
\begin{array}{rcll}
\boldsymbol{\lambda x{:}A.}\ \boldsymbol{N} & \equiv_{\boldsymbol{\alpha}} & \boldsymbol{\lambda y{:}A.}\ \boldsymbol{N}[\boldsymbol{y}/\boldsymbol{x}] & \text{if } \boldsymbol{y} \text{ does not occur in } \boldsymbol{N} \\
(\boldsymbol{\lambda x{:}A.}\ \boldsymbol{N})\ \boldsymbol{M} & \equiv_{\boldsymbol{\beta}} & \boldsymbol{N}[\boldsymbol{M}/\boldsymbol{x}] & \\
\boldsymbol{M} & \equiv_{\boldsymbol{\eta}} & \boldsymbol{\lambda x{:}A.}\ (\boldsymbol{M}\ \boldsymbol{x}) & \text{if } \boldsymbol{x} \text{ does not occur in } \boldsymbol{M}
\end{array}
$$

Equivalence modulo α allows us to rename bound variables. Objects are in β-normal (or β-short) form when all redexes are reduced following the above β rule. Finally, η-long form indicates that all objects of functional type start with a $\boldsymbol{\lambda}$ following the above η rule. Objects in β-normal η-long form are in canonical form and also called canonical objects.

It is best *not* to consider this conversion of objects to canonical form as computation since in this dissertation computation refers to the manipulation of data via mechanisms such as case analysis and recursion. It is best to view objects equal modulo $\alpha\beta\eta$ as belonging to the same equivalence class, and the canonical form is the representative we choose to distinguish between different equivalence classes. Therefore, when we represent object languages using LF, we are mapping different terms of the object language into different equivalence classes in LF. When we represent an object language we are interested that there is an isomorphism between terms in the object language and canonical forms in the encoding.

Encodings consist of a signature and a representation function, which maps elements from our domain of discourse into canonical forms in our logical framework. We say that an encoding is *adequate* if the representation function, $\ulcorner - \urcorner$, is a compositional bijection (one that commutes with substitution). We next present encodings, which will be used throughout this dissertation.

Example 2.1.1 (Natural Numbers)**.**

We start with an encoding of natural numbers $\mathbb{N} ::= 0, 1, 2, 3, \ldots$ *where we represent numbers* $\mathbb{N}$ *as objects of type* **nat**.

$$\begin{array}{rcl} \ulcorner 0 \urcorner & = & \mathbf{z} \\ \ulcorner n+1 \urcorner & = & \mathbf{s} \ulcorner n \urcorner \end{array} \qquad\qquad \begin{array}{lcl} \mathbf{nat} & : & \boldsymbol{type} \\ \mathbf{z} & : & \mathbf{nat} \\ \mathbf{s} & : & \mathbf{nat} \rightarrow \mathbf{nat} \end{array}$$

The signature is written to the right of the encoding. Here **nat** is a type constant

and the object constants (constructors) are **z** and **s**, all of which are saved in $\boldsymbol{\Sigma}$. It is easy to see this is an adequate encoding. Every natural number n can be encoded, and every object $\boldsymbol{M}$**:nat** corresponds to a real natural number m. For example, $\ulcorner 3 \urcorner = \mathbf{s}\ (\mathbf{s}\ (\mathbf{s}\ \mathbf{z}))$.

Example 2.1.2 (Untyped λ-Expressions Represented with HOAS)**.**

For an example with HOAS, consider the untyped λ-calculus $e ::= x \mid \lambda x.\ e \mid e_1\ e_2$.

Here we represent a λ-expression e as an object of type **exp** *as follows:*

$$
\begin{array}{lll}
 & \mathbf{exp} & \mathbf{:}\ \boldsymbol{type} \\
\ulcorner x \urcorner = \boldsymbol{x} & & \\
\ulcorner \lambda x.\ e \urcorner = \mathbf{lam}\ (\boldsymbol{\lambda x}\mathbf{:exp.}\ \ulcorner e \urcorner) & \mathbf{lam} & \mathbf{:}\ (\mathbf{exp} \to \mathbf{exp}) \to \mathbf{exp} \\
\ulcorner e_1\ e_2 \urcorner = \mathbf{app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner & \mathbf{app} & \mathbf{:}\ \mathbf{exp} \to \mathbf{exp} \to \mathbf{exp}
\end{array}
$$

In this example, we represent object-level variables x by LF variables $\boldsymbol{x}$ of type **exp**, which is recorded in the type of **lam**. Some example encodings are:

$$
\begin{aligned}
\ulcorner \lambda x.\ \lambda y.\ x \urcorner &= \mathbf{lam}\ (\boldsymbol{\lambda x}\mathbf{:exp.}\ \mathbf{lam}\ (\boldsymbol{\lambda y}\mathbf{:exp.}\ \boldsymbol{x})) \\
\ulcorner \lambda x.\ \lambda y.\ x\ y \urcorner &= \mathbf{lam}\ (\boldsymbol{\lambda x}\mathbf{:exp.}\ \mathbf{lam}\ (\boldsymbol{\lambda y}\mathbf{:exp.}\ \mathbf{app}\ \boldsymbol{x}\ \boldsymbol{y}))
\end{aligned}
$$

Theorem 2.1.3 (Adequacy)**.**

1. *There exists a bijection between λ-expressions e with free variables among $x_1 \ldots x_n$ and canonical derivations of $\boldsymbol{x_1}$**:exp** $\ldots$ $\boldsymbol{x_n}$**:exp** $\vdash^{\mathsf{lf}}$ $\boldsymbol{M}$ **: exp**.*

2. *The representation function $\ulcorner - \urcorner$ is a* compositional bijection *in the sense that $\ulcorner e_2\ [e_1/x] \urcorner = \ulcorner e_2 \urcorner\ [\ulcorner e_1 \urcorner / \boldsymbol{x}]$. Therefore, substitution is just LF application from the β rule as $\ulcorner e_2 \urcorner\ [\ulcorner e_1 \urcorner / \boldsymbol{x}] = (\boldsymbol{\lambda x}\mathbf{:exp.}\ \ulcorner e_2 \urcorner)\ \ulcorner e_1 \urcorner$.*

Proof. By induction on the structure of e in one direction (for compositionality as well) and the structure of the canonical forms (β-normal η-long) of **exp** in the other.

In Section 2.2 we will present a precise judgment defining what it means to be in canonical form. □

Example 2.1.4 (λ-Expressions using de Bruijn Indices (Simply-Typed Version))**.** *For comparison purposes, we also present a first-order encoding of the untyped λ-calculus* $e ::= x \mid \lambda x.\, e \mid e_1\ e_2$ *.*

A de Bruijn encoding (de Bruijn 1972) represents variables as pointers to their bindings (i.e. if the variable is represented as 2 *, then it refers to the second innermost binding). The complication of this notation is that the same variable can be represented differently depending on where it occurs. The representation of variables depends on the* distance *to the binding, which we model as a function* f *, which parameterizes the encoding as* $\ulcorner e \urcorner_f$ *, defined as follows:*

$$
\begin{array}{lll}
 & \textbf{variable} & : \ \textbf{\textit{type}} \\
\ulcorner 1 \urcorner = \textbf{one} & \textbf{one} & : \ \textbf{variable} \\
\ulcorner n+1 \urcorner = \textbf{succ}\ \ulcorner n \urcorner & \textbf{succ} & : \ \textbf{variable} \longrightarrow \textbf{variable} \\
 & \textbf{term} & : \ \textbf{\textit{type}} \\
\ulcorner x \urcorner_f = \textbf{var'}\ \ulcorner f(\,x\,) \urcorner & \textbf{var'} & : \ \textbf{variable} \longrightarrow \textbf{term} \\
\ulcorner \lambda x.\, e \urcorner_f = \textbf{lam'}\ \ulcorner e \urcorner_g, \ \textit{where} & \textbf{lam'} & : \ \textbf{term} \longrightarrow \textbf{term} \\
\quad g(y) = \begin{cases} 1 & \textit{if } x = y \\ f(y)+1 & \textit{otherwise} \end{cases} & & \\
\ulcorner e_1\ e_2 \urcorner_f = \textbf{app'}\ \ulcorner e_1 \urcorner_f\ \ulcorner e_2 \urcorner_f & \textbf{app'} & : \ \textbf{term} \longrightarrow \textbf{term} \longrightarrow \textbf{term}
\end{array}
$$

For comparison, we revisit the sample encodings from the HOAS version where we define $init$ to be an empty (undefined) function:

$$
\begin{aligned}
\ulcorner\, \lambda x.\, \lambda y.\, x \,\urcorner_{init} &= \textbf{lam' (lam'} \ulcorner\, 2 \,\urcorner\textbf{)} \\
&= \textbf{lam' (lam' (succ one))} \\
\ulcorner\, \lambda x.\, \lambda y.\, x\ y \,\urcorner_{init} &= \textbf{lam' (lam' (app'} \ulcorner\, 2 \,\urcorner \ulcorner\, 1 \,\urcorner\textbf{))} \\
&= \textbf{lam' (lam' (app' (succ one) (one)))}
\end{aligned}
$$

This is indeed an adequate encoding, but we do not get any substitution property for free. Instead, one must write substitution functions manually, which are easier in this representation than if we represented variables using strings. However, notice that there is a cost. The representation of λ is relatively complicated – every time we traverse a new binder, the new variable is mapped to 1 and all other mappings are incremented by one. De Bruijn notation avoids clashes in the representation of variables by updating all variable mappings every time one encounters a new binder.

Example 2.1.5 (Untyped Combinators)**.**

Our final example in this section is an encoding of untyped combinators $\mathcal{C}$*, where* $\mathcal{C} ::= K \mid S \mid \mathcal{C}_1\, \mathcal{C}_2$*. Combinators provide a Turing-complete language just as the untyped* λ*-calculus, but without bindings. In this dissertation, we will show how to convert between the two.*

Here we will represent a $\mathcal{C}$ *as an object of type* **comb** *as follows:*

$$
\begin{array}{lll}
 & \textbf{comb} & \textbf{:}\ \textbf{\textit{type}} \\
\ulcorner\, K \,\urcorner = \textbf{K} & \textbf{K} & \textbf{:}\ \textbf{comb} \\
\ulcorner\, S \,\urcorner = \textbf{S} & \textbf{S} & \textbf{:}\ \textbf{comb} \\
\ulcorner\, \mathcal{C}_1\, \mathcal{C}_2 \,\urcorner = \textbf{MP} \ulcorner\, \mathcal{C}_1 \,\urcorner \ulcorner\, \mathcal{C}_2 \,\urcorner & \textbf{MP} & \textbf{:}\ \textbf{comb} \rightarrow \textbf{comb} \rightarrow \textbf{comb}
\end{array}
$$

Combinators K *and* S *can be thought of as higher-order functions defined as:*

$$\begin{aligned} K\ x\ y &= x \\ S\ x\ y\ z &= (x\ z)\ (y\ z) \end{aligned}$$

$K\ x$ *creates a* constant *function that will always return* x*. Similarly,* $S\ x\ y$ *creates a function that will apply both* x *and* y *to its argument and then apply the results together. We name application* **MP** *for* modus ponens*.*

We have presented the logical framework SimpleLF. This language is well-suited for representing languages with bindings, but it falls short of being able to represent judgments. This will be the framework used in Chapter 3, when we discuss our simply-typed version of Delphin. Next we extend SimpleLF with dependent types, yielding the Edinburgh Logical Framework LF (Harper et al. 1993).

2.2 The Edinburgh Logical Framework (Dependently Typed)

Dependent types can capture invariants about representations that are impossible to express using just simple types. In Chapter 1 we saw an example where a list type could be index by its length, leading to the ability to write list functions with more static guarantees than one can find in traditional languages. For example, a function returning the head of a list can be statically verified to only be called on nonempty lists. Even more importantly, we can use dependent types to adequately represent judgments, such as derivations in logics, operational semantics, typing rules, etc. For example, in its original formulation, proof carrying code employed LF to represent proofs (Necula 1997).

Types	$\boldsymbol{A}, \boldsymbol{B}$	::=	$\boldsymbol{a} \mid \boldsymbol{A}\ \boldsymbol{M} \mid \boldsymbol{\Pi x{:}A.}\ \boldsymbol{B}$
Objects	$\boldsymbol{M}, \boldsymbol{N}$	::=	$\boldsymbol{x} \mid \boldsymbol{c} \mid \boldsymbol{M}\ \boldsymbol{N} \mid \boldsymbol{\lambda x{:}A.}\ \boldsymbol{N}$
Signature	$\boldsymbol{\Sigma}$	::=	$\cdot \mid \boldsymbol{\Sigma, a{:}K} \mid \boldsymbol{\Sigma, c{:}A}$
Kinds	$\boldsymbol{K}$	::=	$\boldsymbol{type} \mid \boldsymbol{\Pi x{:}A.}\ \boldsymbol{K}$
Context	$\boldsymbol{\Gamma}$	::=	$\cdot \mid \boldsymbol{\Gamma, x{:}A}$

Figure 2.3: The Edinburgh Logical Framework LF

The Edinburgh Logical Framework LF (Harper et al. 1993) extends SimpleLF with dependent types. LF is often used for working with proof theories and type systems. Recently, it has been used for representing the typed intermediate language of SML/NJ (Lee et al. 2007) as well as the base logics of proof assistants, such as HOL and Nuprl (Schürmann and Stehr 2006), which rely heavily on dependent types and higher-order encodings.

We present the syntactic categories of LF in Figure 2.3. Function types assign names to their arguments in $\boldsymbol{\Pi x{:}A.}\ \boldsymbol{B}$. We write $\boldsymbol{A} \rightarrow \boldsymbol{B}$ as syntactic sugar (reminiscent of SimpleLF) when $\boldsymbol{x}$ does not occur in $\boldsymbol{B}$. Types may be indexed by objects and we provide the construct $\boldsymbol{A}\ \boldsymbol{M}$ to represent such types. We write $\boldsymbol{x}$ for variables while $\boldsymbol{a}$ and $\boldsymbol{c}$ are type and object constants (or constructors), respectively. We often refer to $\boldsymbol{a}$ as a *type family*. These constants are provided a priori in the signature $\boldsymbol{\Sigma}$. Unlike SimpleLF, type constants are not all of the form $\boldsymbol{a{:}type}$, but rather we introduce a *kind* system K, allowing types to be indexed by objects via $\boldsymbol{\Pi x{:}A.}\ \boldsymbol{K}$, where we also write $\boldsymbol{A} \rightarrow \boldsymbol{K}$ as syntactic sugar when $\boldsymbol{x}$ does not occur in $\boldsymbol{K}$. For example, a list type indexed by its length can be represented as a type family $\mathbf{list{:}\ nat} \rightarrow \boldsymbol{type}$. Now, **list** itself is not a type, but **list z** or **list (s z)** are types representing empty lists and one element lists, respectively.

In the presence of dependencies, not all types are valid. The *kind* system of LF acts as a type system for types. We write $\boldsymbol{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{M} : \boldsymbol{A}$ for valid objects and $\boldsymbol{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{A} : \boldsymbol{K}$ for valid types, in a context $\boldsymbol{\Gamma}$ that assigns types to variables. The typing

$$\frac{(c{:}A)\ \mathbf{in}\ \Sigma}{\Gamma \vdash^{\mathrm{lf}} c : A} \qquad \frac{(x{:}A)\ \mathbf{in}\ \Gamma}{\Gamma \vdash^{\mathrm{lf}} x : A} \qquad \frac{\Gamma, x{:}A \vdash^{\mathrm{lf}} M : B}{\Gamma \vdash^{\mathrm{lf}} \lambda x{:}A.\ M : \Pi x{:}A.\ B}$$

$$\frac{\Gamma \vdash^{\mathrm{lf}} M : \Pi x{:}A.\ B \quad \Gamma \vdash^{\mathrm{lf}} N : A}{\Gamma \vdash^{\mathrm{lf}} M\ N : B[N/x]}$$

$$\frac{(a{:}K)\ \mathbf{in}\ \Sigma}{\Gamma \vdash^{\mathrm{lf}} a : K} \qquad \frac{\Gamma, x{:}A \vdash^{\mathrm{lf}} B : \mathit{type}}{\Gamma \vdash^{\mathrm{lf}} \Pi x{:}A.\ B : \mathit{type}} \qquad \frac{\Gamma \vdash^{\mathrm{lf}} A : \Pi x{:}B.\ K \quad \Gamma \vdash^{\mathrm{lf}} M : B}{\Gamma \vdash^{\mathrm{lf}} A\ M : K[M/x]}$$

$$\frac{\Gamma \vdash^{\mathrm{lf}} M : A \quad A \equiv_{\alpha\beta\eta} B}{\Gamma \vdash^{\mathrm{lf}} M : B} \qquad \frac{\Gamma \vdash^{\mathrm{lf}} A : K \quad K \equiv_{\alpha\beta\eta} K'}{\Gamma \vdash^{\mathrm{lf}} A : K'}$$

Figure 2.4: LF Typing and Kinding Rules

and kinding rules of LF are summarized in Figure 2.4. There are implicit assumptions that the context is well-formed, $\Gamma\ \mathsf{ctx}$, which means that for all $x{:}A$ in Γ, it holds that $\Gamma \vdash^{\mathrm{lf}} A : \mathit{type}$.[1] All LF judgments enjoy the usual weakening and substitution properties on their respective contexts, but exchange is only permitted in limited form due to dependencies. Just as in SimpleLF, we take $\equiv_{\alpha\beta\eta}$ as the underlying notion of definitional equality between objects and now additionally between types. Unlike SimpleLF, as LF objects M may occur inside types A and kinds K, the typing rules are defined modulo such equality. We say objects and types in β-normal η-long form are in canonical form. Note that the implicit well-formedness conditions are made formal in Appendix A.1.

2.2.1 Canonical Forms

In Section 2.1 we demonstrated that canonical forms are necessary for proving the adequacy of our encodings (Theorem 2.1.3). Informally, canonical forms are terms con-

[1] More precisely, in the context $\Gamma, x{:}A, \Gamma'$, it holds that $\Gamma \vdash^{\mathrm{lf}} A : \mathit{type}$.

verted to β-normal η-long form. Formally, we express this property in two mutually dependent judgments determining if objects and types are *canonical* and if they are *atomic* (Schürmann 2000). Canonical forms have the form $\boldsymbol{\lambda x_1{:}A_1 \ldots \lambda x_n{:}A_n.\ M^a}$ where $\boldsymbol{M^a}$ is atomic. Atomic forms have the form $\boldsymbol{h\ M_1^c\ \ldots\ M_n^c}$ where $\boldsymbol{h}$ is either a variable or a constant and all $\boldsymbol{M_i^c}$'s are canonical. To guarantee η-long forms, the type of $\boldsymbol{M^a}$ must be atomic (i.e. not a functional type). For completeness, we also formalize canonical types, which enforce that all objects occurring as indices in type families are also canonical.

Judgments

Canonical objects:	$\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \Uparrow A}$
Atomic objects:	$\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \downarrow A}$
Canonical types:	$\boldsymbol{\Gamma \vdash^{\mathrm{f}} A \Uparrow type}$
Atomic types:	$\boldsymbol{\Gamma \vdash^{\mathrm{f}} A \downarrow K}$

Rules

$$\frac{\boldsymbol{\Gamma \vdash^{\mathrm{f}} A \Uparrow type \quad \Gamma, x{:}A \vdash^{\mathrm{f}} M \Uparrow B}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} \lambda x{:}A.\ M \Uparrow \lambda x{:}A.\ B}} \qquad \frac{\boldsymbol{\Gamma \vdash^{\mathrm{f}} A \downarrow type \quad \Gamma \vdash^{\mathrm{f}} M \downarrow A}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \Uparrow A}}$$

$$\frac{\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \Uparrow A \quad A \equiv_{\alpha\beta\eta} B \quad \Gamma \vdash^{\mathrm{f}} B : type}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \Uparrow B}}$$

$$\frac{\boldsymbol{\Sigma(c) = A}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} c \downarrow A}} \qquad \frac{\boldsymbol{\Gamma(x) = A}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} x \downarrow A}} \qquad \frac{\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \downarrow \Pi x{:}B.\ A \quad \Gamma \vdash^{\mathrm{f}} N \Uparrow B}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} M\ N \downarrow A[N/x]}}$$

$$\frac{\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \downarrow A \quad A \equiv_{\alpha\beta\eta} B \quad \Gamma \vdash^{\mathrm{f}} B : type}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} M \downarrow B}}$$

$$\frac{\boldsymbol{\Gamma \vdash^{\mathrm{f}} A \Uparrow type \quad \Gamma, x{:}A \vdash^{\mathrm{f}} B \Uparrow type}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} \Pi x{:}A.\ B \Uparrow type}} \qquad \frac{\boldsymbol{\Gamma \vdash^{\mathrm{f}} A \downarrow type}}{\boldsymbol{\Gamma \vdash^{\mathrm{f}} A \Uparrow type}}$$

$$\frac{\Sigma(a) = K}{\Gamma \vdash^{\mathrm{lf}} a \downarrow K} \qquad \frac{\Gamma \vdash^{\mathrm{lf}} A \downarrow \Pi x{:}A'.\ K \quad \Gamma \vdash^{\mathrm{lf}} M \Uparrow A'}{\Gamma \vdash^{\mathrm{lf}} A\ M \downarrow K[M/x]}$$

$$\frac{\Gamma \vdash^{\mathrm{lf}} A \downarrow K \quad K \equiv_{\alpha\beta\eta} K' \quad \Gamma \vdash^{\mathrm{lf}} K : \mathit{kind}}{\Gamma \vdash^{\mathrm{lf}} A \downarrow K}$$

Theorem 2.2.1 (Canonical Forms)**.**

1. *If* $\Gamma \vdash^{\mathrm{lf}} M \Uparrow A$*, then* $\Gamma \vdash^{\mathrm{lf}} M : A$*.*

2. *Every well-typed object* M *such that* $\Gamma \vdash^{\mathrm{lf}} M : A$ *possesses a unique canonical form* N *(modulo* α*-renaming) such that* $M \equiv_{\alpha\beta\eta} N$ *and* $\Gamma \vdash^{\mathrm{lf}} N \Uparrow A$*. Additionally,* N *is computable.*

3. *Every type* A *such that* $\Gamma \vdash^{\mathrm{lf}} A : \mathit{type}$ *possesses a unique canonical form* A' *(modulo* α*-renaming) such that* $A \equiv_{\alpha\beta\eta} A'$ *and* $\Gamma \vdash^{\mathrm{lf}} A \Uparrow \mathit{type}$*. Additionally,* A' *is computable.*

Proof. See Harper and Pfenning (2005). □

The existence of unique canonical forms is crucial for LF's support of higher-order encodings and is what makes it well-suited for representations. All adequacy proofs must be done on paper, but the design of the system allows one to represent languages very closely to how they appear on paper, making such proofs easy. The judgment $\Gamma \vdash^{\mathrm{lf}} M \Uparrow A$ gives the programmer the ability to induct over canonical forms, which is necessary for one direction of adequacy proofs (as in Theorem 2.1.3).

Due to this theorem we are justified, without loss of generality, to assume throughout this dissertation that every LF term is kept in canonical form. It is also worth mentioning that there are alternative presentations of LF that always keep terms in canonical form by a special handling of substitutions, called *hereditary substitutions* (Watkins et al. 2002). With this alternative presentation, we would not need this theorem, but it would complicate the presentation of the system.

As in SimpleLF, encodings consist of a signature and a representation function $\ulcorner - \urcorner$, which maps elements from our domain of discourse into canonical forms in our logical framework. All the simply-typed examples in Section 2.1 are still valid in LF. We now show examples utilizing dependent types, namely following the *judgments-as-types* paradigm.

2.2.2 Examples Representing Judgments

We will present two calculi for proving the validity of formulas. Therefore, we first start with the definition of formulas we will use.

Example 2.2.2 (Formulas)**.**

Let $A, B ::= A \supset B \mid p$ *be the language of formulas where* p *is a base proposition (type). Formulas will be represented as objects of type* $\mathbf{o}$*. We will use* $\Rightarrow$ *as a right-associative infix operator below.*

$$\begin{array}{ll} \ulcorner A \supset B \urcorner = \ulcorner A \urcorner \Rightarrow \ulcorner B \urcorner & \mathbf{o} \;:\; \textit{type} \\ \ulcorner p \urcorner = \mathbf{p} & \Rightarrow \;:\; \mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o} \\ & \mathbf{p} \;:\; \mathbf{o} \end{array}$$

Notice that we just have arrow types with one base type p. It is important to point out that LF variables of type $\mathbf{o}$ can also serve to represent propositions, similar to our methodology of representing object-level variables for use in higher-order encodings. However, for simplicity we only consider one concrete base type $\boldsymbol{p}$.

Example 2.2.3 (Natural Deduction Calculus)**.**

Formulas A, B *are as defined in Example 2.2.2 and recall that a formula* A *is represented as an object of type* $\mathbf{o}$*. We define a natural deduction calculus, limited to the implicational fragment of intuitionistic propositional logic, by the following two*

inference rules:

$$\dfrac{\dfrac{}{\vdash^{\mathsf{n}} A}\,u \\ \vdots \\ \vdash^{\mathsf{n}} B}{\vdash^{\mathsf{n}} A \supset B}\,\mathsf{impi}_u \qquad \dfrac{\vdash^{\mathsf{n}} A \supset B \quad \vdash^{\mathsf{n}} A}{\vdash^{\mathsf{n}} B}\,\mathsf{impe}$$

We represent derivations in this calculus following the judgments-as-types *paradigm. Objects of type* $\mathbf{nd}\ulcorner A \urcorner$ *will represent derivations of* $\vdash^{\mathsf{n}} A$. *In other words, derivation trees* $\mathcal{E}$ *of* $\vdash^{\mathsf{n}} A$ *are represented in LF as objects of type* $\mathbf{nd}\ulcorner A \urcorner$, *whose signature is:*

$$\begin{array}{lcl} \mathbf{nd} & : & \mathbf{o} \rightarrow \textit{type} \\ \\ \mathbf{impi} & : & \mathbf{\Pi}\boldsymbol{A}\mathbf{:o.}\ \mathbf{\Pi}\boldsymbol{B}\mathbf{:o.}\ (\mathbf{nd}\ \boldsymbol{A} \rightarrow \mathbf{nd}\ \boldsymbol{B}) \rightarrow \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \\ \mathbf{impe} & : & \mathbf{\Pi}\boldsymbol{A}\mathbf{:o.}\ \mathbf{\Pi}\boldsymbol{B}\mathbf{:o.}\ \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \rightarrow \mathbf{nd}\ \boldsymbol{A} \rightarrow \mathbf{nd}\ \boldsymbol{B} \end{array}$$

The representation of the impi rule shows an example of HOAS in the encoding of judgments, which corresponds to representing the *hypothetical judgment* named u. For example, consider the following derivation of $\vdash^{\mathsf{n}} A \supset B \supset A$:

$$\mathcal{E} = \dfrac{\dfrac{\dfrac{}{\vdash^{\mathsf{n}} A}\,u \qquad \dfrac{}{\vdash^{\mathsf{n}} B}\,u' \\ \vdots \\ \dfrac{}{\vdash^{\mathsf{n}} A}\,u}{\vdash^{\mathsf{n}} B \supset A}\,\mathsf{impi}_{u'}}{\vdash^{\mathsf{n}} A \supset (B \supset A)}\,\mathsf{impi}_u$$

Note that we index the use of impi by the name of the hypothetical judgment introduced, either u or u' in this example. Furthermore, without loss of generality, we will assume that $\ulcorner A \urcorner = \boldsymbol{A}_*$ and $\ulcorner B \urcorner = \boldsymbol{B}_*$, which entails that $\ulcorner A \supset B \supset A \urcorner = \boldsymbol{A}_* \Rightarrow \boldsymbol{B}_* \Rightarrow \boldsymbol{A}_*$. Therefore, we will represent $\mathcal{E}$ as (or equivalently $\ulcorner \mathcal{E} \urcorner$ is) an

$\boldsymbol{M}$ of type $\mathbf{nd}\ \boldsymbol{A}_* \Rightarrow \boldsymbol{B}_* \Rightarrow \boldsymbol{A}_*$ where $\boldsymbol{M}$ is:

$$\mathbf{impi}\ \boldsymbol{A}_*\ (\boldsymbol{B}_* \Rightarrow \boldsymbol{A}_*)\ (\boldsymbol{\lambda u}{:}\mathbf{nd}\ \boldsymbol{A}_*.\ \mathbf{impi}\ \boldsymbol{B}_*\ \boldsymbol{A}_*\ (\boldsymbol{\lambda u'}{:}\mathbf{nd}\ \boldsymbol{B}_*.\ \ \boldsymbol{u}))$$

This illustrates the power and usefulness of both dependent types and HOAS. However, one may also notice that parts of this encoding seem redundant. Namely, the first two arguments of **impi** and **impe** are inferable by their third argument. Therefore, as a notational convenience, one may elect to omit the leading $\mathbf{\Pi}$s from the types when they are inferable. This is, for example, common practice in Twelf (Pfenning and Schürmann 1998). Therefore, the signature we actually write will be:

$$\begin{array}{lcl} \mathbf{nd} & : & \mathbf{o} \longrightarrow \boldsymbol{type} \\ \\ \mathbf{impi} & : & (\mathbf{nd}\ \boldsymbol{A} \longrightarrow \mathbf{nd}\ \boldsymbol{B}) \longrightarrow \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \\ \mathbf{impe} & : & \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \longrightarrow \mathbf{nd}\ \boldsymbol{A} \longrightarrow \mathbf{nd}\ \boldsymbol{B} \end{array}$$

It is crucial to emphasize that this is just a notational enhancement. Although the variables $\boldsymbol{A}$ and $\boldsymbol{B}$ appear to occur free, they are implicitly $\mathbf{\Pi}$-quantified and have the same type we saw earlier. These implicit arguments are filled in automatically when applied. For example, the $\boldsymbol{M}$ above would now be written more concisely as $\mathbf{impi}\ (\boldsymbol{\lambda u}{:}\mathbf{nd}\ \boldsymbol{A}_*.\ \mathbf{impi}\ (\boldsymbol{\lambda u'}{:}\mathbf{nd}\ \boldsymbol{B}_*.\ \ \boldsymbol{u}))$.

Example 2.2.4 (Hilbert-Style Calculus (A.K.A. Combinators))**.**

Formulas A, B *are as defined in Example 2.2.2 and recall that a formula* A *is represented as an object of type* $\mathbf{o}$*. We now define a Hilbert-style calculus, which is an* axiomatic *logic which does not have variable bindings. This extends the untyped combinators in Section 2.1 with types.*

Derivations in this calculus consist of two combinators with modus ponens:

$$\frac{}{\vdash\!\!\!\natural\ A \supset B \supset A}\,\mathsf{K} \qquad \frac{}{\vdash\!\!\!\natural\ (A \supset B \supset C) \supset (A \supset B) \supset A \supset C}\,\mathsf{S} \qquad \frac{\vdash\!\!\!\natural\ A \supset B \quad \vdash\!\!\!\natural\ A}{\vdash\!\!\!\natural\ B}\,\mathsf{MP}$$

Following the judgments-as-types *paradigm, we encode derivations* $\mathcal{E}$ *of* $\vdash A$ *as objects of type* $\mathbf{comb}\ \ulcorner A \urcorner$ *as follows:*

$$
\begin{array}{lcl}
\mathbf{comb} & : & \mathbf{o} \rightarrow \textit{type} \\
 & & \\
\mathbf{K} & : & \mathbf{comb}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B} \Rightarrow \boldsymbol{A}) \\
\mathbf{S} & : & \mathbf{comb}\ ((\boldsymbol{A} \Rightarrow \boldsymbol{B} \Rightarrow \boldsymbol{C}) \Rightarrow (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \Rightarrow \boldsymbol{A} \Rightarrow \boldsymbol{C}) \\
\mathbf{MP} & : & \mathbf{comb}(\boldsymbol{A} \Rightarrow \boldsymbol{B}) \rightarrow \mathbf{comb}\ \boldsymbol{A} \rightarrow \mathbf{comb}\ \boldsymbol{B}
\end{array}
$$

Recall that all free variables are implicitly Π-quantified as discussed in the previous example. In Chapter 1 we saw how to do this in Haskell with GADTs, and we will show the same example derivation here. A derivation $\mathcal{E}$ of $\vdash A \supset A$ is:

$$
\dfrac{\dfrac{\dfrac{}{\vdash (A \supset (A \supset A) \supset A) \supset (A \supset A \supset A) \supset A \supset A}\,\mathsf{S} \quad \dfrac{}{\vdash A \supset (A \supset A) \supset A}\,\mathsf{K}}{\vdash (A \supset A \supset A) \supset A \supset A}\,\mathsf{MP} \quad \dfrac{}{\vdash A \supset A \supset A}\,\mathsf{K}}{\vdash A \supset A}\,\mathsf{MP}
$$

Assuming $\ulcorner A \urcorner = \boldsymbol{A}_*$, the derivation $\mathcal{E}$ is represented in LF as $\mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \mathbf{K})\ \mathbf{K}$, which has type $\mathbf{comb}\ (\boldsymbol{A}_* \Rightarrow \boldsymbol{A}_*)$. One should notice that the type of the second $\mathbf{K}$ is $\mathbf{comb}\ (\boldsymbol{A}_* \Rightarrow \boldsymbol{A}_* \Rightarrow \boldsymbol{A}_*)$, which is hidden information since we made the $\boldsymbol{B}$ implicit in the definition of $\mathbf{K}$, which can be made explicit if we desired.

It is important to note that because of our use of implicit Π-quantification, the object $\mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \mathbf{K})\ \mathbf{K}$ does not contain enough information for us to know exactly what derivation it corresponds to. The reason is that the second $\mathbf{K}$ can be of type $\mathbf{comb}\ (\boldsymbol{A}_* \Rightarrow \boldsymbol{X} \Rightarrow \boldsymbol{A}_*)$, for any $\boldsymbol{X}$. However, there is no ambiguity under-the-hood. In the fully expanded proof-term of $\mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \mathbf{K})\ \mathbf{K}$, the $\boldsymbol{X}$ is explicit. In order to avoid this ambiguity, we can make the $\boldsymbol{B}$ explicitly Π-quantified in the definition of $\mathbf{K}$, but we instead leave it implicitly quantified and indicate the $\boldsymbol{X}$ we desire using a type ascription, i.e. $\mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \mathbf{K})\ (\mathbf{K} : \mathbf{comb}\ (\boldsymbol{A}_* \Rightarrow \boldsymbol{X} \Rightarrow \boldsymbol{A}_*))$

2.2.3 Advanced: λ-Expressions in a Context

Recall that we presented two encodings of λ-expressions using both HOAS (Example 2.1.2) and de Bruijn indices (Example 2.1.4). We can talk about λ-expressions making sense with respect to some context Φ, where $\Phi ::= \cdot \mid \Phi, x$. We say a λ-expression e makes sense in context Φ if the context contains all the free variables in e.

In the HOAS version, this context Φ is represented as the LF context $\mathbf{\Gamma}$ as all variables accessed are variables $\boldsymbol{x}$**:exp** in $\mathbf{\Gamma}$.

However, there is no representation of Φ in the de Bruijn version (Example 2.1.4). We therefore extend our de Bruijn encoding to be indexed by the context Φ in which the λ-expression makes sense. This will prove necessary in our conversion between de Bruijn and HOAS as we will need to save a map between free variables and indices – so we need to know exactly what variables may occur free.

Example 2.2.5 (Well-Formed λ-Expressions Using de Bruijn Indices)**.**
As we model variables as numbers $n \geq 1$*, we will represent the context* Φ *as a natural number (indicating its size) of type* **nat** *as defined in Examples 2.1.1. To be precise,* $\ulcorner \cdot \urcorner = \mathbf{z}$ *and* $\ulcorner \Phi, x \urcorner = \mathbf{s} \ulcorner \Phi \urcorner$*.*

Variables in Φ *are represented as objects of type* **variable** $\ulcorner \Phi \urcorner$*. Similarly,* λ*-expressions in* Φ *are represented as objects of type* **term** $\ulcorner \Phi \urcorner$*. The signature representing this refined grammar of* **variable** *and* **term** *is as follows:*

$$
\begin{array}{lcl}
\textbf{variable} & : & \textbf{nat} \rightarrow \textbf{\textit{type}} \\
\textbf{one} & : & \textbf{variable (s } \boldsymbol{P}\textbf{)} \\
\textbf{succ} & : & \textbf{variable } \boldsymbol{P} \rightarrow \textbf{variable (s } \boldsymbol{P}\textbf{)} \\
 & & \\
\textbf{term} & : & \textbf{nat} \rightarrow \textbf{\textit{type}} \\
\textbf{var'} & : & \textbf{variable } \boldsymbol{P} \rightarrow \textbf{term } \boldsymbol{P} \\
\textbf{lam'} & : & \textbf{term (s } \boldsymbol{P}\textbf{)} \rightarrow \textbf{term } \boldsymbol{P} \\
\textbf{app'} & : & \textbf{term } \boldsymbol{P} \rightarrow \textbf{term } \boldsymbol{P} \rightarrow \textbf{term } \boldsymbol{P}
\end{array}
$$

One would notice that our new grammar is very similar to our original grammar without contexts (Example 2.1.4). Objects are constructed via **var'**, **lam'**, and **app'**. The constant **app'** states that an application in Φ is constructed by two λ-expressions in Φ. The most interesting case is **lam'** which states that an abstraction in Φ is constructed by a λ-expression that makes sense in Φ, x. Variable **one** is accessible in all nonempty contexts, and the rest of the constants are similar.

Example 2.2.6 (Equality of Untyped λ-Expressions (between HOAS and de Bruijn))**.**

We next encode a judgment to determine if two λ-expressions e are equal. This judgment is defined as:

$$\frac{}{\Phi, x \vdash x \equiv x}\, u \quad \vdots$$

$$\frac{\Phi \vdash x \equiv x}{\Phi, y \vdash x \equiv x}\,\mathsf{eqVar} \qquad \frac{\Phi, x \vdash e \equiv f}{\Phi \vdash \lambda x.\, e \equiv \lambda x.\, f}\,\mathsf{eqLam}_u \qquad \frac{\Phi \vdash e_1 \equiv f_1 \quad \Phi \vdash e_2 \equiv f_2}{\Phi \vdash e_1\ e_2 \equiv f_1\ f_2}\,\mathsf{eqApp}$$

We will use this judgment to reason about the equality between λ-expressions represented in HOAS (Example 2.1.2) and λ-expressions represented a la de Bruijn (Example 2.2.5). Following the judgments-as-types *paradigm, we encode derivations $\mathcal{E}$ of $\Phi \vdash e \equiv f$ as objects of type* **equiv** $\ulcorner \Phi \urcorner\ \ulcorner e \urcorner\ \ulcorner f \urcorner$ *where we use the HOAS encoding for $\ulcorner e \urcorner$ and the de Bruijn encoding for $\ulcorner f \urcorner$. In the HOAS encoding, the context Φ is implicitly the LF context. In the de Bruijn encoding, the context Φ is represented as a natural number (as in Example 2.2.5). For clarity we make the context* **P : nat** *explicit, but we could have made it implicit as it occurs in the third argument.*

$$
\begin{array}{lcl}
\textbf{equiv} & : & \textbf{Π}\boldsymbol{P}\textbf{:nat. exp} \rightarrow (\textbf{term } \boldsymbol{P}) \rightarrow \textit{\textbf{type}} \\
\\
\textbf{eqVar} & : & \textbf{equiv } \boldsymbol{P}\ \boldsymbol{E}\ (\textbf{var' } \boldsymbol{X}) \\
 & & \quad \rightarrow \textbf{equiv } (\textbf{s } \boldsymbol{P})\ \boldsymbol{E}\ (\textbf{var' } (\textbf{succ } \boldsymbol{X})) \\
\\
\textbf{eqLam} & : & (\textbf{Π}\boldsymbol{x}\textbf{:exp. equiv } (\textbf{s } \boldsymbol{P})\ \boldsymbol{x}\ (\textbf{var' one}) \\
 & & \qquad\qquad \rightarrow \textbf{equiv } (\textbf{s } \boldsymbol{P})\ (\boldsymbol{E}\ \boldsymbol{x})\ \boldsymbol{T}) \\
 & & \quad \rightarrow \textbf{equiv } \boldsymbol{P}\ \ (\textbf{lam } (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}))\ (\textbf{lam' } \boldsymbol{T}) \\
\\
\textbf{eqApp} & : & \textbf{equiv } \boldsymbol{P}\ \boldsymbol{E_1}\ \boldsymbol{T_1} \\
 & & \quad \rightarrow \textbf{equiv } \boldsymbol{P}\ \boldsymbol{E_2}\ \boldsymbol{T_2} \\
 & & \quad \rightarrow \textbf{equiv } \boldsymbol{P}\ (\textbf{app } \boldsymbol{E_1}\ \boldsymbol{E_2})\ (\textbf{app' } \boldsymbol{T_1}\ \boldsymbol{T_2})
\end{array}
$$

There are a few important notes to make about this encoding. First, the **eqVar** states that if we have a variable $\boldsymbol{Y}$ in context Φ, then the same variable is represented as **succ** $\boldsymbol{Y}$ in context Φ, x. The most important case is **eqLam** which states that two λ-expressions are equivalent in Φ if their bodies are equivalent in Φ, x where a new variable x corresponds to de Bruijn index 1.

2.3 Programming with LF Objects

We have seen that one may use LF to elegantly encode simple datatypes as well as complicated judgments. However, how can one work with, i.e. manipulate, such objects? Unfortunately, we cannot extend LF with more computationally meaningful constructs such as case analysis and recursion as they would inevitably lead to exotic terms in our higher-order encodings (as shown in Chapter 1). Therefore, we define computation on a separate level.

Twelf is one such tool that follows a *logic programming* paradigm. With dependent types, the user can express relations on multiple arguments as a type family indexed by those arguments. Computation is limited to a nondeterministic proof search likened to that of Prolog. Twelf also provides the user with special tools to assign an input/output meaning to the arguments and determine if the execution of

a relation as a logic program will never fail (*coverage*) and always terminate. If it passes these checks, then one may interpret the relation as a proof that the existence of the inputs entails the existence of the outputs.

Alternatively, we define Delphin as a functional programming language to manipulate LF objects. In this paradigm, the input and output are clearly specified by the user, and one may interpret a function as a proof if the function is guaranteed to never fail (*coverage*) as well as to always terminate. Twelf relations can only be interpreted as a proof if the relation actually corresponds to a total function. Delphin offers the user the ability to explicitly write the function with a clear deterministic operational semantics, rather than just verifying that something is equivalent to a total function. Additionally, Delphin is itself higher order, allowing one to write functions that take Delphin functions as input. We will see many examples where we need this functionality. Twelf is not higher order and requires heavy use of a special notation of *blocks* (Schürmann 2000) to express meaningful functions. The higher-order nature of Delphin allows one to do without any notion of blocks. Additionally, the ability to write inner functions eliminates the need for manual *factoring* (Poswolsky and Schürmann 2003) of logic programs. We believe this all leads to a much more elegant methodology to manipulate and reason about LF encodings.

2.4 Summary of Datatypes

In this chapter, we have described and defined all the datatypes that we will use throughout this dissertation. We summarize them here.

2.4.1 SimpleDelphin (Simply Typed)

The signature $\mathbf{\Sigma}$ we will use in discussing simply-typed Delphin (Chapter 3) is:

1. Natural Numbers (Example 2.1.1)
   nat : *type*
   z : nat
   s : nat → nat

2. Untyped λ-expressions using HOAS (Example 2.1.2)
   exp : *type*
   lam : (exp → exp) → exp
   app : exp → exp → exp

3. Untyped λ-expressions a la de Bruijn (Example 2.1.4)

   (a) Variables
   variable : *type*
   one : variable
   succ : variable → variable

   (b) Expressions
   term : : *type*
   var' : variable → term
   lam' : term → term
   app' : term → term → term

4. Untyped Combinators (Example 2.1.5)
   comb : *type*
   K : comb
   S : comb
   MP : comb → comb → comb

2.4.2 Delphin (Dependently Typed)

For Delphin (Chapter 4 and beyond) we extend some of the previous encodings to use dependent types as well as add some additional datatypes, yielding the following signature $\mathbf{\Sigma}$:

1. Natural Numbers (Example 2.1.1)
   $\mathbf{nat} : \mathit{type}$
   $\mathbf{z} : \mathbf{nat}$
   $\mathbf{s} : \mathbf{nat} \rightarrow \mathbf{nat}$

2. Untyped λ-expressions using HOAS (Example 2.1.2)
   $\mathbf{exp} : \mathit{type}$
   $\mathbf{lam} : (\mathbf{exp} \rightarrow \mathbf{exp}) \rightarrow \mathbf{exp}$
   $\mathbf{app} : \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{exp}$

3. Formulas (Example 2.2.2)
   $\mathbf{o} : \mathit{type}$
   $\Rightarrow : \mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}$ **(right-associative infix)**

4. Natural Deduction (Example 2.2.3)
   $\mathbf{nd} : \mathbf{o} \rightarrow \mathit{type}$
   $\mathbf{impi} : (\mathbf{nd}\ \boldsymbol{A} \rightarrow \mathbf{nd}\ \boldsymbol{B}) \rightarrow \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})$
   $\mathbf{impe} : \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \rightarrow \mathbf{nd}\ \boldsymbol{A} \rightarrow \mathbf{nd}\ \boldsymbol{B}$

5. Combinators (Example 2.2.4)
   $\mathbf{comb} : \mathbf{o} \rightarrow \mathit{type}$
   $\mathbf{K} : \mathbf{comb}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B} \Rightarrow \boldsymbol{A})$
   $\mathbf{S} : \mathbf{comb}\ ((\boldsymbol{A} \Rightarrow \boldsymbol{B} \Rightarrow \boldsymbol{C}) \Rightarrow (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \Rightarrow \boldsymbol{A} \Rightarrow \boldsymbol{C})$
   $\mathbf{MP} : \mathbf{comb}(\boldsymbol{A} \Rightarrow \boldsymbol{B}) \rightarrow \mathbf{comb}\ \boldsymbol{A} \rightarrow \mathbf{comb}\ \boldsymbol{B}$

6. Well-formed untyped λ-expressions a la de Bruijn (Example 2.2.5)

   (a) Variables
   $\mathbf{variable} : \mathbf{nat} \rightarrow \mathit{type}$
   $\mathbf{one} : \mathbf{variable}\ (\mathbf{s}\ \boldsymbol{P})$
   $\mathbf{succ} : \mathbf{variable}\ \boldsymbol{P} \rightarrow \mathbf{variable}\ (\mathbf{s}\ \boldsymbol{P})$

   (b) Expressions
   $\mathbf{term} : \mathbf{nat} \rightarrow \mathit{type}$
   $\mathbf{var'} : \mathbf{variable}\ \boldsymbol{P} \rightarrow \mathbf{term}\ \boldsymbol{P}$
   $\mathbf{lam'} : \mathbf{term}\ (\mathbf{s}\ \boldsymbol{P}) \rightarrow \mathbf{term}\ \boldsymbol{P}$
   $\mathbf{app'} : \mathbf{term}\ \boldsymbol{P} \rightarrow \mathbf{term}\ \boldsymbol{P} \rightarrow \mathbf{term}\ \boldsymbol{P}$

7. Equivalence judgment between both encodings (Example 2.2.6)

$$\begin{array}{lcl}
\textbf{equiv} & : & \Pi \boldsymbol{P}\textbf{:nat. exp} \to (\textbf{term } \boldsymbol{P}) \to \textit{type} \\
\\
\textbf{eqVar} & : & \textbf{equiv } \boldsymbol{P}\ \boldsymbol{E}\ (\textbf{var' } \boldsymbol{X}) \\
 & & \to \textbf{equiv (s } \boldsymbol{P})\ \boldsymbol{E}\ (\textbf{var' (succ } \boldsymbol{X})) \\
\\
\textbf{eqLam} & : & (\Pi \boldsymbol{x}\textbf{:exp. equiv (s } \boldsymbol{P})\ \boldsymbol{x}\ (\textbf{var' one}) \\
 & & \qquad \to \textbf{equiv (s } \boldsymbol{P})\ (\boldsymbol{E}\ \boldsymbol{x})\ \boldsymbol{T}) \\
 & & \to \textbf{equiv } \boldsymbol{P}\ \ (\textbf{lam } (\lambda \boldsymbol{x}\textbf{. } \boldsymbol{E}\ \boldsymbol{x}))\ (\textbf{lam' } \boldsymbol{T}) \\
\\
\textbf{eqApp} & : & \textbf{equiv } \boldsymbol{P}\ \boldsymbol{E_1}\ \boldsymbol{T_1} \\
 & & \to \textbf{equiv } \boldsymbol{P}\ \boldsymbol{E_2}\ \boldsymbol{T_2} \\
 & & \to \textbf{equiv } \boldsymbol{P}\ (\textbf{app } \boldsymbol{E_1}\ \boldsymbol{E_2})\ (\textbf{app' } \boldsymbol{T_1}\ \boldsymbol{T_2})
\end{array}$$

Chapter 3

SimpleDelphin (Simply Typed)

We have seen in Chapter 1 that the first challenge of designing a calculus of recursive functions supporting higher-order encodings is to cleanly separate the two function spaces for representation and computation. We achieve this by designing Delphin as a two level system. The representation level is LF, but in this chapter we will limit ourselves to the simply-typed variant, which we named SimpleLF in Chapter 2. SimpleLF is essentially Church's simply-typed λ-calculus extended with a signature. Recall that a signature is simply a collection of datatypes and we often refer to the name of the type as a type constant and its constructors as object constants. We will use the simply-typed datatypes summarized in Section 2.4.1.

The second challenge of designing our calculus is in supporting recursion over higher-order encodings. For example, assume we want to write an evaluator for untyped λ-expressions $e ::= x \mid \lambda x.\ e \mid e_1\ e_2$. If we are only concerned with evaluating closed λ-expressions, we can write an eager evaluator whose pseudocode is:

Example 3.0.1 (Closed Eager Evaluator in Pseudocode)**.**

$$
\begin{array}{lll}
\mathsf{eval}\ (\ e_1\ e_2\ ) = & \text{let } v = \mathsf{eval}\ e_2 & \\
& \quad \text{let } (\ \lambda x.\ f\ ) = \mathsf{eval}\ e_1 & \\
& \quad\quad \text{in } \mathsf{eval}\ (\ f[v/x]\ ) & \\
\mathsf{eval}\ (\ \lambda x.\ e\ ) = & (\ \lambda x.\ e\ ) & \\
\mathsf{eval}\ x & = & \textit{impossible by invariant}
\end{array}
$$

The eval function never looks under λ binders. However, another reasonable reduction strategy is to reduce redexes under λ binders. In pseudocode, we write this function as:

Example 3.0.2 (Open Evaluator in Pseudocode)**.**

$$
\begin{array}{lll}
\mathsf{evalBeta}\ (\ e_1\ e_2\ ) = & \text{let } v = \mathsf{evalBeta}\ e_2 & \\
& \quad \text{in case } (\mathsf{evalBeta}\ e_1\ ) & \\
& \quad\quad \text{of } (\ \lambda x.\ f\ ) \mapsto \mathsf{evalBeta}\ (\ f[v/x]\ ) & \\
& \quad\quad\ \mid\ f \mapsto (\ f\ v\ ) & \\
\mathsf{evalBeta}\ (\ \lambda x.\ e\ ) = & \lambda x.\ (\mathsf{evalBeta}\ e\ ) & \\
\mathsf{evalBeta}\ x & = & x
\end{array}
$$

Delphin is designed to allow one to elegantly write such functions while using a higher-order encoding of the untyped λ-calculus. From Example 2.1.2, we represent untyped λ-expressions as objects of type **exp** with two constructors:

$$
\begin{array}{lcl}
\mathbf{lam} & : & \mathbf{(exp \to exp) \to exp} \\
\mathbf{app} & : & \mathbf{exp \to exp \to exp}
\end{array}
$$

The ability to recurse under a λ in evalBeta (Example 3.0.2) is the subject of our next discussion.

3.1 Closed vs. Open Terms

Recall that we write $\ulcorner - \urcorner$ to stand for the representation function mapping terms of our object language into their encoding. By using HOAS, the representation of $e\ (\lambda x.\ g)$ is **app** $\boldsymbol{E}$ **(lam (**$\boldsymbol{\lambda x}$**:exp.** $\boldsymbol{G}$**))** where $\ulcorner\ e\ \urcorner = \boldsymbol{E}$ and $\ulcorner\ g\ \urcorner = \boldsymbol{G}$. However, there is a crucial difference between $\boldsymbol{E}$ and $\boldsymbol{G}$. Namely, the variable $\boldsymbol{x}$ can occur free in $\boldsymbol{G}$ but not in $\boldsymbol{E}$. In other words, $\boldsymbol{G}$ is *open*. We can view these free variables as additional constructors. For example, if $\boldsymbol{E}$ is an **exp** formed by constructors **app** and **lam**, then $\boldsymbol{G}$ is an **exp** constructed by **app**, **lam**, and $\boldsymbol{x}$. Therefore, we can view higher-order encodings as providing a uniform way of dynamically extending a datatype with new constructors for every variable.

When writing programs, we will assume that free variables make sense outside the scope of all local bindings. In other words, the pattern **lam (**$\boldsymbol{\lambda x}$**:exp.** $\boldsymbol{F'}$**)** refers to object-level functions $\lambda x.\ f$ where x does not occur in f. However, we can match against all functions $\lambda x.\ f$ using the pattern **lam (**$\boldsymbol{\lambda x}$**:exp.** $\boldsymbol{F}\ \boldsymbol{x}$**)**. Notice that $\ulcorner\ f\ \urcorner = \boldsymbol{F}\ \boldsymbol{x}$, where $\boldsymbol{F}$ is the function (of type **exp** $\longrightarrow$ **exp**) resulting from abstracting away all occurrences of $\boldsymbol{x}$ from $\ulcorner\ f\ \urcorner$. This is an example of higher-order matching which will be discussed in more detail in Section 3.3. Recall that **lam (**$\boldsymbol{\lambda x}$**:exp.** $\boldsymbol{F}\ \boldsymbol{x}$**)** is equivalent to the pattern **lam** $\boldsymbol{F}$ since equality on our representation level is modulo $\alpha\beta\eta$ (Chapter 2) and by our above convention, the variable $\boldsymbol{x}$ cannot occur in $\boldsymbol{F}$. Although they are equivalent, we will prefer to write **lam (**$\boldsymbol{\lambda x}$**:exp.** $\boldsymbol{F}\ \boldsymbol{x}$**)**, which is in canonical form (η-long, β-short), highlighting the fact that $\boldsymbol{F}$ is a function. In returning to our previous example, the pattern we would write to match λ-expressions of the form $e\ (\lambda x.\ g)$ is **app** $\boldsymbol{E}$ **(lam (**$\boldsymbol{\lambda x}$**:exp.** $\boldsymbol{G'}\ \boldsymbol{x}$**))**.

The big challenge is in providing a way to recurse over representation-level functions. For example, using our HOAS encoding, the function evalBeta would be a

function from **exp** to **exp** and the second case would be:

$$\mathsf{evalBeta}\ (\mathbf{lam}\ (\boldsymbol{\lambda x}\mathbf{:exp.}\ \boldsymbol{F}\ \boldsymbol{x})) = \ldots$$

where $\boldsymbol{F}$ **:** $(\mathbf{exp} \longrightarrow \mathbf{exp})$. However, what do we do in this case? We cannot recurse on $\boldsymbol{F}$ because it is a function, i.e. it does not have type **exp**. The only thing we can do with functions is apply them, but we have nothing to which $\boldsymbol{F}$ can be applied.

3.2 Uninstantiable Variables (Parameters)

Mainstream programming language such as C, Java, ML, and Haskell, all offer one interpretation of variables. In all of these language, all variables serve as placeholders, which are all *instantiated* to concrete values before execution. The only free variables that occur in values are constants from the signature (i.e. constructors).

Higher-order encodings provide a uniform way of extending a datatype dynamically by new constructors within a term. Therefore, in order to work with higher-order encodings, Delphin provides an explicit mechanism to dynamically extend datatypes.

Delphin's most novel feature is its new variable binding construct $\nu\boldsymbol{x}.\ e$, where ν is pronounced as *new*. Expressions $\nu\boldsymbol{x}.\ e$ evaluate to $\nu\boldsymbol{x}.\ v$, where v is a value. In other words, evaluation of e occurs while the variable $\boldsymbol{x}$ remains uninstantiated. Therefore, for the scope of e, the variable $\boldsymbol{x}$ can be thought of as a new constant in the signature, which we will henceforth call a parameter. The type of $\nu\boldsymbol{x}.\ e$ is $\nabla\boldsymbol{x}.\ \tau$, where ∇ is pronounced as *nabla* (this will be formalized in Section 3.4.1). An expression of type $\nabla\boldsymbol{x}.\ \tau$ is an expression of type τ where the $\boldsymbol{x}$ may occur free, i.e.

it is a τ which makes sense in the signature extended with an $\boldsymbol{x}$ of an appropriate type.

Definition 3.2.1 (Parameter)**.** *A parameter of type* $\boldsymbol{A}$ *is a dynamically introduced constant of type* $\boldsymbol{A}$*; i.e. it is an extension to the signature. Parameters of type* $\boldsymbol{A}$ *are distinguished on the type level by writing* $\boldsymbol{A}^{\#}$*.*

The Delphin calculus distinguishes between *parameters* (extensions of the signature) and *objects* (built from constants and parameters). The type $\boldsymbol{A}^{\#}$ refers to a parameter of type $\boldsymbol{A}$. In the expression $\nu\boldsymbol{x}.\ e$, the variable $\boldsymbol{x}$ has type $\boldsymbol{A}^{\#}$. Additionally, $\boldsymbol{x}$ also has type $\boldsymbol{A}$ and as such $\boldsymbol{A}^{\#}$ can be seen as a subtype of $\boldsymbol{A}$. Although all parameters are objects, the converse does not necessarily hold. The full form of ν and ∇ include a type annotation and are written as $\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e$ and $\nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$, respectively.

Since one may dynamically create arbitrary parameters, a Delphin function expects arguments that are constructed from constants in the signature as well as parameters. As an example, consider the evaluation of $\nu\boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \mathsf{evalBeta}\ \boldsymbol{x}$. As evaluation occurs under the binding for $\boldsymbol{x}$, this expression would get stuck if evalBeta did not provide a case for parameters. If this situation can occur, the function evalBeta must not only define cases for **lam** and **app**, but it must also define a case for parameters $\boldsymbol{x}$. We will see that this situation can indeed occur as we will recurse on representational-level functions by applying them to parameters. Therefore, object-level variables x will correspond to parameters.

As we may introduce parameters, Delphin functions may be called within the scope of arbitrary extensions to the signature. It is important to emphasize that the type system guarantees that the inputs and outputs of functions all make sense with respect to the same extensions to the signature. Therefore, *a function's type does*

not express what parameters may exist when the function is called. The important property is just that we guarantee that exactly the same parameters that exist in the inputs also exist in the outputs. Therefore, if no parameters have been created and we call a function, the result will also be guaranteed to have no parameters. For example, a function foo of type $\tau \supset \nabla \boldsymbol{x}.\ \sigma$ takes as input an expression of type τ in any extension to the signature and returns an expression of type $\nabla \boldsymbol{x}.\ \sigma$ that makes sense with respect to the same extension. In other words, it takes an expression of type τ that makes sense with respect to some arbitrary collection of parameters, and returns an expression of type σ that makes sense with respect to the same collection of parameters augmented by $\boldsymbol{x}$.

Similar to ML and Haskell, Delphin returns a *Match Non-Exhaustive Warning* if it cannot conclude that a list of cases covers all possibilities. This reasoning requires the programmer to declare which parameters their functions are intended to handle, and Delphin then verifies that functions are defined for all possible constants and parameters that may occur. We refer to this meta-level check as *coverage* and defer a detailed discussion of coverage checking to Chapter 5.

3.3 SimpleDelphin Calculus

The Delphin calculus distinguishes between two levels: computational and representational. Its most prominent feature is its newness type constructor ∇, which binds uninstantiable parameters introduced by our ν construct. Figure 3.1 summarizes all syntactic categories of the Delphin calculus. We will explain and illustrate SimpleDelphin with examples building on the encodings of natural numbers **nat** (Example 2.1.1) and λ-expressions **exp** (Example 2.1.2). Afterwards, we will present the formal static and dynamic semantics in Sections 3.4 and 3.5. Section 3.6 will discuss

Types	δ	::=	$\tau \mid \boldsymbol{A} \mid \boldsymbol{A}^{\#}$
Computational Types	τ, σ	::=	$\text{unit} \mid \delta \supset \tau \mid \delta \star \tau \mid \nabla \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ \tau$
Variables	α	::=	$\boldsymbol{x} \mid u$
Expressions	e, f	::=	$\alpha \mid \boldsymbol{M} \mid () \mid e\ f \mid (e,\ f) \mid \nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ e \mid e \backslash \boldsymbol{x}$ $\mid \mu u {\in} \tau.\ e \mid \text{fn}\ (c_1 \mid \ldots \mid c_n)$
Cases	c	::=	$\epsilon \alpha {\in} \delta.\ c \mid \nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ c \mid c \backslash x \mid e \mapsto f$

Figure 3.1: Syntactic Definitions of SimpleDelphin

some meta-theoretic results. We will then finish this chapter with more advanced examples. Section 3.7 demonstrates how to translate between the untyped λ-calculus and untyped combinators and Section 3.8 translates between a de Bruijn and a HOAS encoding of λ-expressions. We summarize and conclude this chapter in Section 3.9.

We write δ to refer to any type as we distinguish between *representational types* $\boldsymbol{A}$, *parameter types* $\boldsymbol{A}^{\#}$, and *computational types* τ. Representational (LF) types $\boldsymbol{A}$ are the SimpleLF types defined in Section 2.1, and are just type constants $\boldsymbol{a}$ and arrow types $\boldsymbol{A_1} \rightarrow \boldsymbol{A_2}$. We write $\boldsymbol{A}^{\#}$ to denote parameters of type $\boldsymbol{A}$. By distinguishing parameters we can write functions that range over parameters as well as write patterns that only match parameters, both of which are crucial features. It is important to reiterate that all parameters of type $\boldsymbol{A}$ are objects of type $\boldsymbol{A}$, but the converse does not necessarily hold.

We also distinguish representation-level and computation-level variables by $\boldsymbol{x}$ and u, respectively. Computational types are constructed from four type constructors: unit, the function type constructor $\supset$, the product type constructor $\star$, and the *newness* type constructor ∇.

Computational types τ disallow computing anything of LF type $\boldsymbol{A}$. For example, functions inhabit the type $\delta \supset \tau$ and τ cannot be an $\boldsymbol{A}$. This syntactically guarantees that the only expressions of representation type $\boldsymbol{A}$ are explicit representational objects $\boldsymbol{M}$. This separation is necessary for our eventual extension to dependent types.

Since LF types may depend on arbitrary objects of type $\boldsymbol{A}$, this distinction enforces that dependencies are only on LF objects $\boldsymbol{M}$ rather than any arbitrary expression e. If we did not make this distinction, then type checking in the dependently-typed version would be undecidable as it may need to perform arbitrary computation just to determine if two types are equal. This is, for example, a viable option exploited in Cayenne (Augustsson 1998), but we prefer to keep type checking decidable and distinguish it from evaluation.

The Delphin type for pairs is $\delta \star \tau$ and its values are of the form $(e,\ f)$, where both e and f are values. The first argument ranges over any type δ and this distinction will be important in the dependently-typed version as we only want to permit the type of the second to depend on the first when the first is an $\boldsymbol{A}$ or $\boldsymbol{A}^{\#}$. Pairs are eliminated via case analysis.

Although computation cannot result in an object of type $\boldsymbol{A}$, it may result in an object of type $\boldsymbol{A} \star \text{unit}$. We abbreviate this type as $\langle \boldsymbol{A} \rangle$, whose values are $\langle \boldsymbol{M} \rangle$, which corresponds to $(\boldsymbol{M}, ())$. Analogously, $\langle \boldsymbol{A}^{\#} \rangle$ stands for $\boldsymbol{A}^{\#} \star \text{unit}$, whose values are $\langle \boldsymbol{x} \rangle$. We summarize our syntactic sugar in Figure 3.2, which will all be explained throughout this section. The interested reader may notice that $\langle \boldsymbol{A} \rangle$ may be viewed as a *monad* distinguishing computation on objects of type $\boldsymbol{A}$. In this chapter all of our examples will range over $\langle \boldsymbol{A} \rangle$ as we do not need to distinguish values from computation. However, in Chapter 4 we will see examples where we need to write functions that range only over values, i.e. functions over $\boldsymbol{A}$.

Since functions and pairs range over δ, they each provide three respective function and pairing constructs– over $\boldsymbol{A}$, $\boldsymbol{A}^{\#}$, and τ. For example, a function whose domain is over **exp** will work on all objects of type **exp** while a function whose domain is over $\mathbf{exp}^{\#}$ can only be applied to parameters. We will often call this second flavor of function a *parameter function*. Many of our examples will take parameter functions

$$
\begin{array}{rcl@{\qquad}rcl}
\langle \boldsymbol{A} \rangle & \equiv & \boldsymbol{A} \star \mathrm{unit} & \mathrm{case}\ e\ \mathrm{of}\ (c_1 \mid \ldots \mid c_n) & \equiv & (\mathrm{fn}\ (c_1 \mid \ldots \mid c_n))\ e \\
\langle \boldsymbol{A}^{\#} \rangle & \equiv & \boldsymbol{A}^{\#} \star \mathrm{unit} & \langle \boldsymbol{x}^{\#} \rangle \mapsto e & \equiv & \epsilon \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ \langle \boldsymbol{x} \rangle \mapsto e \\
\langle \boldsymbol{M} \rangle & \equiv & (\boldsymbol{M}, ()) & \mathrm{let}\ e_{\mathrm{pat}} = e_{\mathrm{arg}}\ \mathrm{in}\ e_{\mathrm{body}} & \equiv & (\mathrm{fn}\ e_{\mathrm{pat}} \mapsto e_{\mathrm{body}})\ e_{\mathrm{arg}} \\
 & & & W\ \mathrm{with}\ e_1 \mapsto e_2 & \equiv & \mathrm{Defined\ in\ Section\ 3.7.2}
\end{array}
$$

Figure 3.2: Abbreviations

as input.

Definition 3.3.1 (Parameter Function)**.** *A parameter function is any function that has at least one input argument of type* $\boldsymbol{A}^{\#}$ *or* $\langle \boldsymbol{A}^{\#} \rangle$*.*

As already stated, functions may range over any type δ. We define values of Delphin functions as a list of cases fn $(c_1 \mid \ldots \mid c_n)$, which means that we do not introduce an explicit computation-level λ-construct. Therefore, a case expression "case e of $(c_1 \mid \ldots \mid c_n)$" is syntactic sugar for "(fn $(c_1 \mid \ldots \mid c_n)$) e" (Figure 3.2). This technique of defining functions by cases avoids the aliasing of bound variables, which helps to simplify the presentation of our calculus when we add dependent types in Chapter 4.

The recursion operator $\mu u {\in} \tau.\ e$ is standard. Note that μ can only recurse on Delphin computational types τ and neither on LF types $\boldsymbol{A}$ nor parameter types $\boldsymbol{A}^{\#}$.

We write a single case as $e \mapsto f$ where e is the *pattern* and f is the *body*. Patterns may contain pattern variables, which are explicitly declared with the ϵ construct. We use ϵ to declare pattern variables of any type representing objects or parameters. For example, fn $\epsilon u {\in} \tau.\ u \mapsto u$ encodes the identity function on type τ. Multiple cases are captured via alternation, $c_1 \mid c_2$, and $\cdot$ stands for an empty list of cases.

Function application is written as $e\ f$. During computation, e is expected to yield a set of cases c, where one that matches the argument is selected and executed. During the matching process, ϵ-bound (pattern) variables are appropriately

instantiated.

Example 3.3.2 (Addition (Fully Explicit))**.**
The function plus *adds two natural numbers.*

$$\begin{aligned}
&\mu\mathsf{plus} \in \langle\mathbf{nat}\rangle \supset \langle\mathbf{nat}\rangle \supset \langle\mathbf{nat}\rangle.\\
&\quad \text{fn } \langle\mathbf{z}\rangle \qquad\qquad\quad \mapsto \text{fn } \epsilon\boldsymbol{M}{\in}\mathbf{nat}.\ \langle\boldsymbol{M}\rangle \mapsto \langle\boldsymbol{M}\rangle\\
&\quad\ \mid \epsilon\boldsymbol{N}{\in}\mathbf{nat}.\ \langle\mathbf{s}\ \boldsymbol{N}\rangle \mapsto \text{fn } \epsilon\boldsymbol{M}{\in}\mathbf{nat}.\ \langle\boldsymbol{M}\rangle \mapsto (\text{case } (\mathsf{plus}\ \langle\boldsymbol{N}\rangle\ \langle\boldsymbol{M}\rangle)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{of } \epsilon\boldsymbol{x}{\in}\mathbf{nat}.\ \langle\boldsymbol{x}\rangle \mapsto \langle\mathbf{s}\ \boldsymbol{x}\rangle)
\end{aligned}$$

Example 3.3.3 (Addition (Free Variables as Pattern Variables))**.**
Here we write plus *but omit the explicit declaration of pattern variables. We will assume that all free variables occurring in patterns are implicitly ϵ-quantified.*

$$\begin{aligned}
&\mu\mathsf{plus} \in \langle\mathbf{nat}\rangle \supset \langle\mathbf{nat}\rangle \supset \langle\mathbf{nat}\rangle.\\
&\quad \text{fn } \langle\mathbf{z}\rangle \quad \mapsto \text{fn } \langle\boldsymbol{M}\rangle \mapsto \langle\boldsymbol{M}\rangle\\
&\quad\ \mid \langle\mathbf{s}\ \boldsymbol{N}\rangle \mapsto \text{fn } \langle\boldsymbol{M}\rangle \mapsto (\text{case } (\mathsf{plus}\ \langle\boldsymbol{N}\rangle\ \langle\boldsymbol{M}\rangle)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{of } \langle\boldsymbol{x}\rangle \mapsto \langle\mathbf{s}\ \boldsymbol{x}\rangle)
\end{aligned}$$

Example 3.3.2 shows how to add two numbers and all pattern variables are explicitly quantified. However, we will typically omit the explicit declaration of pattern (ϵ-bound) variables and interpret all free variables occurring in patterns as ϵ-quantified at the beginning of the case branch (i.e. as far left in the branch as possible). Example 3.3.3 shows this simplified version of plus. The omission of pattern variables may introduce an ambiguity in the type of the function. As an example, consider the identity function fn $\epsilon u{\in}\tau.\ u \mapsto u$. If we omit the declaration of u, then we would write fn $u \mapsto u$, which is undesirable as the type of the function is unclear. Therefore, when necessary, we will use more verbose abbreviations with type ascriptions, e.g. fn $(u{\in}\tau) \mapsto u$.

In order to simplify our code, we will utilize the "let" statements given in Figure 3.2. As an example, the expression "let $\langle\boldsymbol{x}\rangle = e$ in f" is syntactic sugar for "(fn $\langle\boldsymbol{x}\rangle \mapsto f$) e".

The plus function is straightforward. The interesting observation is that we are performing case analysis on the result of the recursive call to extract the data $\boldsymbol{x}$ from the computation-level $\langle \boldsymbol{x} \rangle$. One should observe that plus ranges over computations of **nat** as expressed by the type $\langle \mathbf{nat} \rangle$. Therefore, we can call plus on a computation, such as plus (plus $\langle \mathbf{z} \rangle$ $\langle \mathbf{s}\ \mathbf{z} \rangle$) $\langle \mathbf{s}\ (\mathbf{s}\ \mathbf{z}) \rangle$ which evaluates to $\langle \mathbf{s}\ (\mathbf{s}\ (\mathbf{s}\ \mathbf{z})) \rangle$. However, if the type of plus was $\mathbf{nat} \supset \mathbf{nat} \supset \langle \mathbf{nat} \rangle$, then we could only apply plus to values and the same computation would be achieved by writing let $\langle \boldsymbol{x} \rangle =$ plus $\langle \mathbf{z} \rangle$ $\langle \mathbf{s}\ \mathbf{z} \rangle$ in plus $\boldsymbol{x}$ $(\mathbf{s}\ (\mathbf{s}\ \mathbf{z}))$, which evaluates to the same result.

Before continuing, we first turn to an example which actually uses a higher-order encoding in writing an evaluator for untyped λ-expressions.

Example 3.3.4 (Closed Eager Evaluator)**.** *The function* eval *implements an eager operational semantics on the untyped λ-calculus based on the pseudocode from Example 3.0.1.*

$$
\begin{array}{l}
\mu\mathsf{eval} \in \langle \mathbf{exp} \rangle \supset \langle \mathbf{exp} \rangle. \\
\quad \mathrm{fn}\ \ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle \quad \mapsto \mathrm{case}\ (\mathsf{eval}\ \langle \boldsymbol{E_1} \rangle,\ \mathsf{eval}\ \langle \boldsymbol{E_2} \rangle) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{of}\ (\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{F}\ \boldsymbol{x}) \rangle,\ \langle \boldsymbol{V} \rangle) \mapsto \mathsf{eval}\ \langle \boldsymbol{F}\ \boldsymbol{V} \rangle \\
\quad \mid\ \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle \ \mapsto \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle
\end{array}
$$

The interesting case is the application, $\langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle$, and we will compare the code here to the pseudocode from Example 3.0.1. When we recurse on $\langle \boldsymbol{E_2} \rangle$ we expect any v, which is represented as $\langle \boldsymbol{V} \rangle$. The recursion on $\langle \boldsymbol{E_1} \rangle$ expects a function $\lambda x.\ f$, which is represented as $\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{F}\ \boldsymbol{x}) \rangle$. We now continue evaluation on $f[v/x]$, which simply corresponds to $\langle \boldsymbol{F}\ \boldsymbol{V} \rangle$. This illustrates how one uses LF-level (representation-level) application to perform object-language substitutions. One should note how similar this function looks to the pseudocode we wrote earlier, i.e. using HOAS allows us to focus on the real intended computation alleviating the need to deal explicitly with variables and renamings which are tedious and error-

prone. However, the constructs we have addressed so far do not give us the ability to recurse under LF functions, which is necessary to implement the open evaluator whose pseudocode is given in Example 3.0.2. We now turn to the dynamic creation of parameters.

The *newness type constructor* ∇ binds variables that will always remain *uninstantiated* and hence computation will not be delayed. Again, we refer to this second class of variables as parameters. Delphin's newness type constructor is written as $\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau$ and the corresponding values are $\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ v$, where v is a value. The expression $\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e$ will always evaluate to an expression $\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ v$. In other words, evaluation in an extended signature results in values in the same extended signature. Just as ν dynamically extends the signature, the ∇ type is eliminated via $e \backslash \boldsymbol{x}$, which dynamically shrinks the signature to its form before $\boldsymbol{x}$ was introduced.

Example 3.3.5 (Open Evaluator)**.** *We now reduce redexes under λ-binders following the pseudocode from Example 3.0.2.*

$$
\begin{array}{lll}
\mu \mathsf{evalBeta} \in \langle \mathbf{exp} \rangle \supset \langle \mathbf{exp} \rangle. & & \\
\quad \text{fn} \ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle & \mapsto \text{case } (\mathsf{evalBeta}\ \langle \boldsymbol{E_1} \rangle,\ \mathsf{evalBeta}\ \langle \boldsymbol{E_2} \rangle) & \\
& \quad \text{of } (\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{F}\ \boldsymbol{x}) \rangle,\ \langle \boldsymbol{V} \rangle) & \mapsto \mathsf{evalBeta}\ \langle \boldsymbol{F}\ \boldsymbol{V} \rangle \\
& \quad \mid (\langle \boldsymbol{F} \rangle,\ \langle \boldsymbol{V} \rangle) & \mapsto \langle \mathbf{app}\ \boldsymbol{F}\ \boldsymbol{V} \rangle \\
\quad \mid \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle & \mapsto \text{case } (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \mathsf{evalBeta}\ \langle \boldsymbol{E}\ \boldsymbol{x} \rangle) & \\
& \quad \text{of } \underline{\epsilon \boldsymbol{E'} \in \mathbf{exp} \to \mathbf{exp}.} & \\
& \qquad \underline{(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{E'}\ \boldsymbol{x} \rangle)} & \mapsto \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E'}\ \boldsymbol{x}) \rangle \\
\quad \mid \langle \boldsymbol{x}^{\#} \rangle & \mapsto \langle \boldsymbol{x} \rangle &
\end{array}
$$

The function $\mathsf{evalBeta}$ (Example 3.3.5) has type $\langle \mathbf{exp} \rangle \supset \langle \mathbf{exp} \rangle$, just as our previous eval function (Example 3.3.4). The type only says that the function takes an $\langle \mathbf{exp} \rangle$ in any extension of the signature and returns an $\langle \mathbf{exp} \rangle$ that makes sense with respect to the same extension. Unlike our eval function, we will indeed (recursively) call $\mathsf{evalBeta}$ in scope of extensions to the signature due to our handling of the

$\langle$**lam (**$\boldsymbol{\lambda x.\ E\ x}$**)**$\rangle$ case.

The $\langle$**lam (**$\boldsymbol{\lambda x.\ E\ x}$**)**$\rangle$ case illustrates how we can recurse under functions. Since $\boldsymbol{E}$ is of functional type, we create a new parameter $\boldsymbol{x}$ and continue computation on $\langle \boldsymbol{E}\ \boldsymbol{x} \rangle$. Therefore, we are substituting computation-level parameters for object-level variables. The expression $\nu \boldsymbol{x} \in \mathbf{exp}^{\#}$. evalBeta $\langle \mathbf{E}\ \boldsymbol{x} \rangle$ has type $\nabla \boldsymbol{x} \in \mathbf{exp}^{\#}$. $\langle \mathbf{exp} \rangle$ and triggers evaluation of evalBeta in the signature extended by $\boldsymbol{x}$. Although the introduction of parameters is easy, eliminating them is more difficult. We do this via case analysis by matching the result against $\langle \boldsymbol{E'}\ \boldsymbol{x} \rangle$. Now $\boldsymbol{E'}$ will be an LF function corresponding to the abstraction of $\boldsymbol{x}$ from the result of the recursive call, an example of higher-order matching. It is important to note that the variable $\boldsymbol{x}$ *cannot* occur free in $\boldsymbol{E'}$ because $\boldsymbol{E'}$ was declared outside of the scope of $\boldsymbol{x}$. This lack of dependency is reflected by the lexical scoping in the Delphin code above: the pattern variable $\epsilon \boldsymbol{E'}$ is declared to the left of $\nu \boldsymbol{x}$. We included and underlined the declaration of the pattern variable $\boldsymbol{E'}$ to emphasize it here, but it can be omitted as pattern variables are easily identifiable as the free variables occurring in patterns and are defaultly quantified as far left as possible (i.e. outside the scope of any ν). The desired result of this case indeed corresponds to an object-level function which is thus represented as $\langle$**lam (**$\boldsymbol{\lambda x.\ E'\ x}$**)**$\rangle$.

Since the **lam** case substitutes parameters for object-level variables, we may encounter such parameters as expressed in the last case. We adopt the syntax $\boldsymbol{x}^{\#}$ in patterns, which means that the pattern variable $\boldsymbol{x}$ is a parameter. In other words, the last case $\langle \boldsymbol{x}^{\#} \rangle \mapsto \langle \boldsymbol{x} \rangle$ is shorthand (Figure 3.2) for $\epsilon \boldsymbol{x} \in \mathbf{exp}^{\#}$. $\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{x} \rangle$ and as such will only match against parameters. This case specifies that parameters (object-level variables) evaluate to themselves.

Finally, the **app** case is not much different from the eval example except that it also handles the possibility that the first argument to the application does not

evaluate to a function. We now give a sample execution showing that the identity function evaluates to itself:

Example 3.3.6 (Sample evalBeta Execution)**.**

evalBeta $\langle \mathbf{lam}\ (\boldsymbol{\lambda y}\mathbf{:exp.}\ \boldsymbol{y})\rangle$
$\ldots \rightarrow$ case $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.$ evalBeta $\langle (\boldsymbol{\lambda y}\mathbf{:exp.}\ \boldsymbol{y})\ \boldsymbol{x}\rangle)$
of $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{E'}\ \boldsymbol{x}\rangle)\ \mapsto \langle \mathbf{lam}\ (\boldsymbol{\lambda x}\mathbf{.}\ \boldsymbol{E'}\ \boldsymbol{x})\rangle$ *by* **lam** *case*
$\ldots \rightarrow$ case $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{x}\rangle)$
of $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{E'}\ \boldsymbol{x}\rangle)\ \mapsto \langle \mathbf{lam}\ (\boldsymbol{\lambda x}\mathbf{.}\ \boldsymbol{E'}\ \boldsymbol{x})\rangle$ *by parameter case as* $\langle (\boldsymbol{\lambda y}\mathbf{:exp.}\ \boldsymbol{y})\ \boldsymbol{x}\rangle \equiv_{\boldsymbol{\beta}} \langle \boldsymbol{x}\rangle$
$\ldots \rightarrow \langle \mathbf{lam}\ (\boldsymbol{\lambda x}\mathbf{:exp.}\ \boldsymbol{x})\rangle$
since matching sets $\boldsymbol{E'}$ *to* $\boldsymbol{\lambda x}\mathbf{:exp.}\ \boldsymbol{x}$

As we have seen, one may perform case analysis over a ∇ type. It is important to observe that LF functions can be seen as equivalent to values of the ∇ type, where the latter is just providing a means of performing intermediate computation. For example, we can use case analysis to convert between the value $\langle \boldsymbol{\lambda x}\mathbf{.}\ \boldsymbol{M}\ \boldsymbol{x}\rangle$ and $\nu \boldsymbol{x}.\ \langle \boldsymbol{M}\ \boldsymbol{x}\rangle$, which will be the subject of our next discussion.

A Delphin function that would convert the former into the latter would have type $\langle \boldsymbol{A} \rightarrow \boldsymbol{B}\rangle \supset \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \langle \boldsymbol{B}\rangle$ and be written as fn $\langle \boldsymbol{\lambda x}\mathbf{:}\boldsymbol{A}\mathbf{.}\ \boldsymbol{M}\ \boldsymbol{x}\rangle \mapsto \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \langle \boldsymbol{M}\ \boldsymbol{x}\rangle$. This function takes an LF function and lifts it to Delphin's ∇ type by creating a fresh parameter $\boldsymbol{x}$ to which the input function can be applied.

Conversely, a function of type $(\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \langle \boldsymbol{B}\rangle) \supset \langle \boldsymbol{A} \rightarrow \boldsymbol{B}\rangle$ can be written as fn $(\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \langle \boldsymbol{M}\ \boldsymbol{x}\rangle) \mapsto \langle \boldsymbol{\lambda x}\mathbf{:}\boldsymbol{A}\mathbf{.}\ \boldsymbol{M}\ \boldsymbol{x}\rangle$. This function converts a computation-level ∇ into a representation-level function by matching against values of the first.

Our past two functions both utilize higher-order matching and it is important to emphasize that higher-order matching of this form is both decidable and straightforward. Languages like ML and Haskell all provide mechanisms for doing matching. For example, we can match the pattern $(E,\ F)$ against a value v by noting that v must be of the form $(v_1,\ v_2)$ and then the solution is to assign E to v_1 and to assign

F to v_2. Similarly, a pattern of the form $\langle \boldsymbol{M}\ \boldsymbol{x}\rangle$ can be matched against a value v by noting that v must be of the form $\langle \boldsymbol{N}\rangle$ (where $\boldsymbol{x}$ can occur free) and the solution is to set $\boldsymbol{M}$ to $\boldsymbol{\lambda x.\ N}$. There are two important things to notice. First, the parameter $\boldsymbol{x}$ cannot escape its scope as $\boldsymbol{M}$ can only be assigned to an object where the $\boldsymbol{x}$ does not occur free. Second, higher-order matching of this form is straightforward.

We will heavily use this form of higher-order matching. Just as we introduced the ∇ type to reason over higher-order encodings, we can employ higher-order matching to get rid of it again. However, one can also eliminate a ∇ type using the elimination form $e\backslash\boldsymbol{x}$.

The expression $e\backslash\boldsymbol{x}$ can be thought of as a restricted form of application. For example, if f is a function of type $\delta \supset \tau$, then we can get an expression of type τ by applying f to any argument of type δ. Similarly, if f has type $\nabla \boldsymbol{y}{\in}\boldsymbol{A}^{\#}.\ \tau$, then when we are in an extended signature containing $\boldsymbol{x}$, we can get an expression of type τ by replacing the hypothetical $\boldsymbol{y}$ with the concrete $\boldsymbol{x}$, expressed as $f\backslash\boldsymbol{x}$. The reason it is a *restricted* form of application is that the $\boldsymbol{y}$ represented a new parameter with respect to τ, so we can only replace $\boldsymbol{y}$ with an $\boldsymbol{x}$ which is also a new parameter with respect to τ. This is statically guaranteed by typing (and evaluating) f in the signature shrunk not to include $\boldsymbol{x}$. We will describe and illustrate $e\backslash\boldsymbol{x}$ more thoroughly in the advanced examples.

We also remark that we have $\nu\boldsymbol{x}.\ c$ and $c\backslash\boldsymbol{x}$ over cases, which have a similar meaning to their counterparts over expressions. By allowing these constructs to range over cases, we add further flexibility in what we can express with patterns. For example, this is useful in implementing *exchange* properties as well as the properties that will be proved in Lemma 3.4.2. We conclude this section with a final example counting variables.

Example 3.3.7 (Variable Counting)**.** *Here we write a function that counts the number of variable occurrences in λ-expressions. For example, the number of variables in* $\ulcorner\, \lambda x.\ x\ (\lambda y.\ x\ y)\, \urcorner$ *is* 3.

$$
\begin{array}{l}
\mu\mathsf{cntvar} \in \langle \mathbf{exp} \rangle \supset \langle \mathbf{nat} \rangle. \\
\quad \mathrm{fn}\ \ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle \quad\ \ \mapsto \mathsf{plus}\ (\mathsf{cntvar}\ \langle \boldsymbol{E_1} \rangle)\ (\mathsf{cntvar}\ \langle \boldsymbol{E_2} \rangle) \\
\quad\ \ |\ \ \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle \mapsto \mathrm{case}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \mathsf{cntvar}\ \langle \mathbf{E}\ \boldsymbol{x} \rangle) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathrm{of}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{N} \rangle) \mapsto \langle \boldsymbol{N} \rangle \\
\quad\ \ |\ \ \langle \boldsymbol{x}^{\#} \rangle \qquad\qquad\ \ \mapsto \langle \mathbf{s}\ \mathbf{z} \rangle
\end{array}
$$

We explain the $\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle$ case. Since $\boldsymbol{E}$ is of functional type, we create a parameter $\boldsymbol{x}$ and recurse on $\langle \boldsymbol{E}\ \boldsymbol{x} \rangle$. From the very definition of natural numbers in Example 2.1.1, we deduce that it is impossible for the $\boldsymbol{x}$ to occur in the result and hence we match the result against $\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{N} \rangle$ Note that if it was possible for $\boldsymbol{x}$ to occur in the result, then this case would only match during runtime in situations where the $\boldsymbol{x}$ did not occur free in the result.

Therefore, if the programmer leaves out essential cases, then it is possible to get stuck, corresponding to a *Match Non-Exhaustive* error. For example, if it was possible for the $\boldsymbol{x}$ to occur in the result, then Delphin would complain saying that no case is handled for $\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{N'}\ \boldsymbol{x} \rangle$. However, λ-expressions cannot occur in natural numbers, so this function is complete.

Example 3.3.8 (Sample cntvar Execution)**.**

$$
\begin{array}{ll}
\mathsf{cntvar}\ \langle \mathbf{lam}\ (\boldsymbol{\lambda y}\mathbf{:exp.}\ \mathbf{app}\ \boldsymbol{y}\ \boldsymbol{y}) \rangle & \\
\ldots \to \mathrm{case}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \mathsf{cntvar}\ \langle \mathbf{app}\ \boldsymbol{x}\ \boldsymbol{x} \rangle) & \\
\qquad \mathrm{of}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{N} \rangle)\ \mapsto \langle \boldsymbol{N} \rangle & \textit{by } \mathbf{lam} \textit{ case} \\
\ldots \to \mathrm{case}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \mathsf{plus}\ (\mathsf{cntvar}\ \langle \boldsymbol{x} \rangle)\ (\mathsf{cntvar}\ \langle \boldsymbol{x} \rangle)) & \\
\qquad \mathrm{of}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{N} \rangle)\ \mapsto \langle \boldsymbol{N} \rangle & \textit{by } \mathbf{app} \textit{ case} \\
\ldots \to \mathrm{case}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \mathsf{plus}\ \langle \mathbf{s}\ \mathbf{z} \rangle\ \langle \mathbf{s}\ \mathbf{z} \rangle) & \\
\qquad \mathrm{of}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{N} \rangle)\ \mapsto \langle \boldsymbol{N} \rangle & \textit{by two uses of parameter case} \\
\ldots \to \mathrm{case}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \mathbf{s}\ (\mathbf{s}\ \mathbf{z}) \rangle) & \\
\qquad \mathrm{of}\ (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{N} \rangle)\ \mapsto \langle \boldsymbol{N} \rangle & \textit{by } \mathsf{plus} \\
\ldots \to \langle \mathbf{s}\ (\mathbf{s}\ \mathbf{z}) \rangle & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \textit{since matching sets } \boldsymbol{N} \textit{ to } \mathbf{s}\ (\mathbf{s}\ \mathbf{z})
\end{array}
$$

3.4 Static Semantics

Before presenting the typing rules, the role of contexts deserves special attention.

$$\text{Context } \Omega ::= \cdot \mid \Omega, \alpha{\in}\delta \mid \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$$

A Delphin context, Ω, serves two purposes. Besides assigning types to variables, it also distinguishes variables intended for instantiation from uninstantiable parameters. Instantiable declarations appear in the context with the form $\alpha{\in}\delta$, and we will often refer to these as pattern-variable declarations.[1] Alternatively, declarations of the form $\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ represent extensions to the signature; i.e. uninstantiable parameters introduced by ν. The distinction between $\boldsymbol{x}{\in}\boldsymbol{A}^{\#}$ and $\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ is highlighted by comparing fn $(\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \mapsto e$ and $\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e$. The first binds a parameter that is intended for instantiation while the latter will remain uninstantiated. We assume all declarations in Ω to be uniquely named, and we achieve this goal by tacitly renaming variables. During the actual execution of Delphin programs, Ω only contains declarations of the latter form, which indicates the extension to the signature in which evaluation takes place. In comparison, evaluation in ML and Haskell always occur under no extensions to the signature.

3.4.1 Type System

We write $\Omega \vdash e \in \delta$ for the central derivability judgment, which we present in Figure 3.3.

The rule isLF is the only rule for type $\boldsymbol{A}$ and stipulates that in order for an expression $\boldsymbol{M}$ to be an LF object, we must be able to type it using the LF typing

[1] However, note that both pattern variables and variables for recursive functions appear in this form as they are both instantiated before execution.

judgment under $\|\Omega\|$ (Definition 3.4.1).

Definition 3.4.1 (Casting)**.** *In order to employ LF typing, we define* $\|\Omega\|$ *as casting of a context* Ω*, which throws out all declarations* $u{\in}\tau$ *and converts* $\boldsymbol{x}{\in}\boldsymbol{A}$*,* $\boldsymbol{x}{\in}\boldsymbol{A}^{\#}$*, and* $\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ *all into* $\boldsymbol{x}{:}\boldsymbol{A}$*, yielding an LF context* $\boldsymbol{\Gamma}$*.*

$$\|\Omega\| = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ \|\Omega'\|, \boldsymbol{x}{:}\boldsymbol{A} & \text{if } \Omega = \Omega', \boldsymbol{x}{\in}\delta \text{ and } \delta = \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#} \\ \|\Omega'\|, \boldsymbol{x}{:}\boldsymbol{A} & \text{if } \Omega = \Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \\ \|\Omega'\| & \text{if } \Omega = \Omega', u{\in}\tau \end{cases}$$

The variable rules τvar and $\mathsf{var}^{\#}$ allow one to use assumptions in the context of types τ and $\boldsymbol{A}^{\#}$, respectively. The only expression of type $\boldsymbol{A}^{\#}$ is a variable $\boldsymbol{x}$. One should note that if $\boldsymbol{x}$ has type $\boldsymbol{A}^{\#}$, then it also has type $\boldsymbol{A}$ by isLF. We could have introduced a different syntax to eliminate this ambiguity, but instead we exploit it because all parameters are indeed objects, i.e. $\boldsymbol{A}^{\#}$ can be thought of as a subtype of $\boldsymbol{A}$.

The rest of the rules deal with computational types τ. Function types are introduced via cases c. The introduction rule impl expresses that all branches must have the same type. Note that we allow for an empty list of cases which will occur in our advanced examples utilizing parameter functions. Functions are eliminated through application with impE.

Cases contain pattern variables, which are simply added to the context in cEps. The actual function type is introduced in cMatch illustrating that functions are defined via case analysis. In the branch $e \mapsto f$, e is the pattern and f is the body. Finally, we also have a ν and $c\backslash\boldsymbol{x}$ construct over cases, via cNew and cPop. These have similar semantics to their counterparts on expressions, discussed next.

The introduction form of ∇ is called new. A new parameter $\boldsymbol{x}$ is created by adding $\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ to the context. The expression $\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e$ evaluates e where the parameter

$$\frac{\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\,\mathsf{isLF} \qquad \frac{}{\Omega, u{\in}\tau, \Omega_2 \vdash u \in \tau}\,\tau\mathsf{var} \qquad \frac{((\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\,\mathsf{var}^{\#}$$

$$\frac{i \geq 0, \text{For all } i, \quad \Omega \vdash c_i \in \tau}{\Omega \vdash \text{fn } (c_1 \mid \ldots \mid c_n) \in \tau}\,\mathsf{impI} \qquad \frac{\Omega \vdash e \in \delta \supset \tau \quad \Omega \vdash f \in \delta}{\Omega \vdash e\ f \in \tau}\,\mathsf{impE}$$

$$\frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\,\mathsf{new} \qquad \frac{\Omega \vdash e \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau}\,\mathsf{pop}$$

$$\frac{\Omega \vdash e \in \delta \quad \Omega \vdash f \in \tau}{\Omega \vdash (e,\ f) \in \delta \star \tau}\,\mathsf{pairI} \qquad \frac{\Omega, u{\in}\tau \vdash e \in \tau}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\,\mathsf{fix} \qquad \frac{}{\Omega \vdash () \in \text{unit}}\,\mathsf{top}$$

. . . . . .

$$\frac{\Omega, \alpha{\in}\delta \vdash c \in \tau}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\,\mathsf{cEps} \qquad \frac{\Omega \vdash e \in \delta \quad \Omega \vdash f \in \tau}{\Omega \vdash e \mapsto f \in \delta \supset \tau}\,\mathsf{cMatch}$$

$$\frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\,\mathsf{cNew} \qquad \frac{\Omega \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \in \tau}\,\mathsf{cPop}$$

Figure 3.3: SimpleDelphin Typing Rules

$\boldsymbol{x}$ can occur free. Previously, our examples have shown how to use case analysis to eliminate a ∇ type. However, the elimination rule pop eliminates a ∇ type via an application-like construction, $e\backslash\boldsymbol{x}$, which shifts computation of e to occur without the uninstantiable parameter $\boldsymbol{x}$. If $\Omega \vdash e \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau$, then $\boldsymbol{x'}$ is a fresh parameter with respect to the context Ω. Therefore, in an extended context $\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2$, we can substitute $\boldsymbol{x}$ for $\boldsymbol{x'}$ and yield an expression of type τ. Notice that we have built-in weakening by Ω_2 into τvar and pop. This ensures that weakening holds in our system.

The construct $e\backslash\boldsymbol{x}$ may be the most unusual construct for one to get a feel for when using it in a bottom-up manner. One way of viewing this construct is that if one wants to construct an expression of type τ in the context $\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$, they may

instead shift computation to Ω and construct an expression of type $\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau$. This allows the programmer to do intermediate computation outside the scope of $\boldsymbol{x}$. For example, the expression "(let $R = \mathsf{code}$ in $\nu \boldsymbol{x}.\ R) \backslash \boldsymbol{x}$" allows us to execute code inside Ω, while the entire expression makes sense in $\Omega, \boldsymbol{x} \overset{\nabla}{\in} \boldsymbol{A}^{\#}$.

Lemma 3.4.2. *The following types are inhabited:*

1. $\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (\tau \supset \sigma) \supset (\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau \supset \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \sigma)$
2. $(\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau \supset \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \sigma) \supset \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (\tau \supset \sigma)$
3. $\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (\tau \star \sigma) \supset (\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau \star \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \sigma)$
4. $(\nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau \star \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \sigma) \supset \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (\tau \star \sigma)$

Proof. We give the Delphin functions that inhabit the types above. We explicitly declare some pattern variables using ϵ to emphasize their scope, but they can be inferred.

1. fn $u_1 \mapsto$ fn $u_2 \mapsto \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (u_1 \backslash \boldsymbol{x})\ (u_2 \backslash \boldsymbol{x})$
2. fn $u_1 \mapsto$ fn $\underline{\epsilon u_2.}\ \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ ((u_2 \backslash \boldsymbol{x}) \mapsto (u_1\ u_2) \backslash \boldsymbol{x})$
3. fn $\underline{\epsilon u_1.\ \epsilon u_2.}\ (\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (u_1 \backslash \boldsymbol{x},\ u_2 \backslash \boldsymbol{x})) \mapsto (u_1,\ u_2)$
4. fn $(u_1,\ u_2) \mapsto \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (u_1 \backslash \boldsymbol{x},\ u_2 \backslash \boldsymbol{x})$

□

Lemma 3.4.2 is meant to illustrate that $\nabla \boldsymbol{x}.\ \tau$ means that we have an expression of type τ in a signature extended by $\boldsymbol{x}$. Therefore, it is not surprising that $\nabla \boldsymbol{x}.\ (\tau \supset \sigma)$ is equivalent to $(\nabla \boldsymbol{x}.\ \tau) \supset (\nabla \boldsymbol{x}.\ \sigma)$ since both functions take a τ in an extended signature and return a σ in the same extended signature. This lemma also illustrates where it is useful to have ν over cases as well as expressions. Part 2 needs to use ν over cases. For example, in Part 2 we could not write fn $u_1 \mapsto \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.$ fn $((u_2 \backslash \boldsymbol{x}) \mapsto (u_1\ u_2) \backslash \boldsymbol{x})$ as the expression $(u_2 \backslash \boldsymbol{x})$ would not be typeable because the pattern variable u_2 would be declared in scope of the parameter $\boldsymbol{x}$.

Finally, pairs are introduced via pairI and eliminated using case analysis. The typing rules fix and top are standard.

$$\frac{}{\cdot \vdash \cdot : \cdot} \qquad \frac{\Omega' \vdash \omega : \Omega \quad \Omega' \vdash e \in \delta}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}$$

$$\frac{\Omega' \vdash \omega : \Omega}{(\Omega', \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})}\ * \qquad \frac{\Omega' \vdash \omega : \Omega}{\Omega', \alpha{\in}\delta \vdash \uparrow_{\alpha} \omega : \Omega}$$

. . . . . .

$$\frac{}{\cdot \overset{\text{lf}}{\vdash} \cdot : \cdot} \qquad \frac{\boldsymbol{\Gamma'} \overset{\text{lf}}{\vdash} \boldsymbol{\gamma} : \boldsymbol{\Gamma} \quad \boldsymbol{\Gamma'} \overset{\text{lf}}{\vdash} \boldsymbol{M} \in \boldsymbol{A}}{\boldsymbol{\Gamma'} \overset{\text{lf}}{\vdash} (\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) : (\boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A})} \qquad \frac{\boldsymbol{\Gamma'} \overset{\text{lf}}{\vdash} \boldsymbol{\gamma} : \boldsymbol{\Gamma}}{\boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A} \overset{\text{lf}}{\vdash} \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}}$$

Figure 3.4: Typing Substitutions (SimpleDelphin and SimpleLF)

3.4.2 Substitutions

We now turn to substitutions. Recall that a context Ω serves two roles. Pattern variables appear in the context with the form $\alpha{\in}\delta$. Extensions to the signature appear in the form $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$. Our definition of substitution permits the instantiation of pattern variables (the former) while preserving the signature (the latter).

Definition 3.4.3 (Substitutions)**.**

$$\begin{array}{lll} \textit{Substitutions:} & \omega & ::= \cdot \mid \omega, e/\alpha \mid \ \uparrow_{\alpha} \omega \\ \textit{LF Substitutions:} & \boldsymbol{\gamma} & ::= \cdot \mid \boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x} \mid \ \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma} \end{array}$$

We present the typing rules for substitutions in Figure 3.4. The judgment $\Omega' \vdash \omega : \Omega$ states that substitution ω brings one from the domain Ω to the codomain Ω'. Additionally, the judgment $\boldsymbol{\Gamma'} \overset{\text{lf}}{\vdash} \boldsymbol{\gamma} : \boldsymbol{\Gamma}$ is a SimpleLF substitution from $\boldsymbol{\Gamma}$ to $\boldsymbol{\Gamma'}$. Note that weakening is not built into the rules. We have an explicit shift construct, "$\uparrow_{\alpha}$", to weaken the substitution, which binds tighter than ",". We tacitly rename variables so that all elements in substitutions are uniquely named just as we did for contexts.

The interesting rule handles extensions to the signature, which is marked with an *. The $\boldsymbol{x}$ is a fresh parameter with respect to Ω and the substitution ω maps

expressions from Ω into Ω'. Therefore, we only allow a renaming of the variable name by mapping it to an $\boldsymbol{x'}$ which is fresh with respect to Ω'. In other words, our definition of substitution guarantees that the signature portion of the domain is equivalent (modulo renaming) to the signature portion of the codomain.

Definition 3.4.4 (Casting Substitution)**.** *We define $\|\omega\|$, which turns a computation-level substitution into an LF one by throwing out non-LF assumptions.*

$$\begin{array}{lcl} \|\cdot\| & = & \cdot \\ \|\omega, \boldsymbol{M}/\boldsymbol{x}\| & = & \|\omega\|, \boldsymbol{M}/\boldsymbol{x} \\ \|\omega, e/u\| & = & \|\omega\| \end{array} \qquad \begin{array}{lcl} \| \uparrow_{\boldsymbol{x}} \omega\| & = & \uparrow_{\boldsymbol{x}} \|\omega\| \\ \| \uparrow_{u} \omega\| & = & \|\omega\| \end{array}$$

The casting operation will translate a computational ω into an LF $\boldsymbol{\gamma}$ by discarding the computation-level aspects of ω. Recall that typing appealed to the LF typing judgment (rule isLF) after casting the context. Similarly, substitution application on an $\boldsymbol{M}$ will utilize substitution application as defined solely on LF objects by first casting the substitution.

Definition 3.4.5 (Identity Substitutions)**.** *We next define identity substitutions as follows:*

$$\begin{array}{lcl} \mathrm{id}_{\cdot} & = & \cdot \\ \mathrm{id}_{\Omega,\alpha\in\delta} & = & \uparrow_{\alpha} \mathrm{id}_{\Omega}, \alpha/\alpha \\ \mathrm{id}_{\Omega,\boldsymbol{x}\in\overset{\triangledown}{\boldsymbol{A}^{\#}}} & = & \uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega}, \boldsymbol{x}/\boldsymbol{x} \end{array} \quad \Bigg| \quad \begin{array}{lcl} \mathbf{id}_{\cdot} & = & \cdot \\ \mathbf{id}_{\boldsymbol{\Gamma},\boldsymbol{x}:\boldsymbol{A}} & = & \uparrow_{\boldsymbol{x}} \mathbf{id}_{\boldsymbol{\Gamma}}, \boldsymbol{x}/\boldsymbol{x} \end{array}$$

Identity substitutions are well-typed and do not have any effect with respect to substitution application. Formally, $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \mathbf{id}_{\boldsymbol{\Gamma}} : \boldsymbol{\Gamma}$ (appendix, Lemma B.1.3) and $\Omega \vdash \mathrm{id}_{\Omega} : \Omega$ (appendix, Lemma B.5.5). Additionally, $\boldsymbol{M}[\mathbf{id}_{\boldsymbol{\Gamma}}] = \boldsymbol{M}$ (appendix, Lemma B.1.4) and $e[\mathrm{id}_{\Omega}] = e$ (appendix, Lemma B.5.3). Substitution application will be discussed in the following section, Section 3.5.

Our meta-theory will require the composition of substitutions. If $\Omega_1 \vdash \omega_1 : \Omega_0$ and $\Omega_2 \vdash \omega_2 : \Omega_1$, then we define $\omega_1 \circ \omega_2$ such that $\Omega_2 \vdash \omega_1 \circ \omega_2 : \Omega_0$. To avoid confusion we should point out that our definition of substitution composition is reversed from

standard function composition as functions are applied to terms, but we speak of terms under substitutions. Function composition $(f \circ g)(x)$ is usually interpreted as $f(g(x))$. Similarly, the composition of substitutions satisfy the property $e[\omega_1 \circ \omega_2] = (e[\omega_1])[\omega_2]$, which will be proved in Chapter 5.

Definition 3.4.6 (Substitution Composition)**.** *We define substitution composition as:*

$$
\begin{array}{lcl}
\omega \circ (\uparrow_\alpha \omega') & = & \uparrow_\alpha (\omega \circ \omega') \\
\cdot \circ \omega', \ \textit{where } \omega' \neq \uparrow_{\alpha''} \omega'' & = & \omega' \\
(\omega, e/\alpha) \circ \omega', \ \textit{where } \omega' \neq \uparrow_{\alpha''} \omega'' & = & (\omega \circ \omega', e[\omega']/\alpha) \\
(\uparrow_\alpha \omega) \circ (\omega', e/\alpha) & = & \omega \circ \omega'
\end{array}
$$

$$
\begin{array}{lcl}
\boldsymbol{\gamma} \circ^{\mathbf{lf}} \ (\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma'}) & = & \uparrow_{\boldsymbol{x}} (\boldsymbol{\gamma} \circ^{\mathbf{lf}} \ \boldsymbol{\gamma'}) \\
\cdot \circ^{\mathbf{lf}} \ \boldsymbol{\gamma'}, \ \textit{where } \boldsymbol{\gamma'} \neq \uparrow_{\boldsymbol{x}''} \boldsymbol{\gamma''} & = & \boldsymbol{\gamma'} \\
(\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathbf{lf}} \ \boldsymbol{\gamma'}, \ \textit{where } \boldsymbol{\gamma'} \neq \uparrow_{\boldsymbol{x}''} \boldsymbol{\gamma''} & = & (\boldsymbol{\gamma} \circ^{\mathbf{lf}} \ \boldsymbol{\gamma'}, \boldsymbol{M}[\boldsymbol{\gamma'}]/\boldsymbol{x}) \\
(\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}) \circ^{\mathbf{lf}} \ (\boldsymbol{\gamma'}, \boldsymbol{M}/\boldsymbol{x}) & = & \boldsymbol{\gamma} \circ^{\mathbf{lf}} \ \boldsymbol{\gamma'}
\end{array}
$$

Our definition of LF substitutions $\boldsymbol{\Gamma'} \vdash^{\mathbf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}$, differs from the original one (Harper and Pfenning 2005) in that weakening is made explicit. One may simply remove all occurrences of $\uparrow_{\boldsymbol{x}}$ and obtain an equivalent substitution typeable in their system. However, an interesting side note is that substitution composition is not well-typed with their definition, but it is with ours (Chapter 5, Lemma 5.5.1).

3.5 Dynamic Semantics

Execution of programs in ML and Haskell all occur in an empty context. However, Delphin programs are executed in a context Ω which is empty with respect to pattern variable definitions $\alpha{\in}\delta$ and consists exclusively of signature extensions $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$. Therefore, execution of Delphin programs occurs in a context Ω, where Ω represents the dynamic extensions to the signature in which execution is taking place.

Definition 3.5.1 (Values)**.** *The set of values are:*

$$\textit{Values:}\quad v \;::=\; () \mid \mathrm{fn}\ (c_1 \mid \ldots \mid c_n) \mid \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ v \mid (v_1,\ v_2) \mid \boldsymbol{M}$$

As usual for a call-by-value language, functions are considered values. A newness expression $\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ v$ is a value only if its body is a value, which is achieved via evaluation under the ν construct. LF objects $\boldsymbol{M}$ are the only values (and expressions) of type $\boldsymbol{A}$, and pairs are considered values only if their components are values. Therefore, expressions of the form $\langle \boldsymbol{M} \rangle$ are the only values of type $\langle \boldsymbol{A} \rangle$ (Figure 3.2).

We present the small-step operational semantics, $\Omega \vdash e \to f$, in Figure 3.5. The first rule illustrates that the evaluation of $\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e$ simply evaluates e under the context extended with $\boldsymbol{x}$. The declaration is marked as $\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ as this represents an extension to the signature. Notice that after it takes a step in the extended signature, it returns back to the original context and glues the ν back in front as the result makes sense in the extended signature. Evaluation under ν drives our ability to reason under LF λ-binders. Additionally, we evaluate $e'\backslash \boldsymbol{x}$ by first evaluating e' down to $\nu \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ e$ and then substitute $\boldsymbol{x}$ for $\boldsymbol{x'}$. Therefore, we see that $e'\backslash \boldsymbol{x}$ behaves as an application. Notice that if the expression $(\nu \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ e)\backslash \boldsymbol{x}$ is well-typed in $\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2$, then e makes sense in $\Omega, \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ and hence the substitution to replace $\boldsymbol{x}$ for $\boldsymbol{x'}$ is $(\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'})$ where $\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash (\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}) : \Omega, \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$. Therefore, the expression $e[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]$ is well-typed in $\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2$ by substitution (Lemma 3.6.2) and weakening (Lemma 3.6.1).

The small-step operational semantics for cases, $\Omega \vdash c \to c'$, is also shown in Figure 3.5. The first rule non-deterministically instantiates the pattern variables. The substitution $(\mathrm{id}_\Omega, v/\alpha)$ takes expressions from $\Omega, \alpha{\in}\delta$ into Ω by replacing α with an arbitrary well-typed expression v. In our implementation the choice of v is delayed

$$\frac{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash e \to f}{\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e \to \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ f} \qquad \frac{\Omega \vdash e \to e'}{\Omega \vdash (e,\ f) \to (e',\ f)} \qquad \frac{\Omega \vdash f \to f'}{\Omega \vdash (e,\ f) \to (e,\ f')}$$

$$\frac{\Omega \vdash e \to e'}{\Omega \vdash e\ f \to e'\ f} \qquad \frac{\Omega \vdash f \to f'}{\Omega \vdash e\ f \to e\ f'} \qquad \frac{\Omega \vdash e \to f}{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash e \backslash \boldsymbol{x} \to f \backslash \boldsymbol{x}}$$

$$\frac{}{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ e) \backslash \boldsymbol{x} \to e[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega}, \boldsymbol{x} / \boldsymbol{x'}]}$$

$$\frac{}{\Omega \vdash (\mathrm{fn}\ (c_1 \mid \ldots \mid c_n)) \backslash \boldsymbol{x} \to \mathrm{fn}\ ((c_1 \backslash \boldsymbol{x}) \mid \ldots \mid (c_n \backslash \boldsymbol{x}))}$$

$$\frac{\Omega \vdash c_i \overset{*}{\to} (v \mapsto e)}{\Omega \vdash (\mathrm{fn}\ (\ldots \mid c_i \mid \ldots))\ v \to e}\ \boldsymbol{*} \qquad \frac{}{\Omega \vdash \mu u \in \tau.\ e \to e[\mathrm{id}_{\Omega}, \mu u \in \tau.\ e / u]}$$

. . . . . .

$$\frac{\Omega \vdash v \in \delta}{\Omega \vdash \epsilon \alpha \in \delta.\ c \to c[\mathrm{id}_{\Omega}, v / \alpha]} \qquad \frac{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash c \to c'}{\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ c \to \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ c'} \qquad \frac{\Omega \vdash c \to c'}{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash c \backslash \boldsymbol{x} \to c' \backslash \boldsymbol{x}}$$

$$\frac{}{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ c) \backslash \boldsymbol{x} \to c[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega}, \boldsymbol{x} / \boldsymbol{x'}]} \qquad \frac{\Omega \vdash e \to e'}{\Omega \vdash (e \mapsto f) \to (e' \mapsto f)}$$

Figure 3.5: SimpleDelphin Small-Step Operational Semantics

by using an eigenvariable that is instantiated by unification during pattern matching, which is discussed in Chapter 6. The next three rules allow us to work with ν over cases, which is the same for the ν over expressions. We provide a rule to reduce the pattern of a case branch, which can be any arbitrary expression.

Delphin is a call-by-value language and the execution of functions occurs in the rule marked with *****. The judgment $\Omega \vdash c \overset{*}{\to} c'$ refers to the transitive closure (0 or more steps) of the $\to$ judgment. Therefore, if we can reduce c_i so that the pattern matches the argument the function is called on, then we can continue with the body of the resulting case branch. We express matching via syntactic equality as we assume that all LF objects are implicitly reduced to canonical form, which is justified as all LF objects possess a unique canonical form (Theorem 2.2.1). Additionally, we implicitly perform α equality, which is actually implemented as syntactic equality since our implementation (Chapter 6) represents LF functions a la de Bruijn.

Expressions

$u[\omega, e/u] = e$

$u[\omega, e/\alpha] = u[\omega]$ where $u \neq \alpha$

$u[\uparrow_\alpha \omega] = u[\omega]$

$\boldsymbol{M}[\omega] = \boldsymbol{M}[\|\omega\|]$

$()[\omega] = ()$

$(e\ f)[\omega] = (e[\omega])\ (f[\omega])$

$(e,\ f)[\omega] = (e[\omega],\ f[\omega])$

$(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)[\omega] = \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e[\uparrow_{\boldsymbol{x}} \omega, \boldsymbol{x}/\boldsymbol{x}]$

$(e\backslash \boldsymbol{x})[\omega] = e[\omega]\backslash \boldsymbol{x}[\omega]$

$(\mu u{\in}\tau.\ e)[\omega] = \mu u{\in}\tau.\ e[\uparrow_u \omega, u/u]$

$(\text{fn}\ (c_1 \mid \ldots \mid c_n))[\omega] = \text{fn}\ (c_1[\omega] \mid \ldots \mid c_n[\omega])$

Cases

$(\epsilon \alpha{\in}\delta.\ c)[\omega] = \epsilon \alpha{\in}\delta.\ c[\uparrow_\alpha \omega, \alpha/\alpha]$

$(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c)[\omega] = \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c[\uparrow_x \omega, x/x]$

$(c\backslash \boldsymbol{x})[\omega] = c[\omega]\backslash \boldsymbol{x}[\omega]$

$(e \mapsto f)[\omega] = (e[\omega] \mapsto f[\omega])$

LF Objects

$\boldsymbol{c}[\boldsymbol{\gamma}] = \boldsymbol{c}$

$\boldsymbol{x}[\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}] = \boldsymbol{M}$

$\boldsymbol{x}[\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x'}] = \boldsymbol{x}[\boldsymbol{\gamma}]$ where $\boldsymbol{x} \neq \boldsymbol{x'}$

$\boldsymbol{x}[\uparrow_{\boldsymbol{x'}} \boldsymbol{\gamma}] = \boldsymbol{x}[\boldsymbol{\gamma}]$

$(\boldsymbol{\lambda x{:}A.\ M})[\boldsymbol{\gamma}] = \boldsymbol{\lambda x{:}A.\ M}[\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}, \boldsymbol{x}/\boldsymbol{x}]$

$(\boldsymbol{M\ N})[\boldsymbol{\gamma}] = (\boldsymbol{M}[\boldsymbol{\gamma}])\ (\boldsymbol{N}[\boldsymbol{\gamma}])$

Figure 3.6: SimpleDelphin Substitution Application

As we allow patterns to be arbitrary expressions, this also means that one may technically perform case analysis over computation-level functions. As just mentioned, our equality is syntactic equality, even on functions. However, in order to prove coverage, we will restrict patterns in Chapter 5. For example, computation-level functions can only be matched against patterns that consist solely of a pattern variable, as is what is allowed in ML and Haskell. We could have additionally restricted our grammar to distinguish between expressions and patterns, but this distinction is not necessary for the core type theory of Delphin.

Finally, we define substitution application in Figure 3.6. Note that the shift construct is only used for the typing and composition of substitutions; it does not play a role in substitution application.

3.6 Meta-Theoretic Results

SimpleDelphin is a proper subset of the dependently-typed variant in Chapter 4. In this dissertation, we will detail the proofs of the more general system, which straightforwardly also apply to SimpleDelphin. However, we take the opportunity

here to summarize the results with respect to simple types.

Lemma 3.6.1 (Weakening)**.**

If $\Omega \vdash e \in \delta$*, then* $\Omega, \Omega_2 \vdash e \in \delta$*.*

Lemma 3.6.2 (Substitution)**.**

If $\Omega \vdash e \in \delta$ *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash e[\omega] \in \delta$*.*

Theorem 3.6.3 (Type Preservation)**.**

If $\Omega \vdash e \in \tau$ *and* $\Omega \vdash e \rightarrow f$*, then* $\Omega \vdash f \in \tau$*.*

Corollary 3.6.4 (Soundness)**.** *Parameters cannot escape their scope. If* $\Omega \vdash e \in \tau$ *and* $\Omega \vdash e \rightarrow e'$*, then all parameters in* e *and* e' *are declared in* Ω*.*

Proof. Follows directly from type preservation noting that typing in a context Ω guarantees that all encountered parameters are in Ω (easily proved by induction on the typing rules). □

Theorem 3.6.5 (Progress)**.**

Under the condition that all cases in e *are exhaustive, if* $\Omega \vdash e \in \tau$ *and* Ω *only contains declarations of the form* $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$*, then* $\Omega \vdash e \rightarrow f$ *or* e *is a value.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash e \in \tau$. In matching (rule *****) we assume that cases are exhaustive. □

Rather than just assuming that all cases are exhaustive, we formally extend the system appropriately in Chapter 5 so we can formally guarantee that programs do not get stuck and hence Delphin is a type safe language. Although the problem of checking an arbitrary list of cases is undecidable, it is always possible to generate an exhaustive list of cases for any type δ, and this is the approach we will take.

3.7 Untyped λ-Calculus vs. Combinators

In this section we will use Delphin to illustrate a key relationship between the untyped λ-calculus and untyped combinators. The untyped λ-calculus lends itself naturally to programming with the usual notion of function abstraction and application. Alternatively, combinators are an axiomatic system with a notion of application but no abstraction. Although combinators do not provide any method of variable bindings, they are just as expressive as the untyped λ-calculus with respect to the computational behavior that can be performed. We will demonstrate this by converting λ-expressions of one into the other using the representations **exp** (Example 2.1.2) and **comb** (Example 2.1.5). There actually exists a well-developed, but necessarily (and intensionally) obfuscated language based solely on using combinators, called Unlambda (Madore 2008). Although this example illustrates how one may do without variable bindings, it also illustrates how necessary bindings are in order to understand the programs we write. Just as function abstraction in the untyped λ-calculus is a tool for cleanly expressing computation, the use of HOAS in Delphin is a tool to cleanly manipulate and reason about bindings.

3.7.1 From Combinators

We start with the easy direction of converting combinators into λ-expressions. In our encoding of combinators, Example 2.1.5, we discussed how the K and S combinator can be thought of as higher-order functions. The K combinator corresponds to the function $\lambda x.\ \lambda y.\ x$ which takes two arguments and returns the first. Similarly, the S combinator corresponds to $\lambda x.\ \lambda y.\ \lambda z.\ (x\ z)\ (y\ z)$. We call this function comb2exp and it is written as:

Example 3.7.1 (Combinators to λ-Expressions)**.**

μcomb2exp $\in \langle \mathbf{comb} \rangle \supset \langle \mathbf{exp} \rangle$.

$$
\begin{array}{lll}
\text{fn } \langle \mathbf{K} \rangle & \mapsto & \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \mathbf{lam}\ (\boldsymbol{\lambda y.}\ \boldsymbol{x})) \rangle \\
\mid \langle \mathbf{S} \rangle & \mapsto & \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \mathbf{lam}\ (\boldsymbol{\lambda y.} \\
 & & \quad \mathbf{lam}\ (\boldsymbol{\lambda z.}\ \mathbf{app}\ (\mathbf{app}\ \boldsymbol{x}\ \boldsymbol{z})\ (\mathbf{app}\ \boldsymbol{y}\ \boldsymbol{z})))) \rangle \\
\mid \langle \mathbf{MP}\ \boldsymbol{C_1}\ \boldsymbol{C_2} \rangle & \mapsto & \text{let } \langle \boldsymbol{E_1} \rangle = \mathsf{comb2exp}\ \langle \boldsymbol{C_1} \rangle \text{ in} \\
 & & \text{let } \langle \boldsymbol{E_2} \rangle = \mathsf{comb2exp}\ \langle \boldsymbol{C_2} \rangle \text{ in} \\
 & & \quad \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle
\end{array}
$$

This function is straightforward. The **K** and **S** combinators are mapped to their equivalent higher-order functions in the untyped λ-calculus. Additionally, application in the combinators (represented with **MP**) corresponds to application in the untyped λ-calculus (represented with **app**).

Although this function is complete, the translations can be quite large. For example, the expression comb2exp $\langle$**MP (MP S K) K**$\rangle$ evaluates to id1, where:

$$
\begin{array}{l}
\mathsf{id1} = \langle\ \mathbf{app} \\
\qquad \mathbf{(app} \\
\qquad\quad \mathbf{(lam}\ (\boldsymbol{\lambda x.}\ \mathbf{lam}\ (\boldsymbol{\lambda y.}\ \mathbf{lam}\ (\boldsymbol{\lambda z.}\ \mathbf{app}\ (\mathbf{app}\ \boldsymbol{x}\ \boldsymbol{z})\ (\mathbf{app}\ \boldsymbol{y}\ \boldsymbol{z})))))\ \mathbf{(lam}\ (\boldsymbol{\lambda x.}\ \mathbf{lam}\ (\boldsymbol{\lambda y.}\ \boldsymbol{x}))))\ \mathbf{(lam}\ (\boldsymbol{\lambda x.}\ \mathbf{lam}\ (\boldsymbol{\lambda y.}\ \boldsymbol{x})))\ \rangle
\end{array}
$$

At first glance it is quite difficult to understand this result. However, recall that we have already defined a function evalBeta (Example 3.3.5) that allows us to reduce λ-expressions. The expression evalBeta id1 evaluates to $\langle$**lam** $(\boldsymbol{\lambda x.}\ \boldsymbol{x})\rangle$ which tells us that the identity combinator (**MP** (**MP S K**) **K**) is mapped to a λ-expression equivalent to the identity function.

3.7.2 To Combinators

The conversion from **exp** to **comb** is more complicated as we need to get rid of abstractions. Our translation will follow a two-step algorithm.

The first step is bracket abstraction, or ba, which internalizes abstraction with respect to combinators. If $\boldsymbol{M}$ has type $\mathbf{(comb \longrightarrow comb)}$, then we can use ba to get a combinator $\boldsymbol{M'}$ (of type **comb**). Subsequently, if we have a combinator $\boldsymbol{C}$, then the combinator $\mathbf{MP}\ \boldsymbol{M'}\ \boldsymbol{C}$ will be equivalent to the object $\boldsymbol{M}\ \boldsymbol{C}$. Formally, ba is written as:

Example 3.7.2 (Bracket Abstraction)**.**

$$
\begin{array}{lll}
\mu\mathsf{ba} \in \langle \mathbf{comb} \longrightarrow \mathbf{comb} \rangle \supset \langle \mathbf{comb} \rangle. & & \\
\quad \text{fn}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{x} \rangle & \mapsto & \langle \mathbf{MP\ (MP\ S\ K)\ K} \rangle \\
\quad\ \mid \langle \boldsymbol{\lambda x.}\ \mathbf{MP}\ (\boldsymbol{C_1}\ \boldsymbol{x})\ (\boldsymbol{C_2}\ \boldsymbol{x}) \rangle & \mapsto & \text{let } \langle \boldsymbol{C_1'} \rangle = \mathsf{ba}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{C_1}\ \boldsymbol{x} \rangle \text{ in} \\
 & & \quad \text{let } \langle \boldsymbol{C_2'} \rangle = \mathsf{ba}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{C_2}\ \boldsymbol{x} \rangle \text{ in} \\
 & & \quad\ \langle \mathbf{MP\ (MP\ S}\ \boldsymbol{C_1'}\mathbf{)}\ \boldsymbol{C_2'} \rangle \\
\quad\ \mid \langle \boldsymbol{\lambda x.}\ \boldsymbol{C} \rangle & \mapsto & \langle \mathbf{MP\ K}\ \boldsymbol{C} \rangle
\end{array}
$$

We will briefly discuss the bracket abstraction algorithm and refer the reader to Pfenning (2001) for a detailed explanation of the algorithm. The first case handles the creation of identity combinators by mapping identity functions to the combinator $\langle \mathbf{MP\ (MP\ S\ K)\ K} \rangle$. The second case handles **MP** by performing bracket abstraction on both parts and using the **S** combinator on the results. The third case handles combinators $\boldsymbol{C}$ that do not use the hypothetical combinator $\boldsymbol{x}$; in this situation, we use the $\boldsymbol{K}$ combinator. This function covers all possibilities and illustrates an important feature of the existence of canonical forms – we can perform case analysis over LF functions. In systems such as ML and Haskell there is no hope of performing a non-trivial case analysis over functions, as shown here.

Next we write the function exp2comb which traverses λ-expressions and uses ba to convert them into combinators. In this function, we will need to introduce new parameters of **exp** and **comb** together. In order to hold onto the relationship between these parameters, we pass around a parameter function (Def. 3.3.1) W of type $\langle \mathbf{exp}^{\#} \rangle \supset \langle \mathbf{comb} \rangle$. For readability, we will employ type aliasing:

$$\text{type convParamFun} = \langle \mathbf{exp}^{\#} \rangle \supset \langle \mathbf{comb} \rangle$$

Example 3.7.3 (λ-Expressions to Combinators)**.**
Note that $W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle}$ is a placeholder for a snippet of code that will eventually be filled in by the expression "W with $\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle$", whose explanation follows.

μexp2comb $\in$ convParamFun $\supset \langle \mathbf{exp} \rangle \supset \langle \mathbf{comb} \rangle$.
  fn $W \mapsto$
    fn $\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle \ \mapsto$
      (case ($\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \nu \boldsymbol{u} \in \mathbf{comb}^{\#}.$ exp2comb $(W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle})\ \langle \boldsymbol{E}\ \boldsymbol{x} \rangle$)
        of ($\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \nu \boldsymbol{u} \in \mathbf{comb}^{\#}.\ \langle \boldsymbol{C}\ \boldsymbol{u} \rangle$) $\mapsto$ ba $\langle \boldsymbol{\lambda u.}\ \boldsymbol{C}\ \boldsymbol{u} \rangle$)

    | $\langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle \ \mapsto$
      let $\langle \boldsymbol{C_1} \rangle =$ exp2comb $W\ \langle \boldsymbol{E_1} \rangle$ in
      let $\langle \boldsymbol{C_2} \rangle =$ exp2comb $W\ \langle \boldsymbol{E_2} \rangle$ in
        $\langle \mathbf{MP}\ \boldsymbol{C_1}\ \boldsymbol{C_2} \rangle$

    | $\langle \boldsymbol{x}^{\#} \rangle \ \mapsto \ W\ \langle \boldsymbol{x} \rangle$

Example 3.7.3 shows us the exp2comb function but leaves us to fill in code for $W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle}$. Before discussing how to fill this in, we will discuss the big picture of the function. The first argument to exp2comb is a parameter function W, and we will need to update this function when we create new parameters. One may think of W as a function we are building up to handle dynamically created parameters. In the **lam** case, we recurse on $\boldsymbol{E}$ by first creating a parameter $\boldsymbol{x}$ to which $\boldsymbol{E}$ can be applied. However, W is not defined with respect to the new $\boldsymbol{x}$. Therefore, before recursing, we create a fresh combinator $\boldsymbol{u}$ and write

$W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle}$ to refer to the function extending W by mapping $\boldsymbol{x}$ to $\boldsymbol{u}$. The expression $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \nu \boldsymbol{u} \in \mathbf{comb}^{\#}.\ \mathsf{exp2comb}\ (W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle})\ \langle \boldsymbol{E}\ \boldsymbol{x} \rangle)$ contains a recursive call in the presence of the new $\boldsymbol{x}$ and $\boldsymbol{u}$. During the recursion, the $\boldsymbol{x}$ will be translated to $\boldsymbol{u}$ by the last case of $\mathsf{exp2comb}$. Because the $\boldsymbol{u}$ may occur in the result, we utilize higher-order matching by matching against $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \nu \boldsymbol{u} \in \mathbf{comb}^{\#}.\ \langle \boldsymbol{C}\ \boldsymbol{u} \rangle)$. The variable $\boldsymbol{C}$ is an LF function resulting from abstracting away all occurrences of $\boldsymbol{u}$ in the result. The call to ba internalizes this abstraction to create an equivalent combinator. The **app** case recurses on both components and glues the results together with **MP**. Finally, the last case handles parameters by applying the parameter function W.

We now turn our attention to $W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle}$, which exists within the scope of the dynamically created $\boldsymbol{x}$ and $\boldsymbol{u}$. We will start with a very natural way to express this function, discuss its limitations, and then build the more expressive (yet more complicated) function that we will use. As our first attempt, consider the following assignment:

$$\begin{array}{lrcl} \text{Simple Assignment:} & W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle} & = & \text{fn}\ \langle \boldsymbol{x} \rangle\ \mapsto \langle \boldsymbol{u} \rangle \\ & & & \quad \mid\ \langle \boldsymbol{y}^{\#} \rangle \mapsto W\ \langle \boldsymbol{y} \rangle \end{array}$$

This function does indeed typecheck but has some limitations. First it is important to emphasize that the only pattern variable is $\boldsymbol{y}$. The $\boldsymbol{x}$ in the first case is not a free variable so this case states if the function is called on the newly created $\boldsymbol{x}$, then we return the newly created $\boldsymbol{u}$. Unfortunately, this function has overlapping cases. In order for this function to have the desired behavior it is necessary to guarantee that the matching of the second case only occurs when the first case does not match. Recall that W was taken as input and hence it may be undefined with respect to the newly created $\boldsymbol{x}$. Therefore, we must argue that $\boldsymbol{y}$ will not be assigned to $\boldsymbol{x}$ so that

$W\ \langle \boldsymbol{y} \rangle$ would not get stuck. Informally, if we stipulate that the operational behavior of cases is to try the cases in order, then we can indeed make the argument that this function is total. Therefore, informally this function does indeed express the function we desire. This current form of the function cannot be verified total by Delphin (in Chapter 5) as our reasoning does not commit to an order in the evaluation of cases. However, the Delphin type system affords the user the power to express this function with non-overlapping cases, which we discuss next:

$$\begin{array}{lrcl} \text{Better Assignment:} & W_{\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle} & = & (\text{fn}\ \ \nu \boldsymbol{x}_{\boldsymbol{\alpha}}.\ \nu \boldsymbol{u}_{\boldsymbol{\alpha}}.\ (\ \langle \boldsymbol{x}_{\boldsymbol{\alpha}} \rangle\ \mapsto\ \langle \boldsymbol{u}_{\boldsymbol{\alpha}} \rangle\ ) \\ & & & \quad \mid\ \nu \boldsymbol{x}_{\boldsymbol{\alpha}}.\ \nu \boldsymbol{u}_{\boldsymbol{\alpha}}.\ (\ \langle \boldsymbol{y}^{\#} \rangle \mapsto\ W\ \langle \boldsymbol{y} \rangle\ ) \\ & & &)\backslash \boldsymbol{x} \backslash \boldsymbol{u} \end{array}$$

Just as the ν construct dynamically extends the signature, the $e\backslash \boldsymbol{x}$ construct indicates that e is typed and evaluated in the signature dynamically shrunk not to include $\boldsymbol{x}$. The use of this construct in our assignment above tells us that instead of writing a function of type convParamFun within the scope of $\boldsymbol{x}$ and $\boldsymbol{u}$, we will alternatively write a function of type $\nabla \boldsymbol{x}_{\boldsymbol{\alpha}} \in \mathbf{exp}^{\#}.\ \nabla \boldsymbol{u}_{\boldsymbol{\alpha}} \in \mathbf{comb}^{\#}.$ convParamFun outside the scope of $\boldsymbol{x}$ and $\boldsymbol{u}$. For illustrative purposes, we have used $\boldsymbol{x}_{\boldsymbol{\alpha}}$ and $\boldsymbol{u}_{\boldsymbol{\alpha}}$, but when this function is evaluated, they will be renamed (via appropriate substitutions) to $\boldsymbol{x}$ and $\boldsymbol{u}$, respectively (Figure 3.5). When defining a function of this latter type we use ν over cases so that the pattern variable $\boldsymbol{y}$ is declared outside of the scope of $\boldsymbol{x}_{\boldsymbol{\alpha}}$ (i.e. it is implicitly ϵ-quantified to the left of $\nu \boldsymbol{x}_{\boldsymbol{\alpha}}$). It is the scope of the pattern variable $\boldsymbol{y}$ that allows us to express this function with non-overlapping cases. Lemma 3.4.2 also showed an example where we used ν over cases in order to control the scope of pattern variables. However, we are not yet done. Although we have been able to statically express that $\boldsymbol{y}$ cannot match $\boldsymbol{x}_{\boldsymbol{\alpha}}$, the application $W\ \langle \boldsymbol{y} \rangle$ still occurs within the scope of $\boldsymbol{x}_{\boldsymbol{\alpha}}$. Again, we can informally argue that W does not need

to be defined for the $\boldsymbol{x_\alpha}$ as it is being called on a $\boldsymbol{y}$ guaranteed not to be $\boldsymbol{x_\alpha}$. This type of non-local reasoning is hard to automate. Instead, we can again use the \ construct to statically guarantee that the application also occurs outside the scope of $\boldsymbol{x_\alpha}$.

Actual Assignment:

$$
\begin{array}{rcl}
W_{\langle \boldsymbol{x}\rangle \mapsto \langle \boldsymbol{u}\rangle} & = & (\text{fn}\ \ \nu \boldsymbol{x_\alpha}.\, \nu \boldsymbol{u_\alpha}.\, (\ \langle \boldsymbol{x_\alpha}\rangle\ \mapsto\ \langle \boldsymbol{u_\alpha}\rangle\ ) \\
& & \quad \mid\ \nu \boldsymbol{x_\alpha}.\, \nu \boldsymbol{u_\alpha}.\, (\ \langle \boldsymbol{y}^{\#}\rangle \mapsto \\
& & \qquad\qquad (\text{let}\ \langle \boldsymbol{C}\rangle = W\ \langle \boldsymbol{y}\rangle \\
& & \qquad\qquad\ \ \text{in}\ \nu \boldsymbol{x_\beta}.\, \nu \boldsymbol{u_\beta}.\, \langle \boldsymbol{C}\rangle)\backslash\ \boldsymbol{x_\alpha}\backslash \boldsymbol{u_\alpha}\ ) \\
& &)\backslash \boldsymbol{x}\backslash \boldsymbol{u}
\end{array}
$$

The assignment above will work regardless of the order in which we evaluate cases and statically enforces that the input function W is never used in an extended signature. Additionally, and most importantly, the above function can be verified total by Delphin's meta-theory (Chapter 5). This assignment illustrates the proper way to extend (build up) parameter functions. Most of our examples (available on the Delphin website) require extending parameter functions in this manner. Therefore, the Delphin implementation (Chapter 6) offers a shorthand where one may express the extension $W_{\langle \boldsymbol{x}\rangle \mapsto \langle \boldsymbol{u}\rangle}$ by writing "W with $\langle \boldsymbol{x}\rangle \mapsto \langle \boldsymbol{u}\rangle$". We alluded to the existence of "with" which we define here. Formally, the desugaring of "W with $e_1 \mapsto e_2$" in the scope of new parameters $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$, is:

$$
\begin{array}{rcl}
W \text{ with } e_1 \mapsto e_2 & \equiv & (\text{fn}\ \ \nu \boldsymbol{x_1}.\, , \ldots , \nu \boldsymbol{x_n}.\, (\ e_1\ \mapsto\ e_2\ ) \\
& & \quad \mid\ \nu \boldsymbol{x_1}.\, , \ldots , \nu \boldsymbol{x_n}.\, (\ \langle \boldsymbol{y}^{\#}\rangle \mapsto \\
& & \qquad\qquad (\text{let}\ \langle \boldsymbol{R}\rangle = W\ \langle \boldsymbol{y}\rangle \\
& & \qquad\qquad\ \ \text{in}\ \nu \boldsymbol{x_1}.\, , \ldots , \nu \boldsymbol{x_n}.\, \langle \boldsymbol{R}\rangle)\ \backslash \boldsymbol{x_1} \ldots \backslash \boldsymbol{x_n}) \\
& &)\backslash \boldsymbol{x_1} \ldots \backslash \boldsymbol{x_n}
\end{array}
$$

The desugaring above works on parameter functions of type $\langle \boldsymbol{A}^{\#}\rangle \supset \langle \boldsymbol{B}\rangle$. We showed a special case where the result type is $\langle \boldsymbol{B}\rangle$, but we also allow the extension

of parameter functions with more complicated types. For example, if the result was a pair $\langle \boldsymbol{B_1} \rangle \star \langle \boldsymbol{B_2} \rangle$, then the pattern $\langle \boldsymbol{R} \rangle$ would be replaced with $(\langle \boldsymbol{R_1} \rangle, \langle \boldsymbol{R_2} \rangle)$. In other words, the pattern $\langle \boldsymbol{R} \rangle$ is replaced with an appropriate pattern based on the type of the function.[2]

Our definition and explanation of the exp2comb function is complete and we next present a sample execution converting the identity function into a combinator:

Example 3.7.4 (Sample exp2comb Execution)**.**

exp2comb (fn ·) $\langle$**lam (**$\boldsymbol{\lambda x}$**:exp.** $\boldsymbol{x}$**)**$\rangle$
$\ldots \rightarrow$ case $(\nu \boldsymbol{x}.\ \nu \boldsymbol{u}.$ exp2comb $(W$ with $\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle)\ \langle \boldsymbol{x} \rangle)$
of $(\nu \boldsymbol{x}.\ \nu \boldsymbol{u}.\ \langle \boldsymbol{C}\ \boldsymbol{u} \rangle) \mapsto$ ba $\langle \boldsymbol{\lambda u}$**:comb.** $\boldsymbol{C}\ \boldsymbol{u} \rangle$
$\ldots \rightarrow$ case $(\nu \boldsymbol{x}.\ \nu \boldsymbol{u}.\ (W$ with $\langle \boldsymbol{x} \rangle \mapsto \langle \boldsymbol{u} \rangle)\ \langle \boldsymbol{x} \rangle)$
of $(\nu \boldsymbol{x}.\ \nu \boldsymbol{u}.\ \langle \boldsymbol{C}\ \boldsymbol{u} \rangle) \mapsto$ ba $\langle \boldsymbol{\lambda u}$**:comb.** $\boldsymbol{C}\ \boldsymbol{u} \rangle$
$\ldots \rightarrow$ case $(\nu \boldsymbol{x}.\ \nu \boldsymbol{u}.\ \langle \boldsymbol{u} \rangle)$
of $(\nu \boldsymbol{x}.\ \nu \boldsymbol{u}.\ \langle \boldsymbol{C}\ \boldsymbol{u} \rangle) \mapsto$ ba $\langle \boldsymbol{\lambda u}$**:comb.** $\boldsymbol{C}\ \boldsymbol{u} \rangle$
$\ldots \rightarrow$ ba $\langle \boldsymbol{\lambda u}$**:comb.** $\boldsymbol{u} \rangle$
$\ldots \rightarrow \langle$**MP (MP S K) K**$\rangle$

Not only is the "with" syntax heavily used in our online example suite, but the class of examples that fall within this manner of programming is quite large. For example, Twelf (Pfenning and Schürmann 1998) programs can be converted quite elegantly and concisely into Delphin using parameter functions which are only extended in this manner (Chapter 6). Our next example will require a more complicated update of a parameter function.

3.8 HOAS vs. de Bruijn

In this section we will use Delphin to convert between representations of λ-expressions. The type **exp** (Example 2.1.2) refers to the encoding utilizing HOAS while the types

[2] In this Chapter we could have replaced $\langle \boldsymbol{R} \rangle$ with a computation-level pattern-variable u, but we will see in Section 5.2.6 that it is important that we use more descriptive patterns in order to guarantee progress.

variable and **term** are used to encode λ-expressions in a first-order manner a la de Bruijn.

A de Bruijn encoding represents a variable as a number pointing to its binding. The complication of this notation is that the same variable can be represented differently depending on where it occurs. In Example 2.1.4, our representation function passed around a function that mapped object-level variables to de Bruijn indices. The complication was that this function needed to be updated whenever we encounter a new binding so that the newest binding was mapped to `1` and all other mappings were incremented. Similarly, our translation algorithm will need to pass around an analogous function.

3.8.1 Converting HOAS to de Bruijn

Example 3.8.1 (HOAS to de Bruijn)**.**

$$
\begin{array}{l}
\mu\mathsf{toDebruijn} \in (\langle \mathbf{exp}^{\#} \rangle \supset \langle \mathbf{variable} \rangle) \supset \langle \mathbf{exp} \rangle \supset \langle \mathbf{term} \rangle. \\
\quad \text{fn } W \mapsto \\
\quad\quad \text{fn } \langle \mathbf{lam\ (\lambda} \boldsymbol{x}\mathbf{.\ } \boldsymbol{E}\ \boldsymbol{x}\mathbf{)} \rangle \ \mapsto \\
\quad\quad\quad\quad \text{let } W' = (\text{fn } \langle \boldsymbol{x}^{\#} \rangle \mapsto \text{let } \langle \boldsymbol{Y} \rangle = W\ \langle \boldsymbol{x} \rangle \text{ in } \langle \mathbf{succ}\ \boldsymbol{Y} \rangle) \\
\quad\quad\quad\quad\quad \text{in } (\text{case } (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \mathsf{toDebruijn}\ (W' \text{ with } \langle \boldsymbol{x} \rangle \mapsto \langle \mathbf{one} \rangle)\ \langle \boldsymbol{E}\ \boldsymbol{x} \rangle) \\
\quad\quad\quad\quad\quad\quad \text{of } (\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ \langle \boldsymbol{T} \rangle) \ \mapsto \ \langle \mathbf{lam'}\ \boldsymbol{T} \rangle) \\
\\
\quad\quad | \ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle \ \mapsto \\
\quad\quad\quad\quad \text{let } \langle \boldsymbol{T_1} \rangle = \mathsf{toDebruijn}\ W\ \langle \boldsymbol{E_1} \rangle \text{ in} \\
\quad\quad\quad\quad \text{let } \langle \boldsymbol{T_2} \rangle = \mathsf{toDebruijn}\ W\ \langle \boldsymbol{E_2} \rangle \text{ in} \\
\quad\quad\quad\quad\quad \langle \mathbf{app'}\ \boldsymbol{T_1}\ \boldsymbol{T_2} \rangle \\
\\
\quad\quad | \ \langle \boldsymbol{x}^{\#} \rangle \ \mapsto \\
\quad\quad\quad\quad \text{let } \langle \boldsymbol{Y} \rangle = W\ \langle \boldsymbol{x} \rangle \text{ in} \\
\quad\quad\quad\quad\quad \langle \mathbf{var'}\ \boldsymbol{Y} \rangle
\end{array}
$$

Example 3.8.1 gives all the code for our function `toDebruijn` which converts λ-expressions encoded using **exp** into a first-order representation encoded using **term**. This function proceeds quite similarly to our `exp2comb` function from Example 3.7.3.

The first argument is a parameter function mapping parameters to variables which will be updated when we create new parameters. The interesting case is in our handling of $\langle \mathbf{lam\ (\lambda}\boldsymbol{x}\mathbf{.}\ \boldsymbol{E}\ \boldsymbol{x}\mathbf{)} \rangle$. We create a new $\boldsymbol{x}$ so that we can recurse on $\langle \boldsymbol{E}\ \boldsymbol{x} \rangle$ with an updated parameter function where $\boldsymbol{x}$ is mapped to **one** and all other mappings are incremented. Our update of the parameter function is very similar to the one discussed in exp2comb. However, we first create a parameter function W' which simply increments the mappings for all parameters in which W is defined. Then we use the "with" syntax to extend W' to map the new $\boldsymbol{x}$ to **one**.

3.8.2 Converting de Bruijn to HOAS

Example 3.8.2 (de Bruijn to HOAS)**.**

```
μtoHOAS ∈ (⟨variable⟩ ⊃ ⟨exp⟩) ⊃ ⟨exp⟩ ⊃ ⟨term⟩.
    fn W ↦
        fn ⟨lam' T⟩ ↦
                (case (νx∈exp#.
                            toHOAS
                                ((fn νx. ( ⟨one⟩ ↦ ⟨x⟩ )
                                    | νx. ( ⟨succ X⟩ ↦ (let ⟨E⟩ = W ⟨X⟩
                                                        in νx. ⟨E⟩) \x ))
                                  \x)
                                ⟨T⟩
                        )
                    of (νx∈exp#. ⟨E x⟩) ↦ ⟨lam (λx. E x)⟩)

            | ⟨app' T1 T2⟩ ↦
                    let ⟨E1⟩ = toHOAS W ⟨T1⟩ in
                    let ⟨E2⟩ = toHOAS W ⟨T2⟩ in
                        ⟨app E1 E2⟩

            | ⟨var' X⟩ ↦ W ⟨X⟩
```

Example 3.8.2 gives all the code for our function toHOAS which converts λ-expressions encoded using **term** into a higher-order representation encoded using **exp**. Similar to our other functions, the first argument, W, is itself a function to

handle object-level variables. Typically this first argument has been a parameter function as object-level variables corresponded to parameters. As this is a first-order encoding, object-level variables correspond to objects of type **variable** and hence the first argument to this function is a function to handle variables. Alternatively, the first argument could have been a list of object-level variables and the index would correspond to a lookup in the list. However, our approach here is more concise and the list technique is problematic when we turn to our dependently-typed version. The most interesting case is $\langle$**lam' $\boldsymbol{T}$**$\rangle$. In this situation we do not need to apply $\boldsymbol{T}$ to anything, but we still need to create a new parameter $\boldsymbol{x}$ as in the body of $\boldsymbol{T}$, the variable **one** is meant to correspond to a fresh object-level variable. Additionally, variable **succ N** now corresponds to W's mapping for **N**. Although W is not a parameter function and cannot be updated using our "with" syntax, we follow the same methodology and hence it resembles the desugared version of our two previous functions.

3.8.3 Example Execution

We conclude this section with a sample of converting $\lambda x.\ \lambda y.\ x\ y$ between its representation utilizing HOAS and its representation a la de Bruijn.

Example 3.8.3 (Sample Executions of toDebruijn and toHOAS)**.**
We just show the final results of some executions. The step-by-step details are straightforward based on our previous sample executions.

toDebruijn (fn $\cdot$) $\langle$**lam ($\boldsymbol{\lambda x}$. lam ($\boldsymbol{\lambda y}$. app $\boldsymbol{x}$ $\boldsymbol{y}$))**$\rangle$
$\ldots \rightarrow \langle$**lam' (lam' (app' (var' (succ one)) (var' one)))**$\rangle$

toHOAS (fn $\cdot$) $\langle$**lam' (lam' (app' (var' (succ one)) (var' one)))**$\rangle$
$\ldots \rightarrow \langle$**lam ($\boldsymbol{\lambda x}$. lam ($\boldsymbol{\lambda y}$. app $\boldsymbol{x}$ $\boldsymbol{y}$))**$\rangle$

3.9 Conclusion

In this chapter we have discussed and demonstrated the core type theory of the Delphin programming language. The novelty of this work is in providing a way to perform computation under LF λ-binders, where the notions of context and substitutions remain implicit in computations as well as representations.

In the next chapter, Chapter 4, we will extend this simply-typed version to dependent types as we may then take advantage of the full power of LF and work with judgments. Afterwards, in Chapter 5, we will discuss meta-theoretical results including the issue of coverage, which is necessary in order to reason that our functions are total and hence realize that Delphin is also well-suited in writing proofs. In particular, we will use Delphin to show that our functions in the next chapter are also proofs. All examples in this chapter can be experimented with in our implementation (Chapter 6), and the concrete code corresponding to these examples is available in Section 6.10.1.

Chapter 4

Delphin (Dependently Typed)

Chapter 3 presented the key underlying type theory of Delphin where we restricted ourselves to the simply-typed version of LF. In this chapter we add support for full LF. As we have seen in Chapter 2, LF allows one to elegantly represent data utilizing dependent types and higher-order abstract syntax (HOAS). Section 2.4.2 summarizes the datatypes, or gives the signature, that we will use throughout this chapter. Section 4.1 discusses the evolution from SimpleDelphin (Chapter 3) to Delphin. The new static and dynamic semantics are discussed in Sections 4.2 and 4.3, respectively. We summarize some meta-theoretic results in Section 4.4. There are two main examples we will show. In Section 4.5 we will translate between derivations in our natural deduction calculus and our Hilbert-style calculus (i.e. types combinators). Then in Section 4.6 we will extend our translators between the HOAS and de Bruijn representation of λ-expressions (from Section 3.8) to return a proof that the translation is correct. Finally, we conclude this chapter in Section 4.7.

Types	δ	::=	$\tau \mid \boldsymbol{A} \mid \boldsymbol{A}^{\#}$
Computational Types	τ, σ	::=	$\text{unit} \mid \forall\overline{\alpha{\in}\delta}.\ \tau \mid \exists\alpha{\in}\delta.\ \tau \mid \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$
Variables	α	::=	$\boldsymbol{x} \mid u$
Expressions	e, f	::=	$\alpha \mid \boldsymbol{M} \mid () \mid e\ \overline{f} \mid (e,\ f) \mid \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \mid e\backslash\boldsymbol{x}$ $\mid \mu u{\in}\tau.\ e \mid \text{fn}\ \overline{c}$
Cases	c	::=	$\epsilon\alpha{\in}\delta.\ c \mid \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \mid c\backslash\boldsymbol{x} \mid \overline{e} \mapsto f$
Context	Ω	::=	$\cdot \mid \Omega, \alpha{\in}\delta \mid \Omega, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}$

Figure 4.1: Syntactic Definitions of Delphin

4.1 Delphin Calculus

Figure 4.1 summarizes all syntactic categories of the Delphin calculus. We will focus on the changes undergone to evolve SimpleDelphin (Figure 3.1) into its dependently-typed variant.

Our evolution requires two main changes. First, in Section 4.1.1, we generalize the types of functions and pairs into $\forall\alpha{\in}\delta.\ \tau$ and $\exists\alpha{\in}\delta.\ \tau$, respectively. Second, in Section 4.1.2, we generalize functions to range over lists with the type $\forall\overline{\alpha{\in}\delta}.\ \tau$. We use the notation $\overline{\mathcal{X}}$ to refer to a list of $\mathcal{X}$ where the empty list is universally referred to as *nil*. We use ; to deconstruct lists, i.e. $Y; \overline{\mathcal{X}}$ and $\overline{\mathcal{X}}; Y$ represent lists starting and ending with Y. We will often just write Y to refer to the singleton list. A list attached to a context, $\Omega, \overline{\alpha{\in}\delta}$, refers to the context extended with all the declarations in the list. Additionally, a list attached to a substitution, $\omega, \overline{f}/\overline{\alpha}$, expresses an extension to the substitution. For example, $\Omega, (\alpha_1{\in}\delta_1; \alpha_2{\in}\delta_2; \mathit{nil})$ stands for $\Omega, \alpha_1{\in}\delta_1, \alpha_2{\in}\delta_2$ and the substitution $\omega, ((f_1; f_2; \mathit{nil})/(\alpha_1; \alpha_2; \mathit{nil}))$ stands for $\omega, f_1/\alpha_1, f_2/\alpha_2$. Since we introduced a list notation, functions are now formally expressed as "fn $\overline{c}$," but we will write our examples using the former syntax "fn $(c_1 \mid \ldots \mid c_n)$" where "fn $\cdot$" now corresponds to "fn *nil*".

In Section 4.1.3 we simplify the explicit exposition of Delphin programs by allow-

ing for an implicit quantification analogous to our implicit Π-quantification in LF constants. Similar to LF, the implicit quantification is just a convenience afforded to the user in order to eliminate the explicit specification of redundant or easily inferable information; it does not affect the underlying theory.

In order to illustrate and motivate this section we will manipulate derivations of the natural deduction calculus from Example 2.2.3. In this example, formulas A are represented as objects of type **o** and derivations are represented as objects of type **nd** $\ulcorner A \urcorner$. The signature is:

$$
\begin{array}{lcl}
\mathbf{o} & : & \textit{type} \\
\Rightarrow & : & \mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o} \quad \textbf{(right-associative infix)} \\
 & & \\
\mathbf{nd} & : & \mathbf{o} \rightarrow \textit{type} \\
\mathbf{impi} & : & (\mathbf{nd}\ \boldsymbol{A} \rightarrow \mathbf{nd}\ \boldsymbol{B}) \rightarrow \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \\
\mathbf{impe} & : & \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \rightarrow \mathbf{nd}\ \boldsymbol{A} \rightarrow \mathbf{nd}\ \boldsymbol{B}
\end{array}
$$

4.1.1 Generalization of τ (and Values vs. Computation)

In Chapter 2 we saw that SimpleLF evolved into LF by generalizing $\boldsymbol{A} \rightarrow \boldsymbol{B}$ into $\boldsymbol{\Pi x{:}A.\ B}$. Analogously, the first step of turning SimpleDelphin into Delphin is generalizing the function type $\delta \supset \tau$ into $\forall \alpha{\in}\delta.\ \tau$. Similarly, we generalize pairs $\delta \star \tau$ into $\exists \alpha{\in}\delta.\ \tau$. Similar to LF, we will still use the simply-typed syntax when the variable α does not occur free in τ. For example, we may still write $\langle \mathbf{exp} \rangle \supset \langle \mathbf{exp} \rangle$, which is now shorthand for $\forall u{\in}\langle \mathbf{exp} \rangle.\ \langle \mathbf{exp} \rangle$.

Although the $\forall$ type ranges over any type δ, we only permit dependencies on LF objects. The only true dependencies will occur in the form $\forall \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau$ or $\forall \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$. Therefore, the type $\forall u{\in}\tau.\ \sigma$ can always be abbreviated as $\tau \supset \sigma$ as the u will never occur in σ.

As an example, we can express the identity function id on derivations of our natural deduction calculus as a function of type $\forall \boldsymbol{x}{\in}\mathbf{o}.\ \langle \mathbf{nd}\ \boldsymbol{x} \rangle \supset \langle \mathbf{nd}\ \boldsymbol{x} \rangle$, which is

shorthand for $\forall \boldsymbol{x} \in \mathbf{o}.\ \forall u \in \langle \mathbf{nd}\ \boldsymbol{x} \rangle.\ \langle \mathbf{nd}\ \boldsymbol{x} \rangle$. However, there are important observations to be made concerning this function.

The first important observation about the above id function is that its type illustrates our distinction between $\boldsymbol{A}$ and $\langle \boldsymbol{A} \rangle$. As the type of the first argument is $\boldsymbol{o}$, it must be provided with a concrete $\boldsymbol{M}$. This illustrates why we distinguish values $\boldsymbol{M}$ from arbitrary computations e. The type of the expression "id $(\mathbf{p} \Rightarrow \mathbf{p})$" is "$\langle \mathbf{nd}\ (\mathbf{p} \Rightarrow \mathbf{p}) \rangle \supset \langle \mathbf{nd}\ (\mathbf{p} \Rightarrow \mathbf{p}) \rangle$". If we had not made this distinction, then one could have written the expression "id e" where e could be any arbitrary expression. In this scenario, determining if two types are equal corresponds to determining if arbitrary expressions are equal. Languages such as Cayenne (Augustsson 1998) perform this test by having the type checker evaluate the expressions, which blurs the separation between type checking and computation and hence also leads to an undecidable type checker. By separating $\boldsymbol{A}$ from $\langle \boldsymbol{A} \rangle$, we do not pollute LF dependencies with computation and hence type checking does not perform any computation and is decidable. As dependencies are only on LF objects $\boldsymbol{M}$, determining if two types are equal may correspond to checking if two $\boldsymbol{M}$'s are equal, which is decidable by the existence of unique canonical forms (Theorem 2.2.1).

4.1.2 Generalization to Lists

There are many ways to write the above id function. An easy solution is to write fn $(\boldsymbol{A} \in \mathbf{o}) \mapsto$ fn $\langle \boldsymbol{D} \in (\mathbf{nd}\ \boldsymbol{A}) \rangle \mapsto \langle \boldsymbol{D} \rangle$. However, for illustrative purposes we will consider writing an identity function that recursively traverses the derivation $\langle \boldsymbol{D} \rangle$. With the tools we have provided so far, this function can be written as:

Example 4.1.1 (First Attempt at Identity Function ($\mathsf{id1}$))**.**

$$
\begin{array}{l}
\mu\mathsf{id1} \in \forall \boldsymbol{A}{\in}\mathbf{o}.\ \langle \mathbf{nd}\ \boldsymbol{A}\rangle \supset \langle \mathbf{nd}\ \boldsymbol{A}\rangle. \\
\quad \text{fn}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \ \mapsto\ (\text{fn}\ \langle \mathbf{impi}\ (\boldsymbol{\lambda d}{:}(\mathbf{nd}\ \boldsymbol{A}).\ \boldsymbol{D}\ \boldsymbol{d})\rangle \ \mapsto \\
\qquad\qquad\qquad (\text{case}\ (\nu \boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A})^{\#}.\ \mathsf{id1}\ \mathbf{B}\ \langle \boldsymbol{D}\ \boldsymbol{d}\rangle) \\
\qquad\qquad\qquad\quad \text{of}\ (\nu \boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A})^{\#}.\ \langle \boldsymbol{D'}\ \boldsymbol{d}\rangle)\ \mapsto\ \langle \mathbf{impi}\ (\boldsymbol{\lambda d}.\ \boldsymbol{D'}\ \boldsymbol{d})\rangle)) \\
\quad \mid \boldsymbol{B} \qquad\quad \mapsto\ (\text{fn}\ \langle \mathbf{impe}\ \boldsymbol{D_1}\ \boldsymbol{D_2}\rangle \ \mapsto \\
\qquad\qquad\qquad \text{let}\ \langle \boldsymbol{D'_1}\rangle = \mathsf{id1}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})\ \langle \boldsymbol{D_1}\rangle\ \text{in} \\
\qquad\qquad\qquad \text{let}\ \langle \boldsymbol{D'_2}\rangle = \mathsf{id1}\ \boldsymbol{A}\ \langle \boldsymbol{D_2}\rangle\ \text{in} \\
\qquad\qquad\qquad \langle \mathbf{impe}\ \boldsymbol{D'_1}\ \boldsymbol{D'_2}\rangle) \\
\quad \mid \boldsymbol{A} \qquad\quad \mapsto\ (\text{fn}\ \langle \boldsymbol{d}^{\#}\rangle \ \mapsto\ \langle \boldsymbol{d}\rangle)
\end{array}
$$

The $\mathsf{id1}$ function from Example 4.1.1 performs case analysis over objects of type $\mathbf{nd}\ \boldsymbol{A}$. We provide cases for **impi**, **impe**, and parameters $\boldsymbol{d}^{\#}$. However, before performing the case analysis over the type $\mathbf{nd}\ \boldsymbol{A}$, we must first refine the $\boldsymbol{A}$. For clarity, we will illustrate the typing of the **impi** case. Recall that we start out writing a function of type $\forall \boldsymbol{A}{\in}\mathbf{o}.\ \langle \mathbf{nd}\ \boldsymbol{A}\rangle \supset \langle \mathbf{nd}\ \boldsymbol{A}\rangle$. The first step refines the formula to be of the form $(\boldsymbol{A} \Rightarrow \boldsymbol{B})$, which subsequently means that the body must have the type $\langle \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})\rangle \supset \langle \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})\rangle$. We then consider expressions of the form $\langle \mathbf{impi}\ (\boldsymbol{\lambda d}{:}(\mathbf{nd}\ \boldsymbol{A}).\ \boldsymbol{D}\ \boldsymbol{d})\rangle$, whose type matches the argument type above. Finally, we continue typing with the stipulation that the body of this case must also have type $\langle \mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})\rangle$.

The problem with the above function is that it does not express the correct operational behavior. If the formula is of the form $(\boldsymbol{A} \Rightarrow \boldsymbol{B})$ it may choose the first case which subsequently provides a function that *only* handles the **impi** case. However, it is possible that we have an object of type $\mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})$ whose top-level constructor is **impe** or a parameter. As we do not formally commit to an order in which the cases are executed, it is possible it will execute appropriately, but it will get stuck if it commits to the first branch. This demonstrates that we do not want to

commit to the first branch just because the formula is of the form $\boldsymbol{A} \Rightarrow \boldsymbol{B}$. In other words, we want to do case analysis on the first two arguments together. Case (1) should express that the formula is of the form $\boldsymbol{A} \Rightarrow \boldsymbol{B}$ *and* the natural deduction derivation is an **impi**. Case (2) should express that we have any formula and an **impe**. Case (3) should express that we have any formula and a parameter.

The easiest way to perform case analysis on multiple arguments is to use pairs. In ML, if one wanted to write a function of type `int -> int -> int` considering case analysis on the two arguments together, then one can just write a function of type `(int * int) -> int`. Although Delphin provides pairs, this technique does not suffice because of dependencies. The type of our `id1` function is $\forall \boldsymbol{A} \in \mathbf{o}.\ \langle \mathbf{nd}\ \boldsymbol{A} \rangle \supset \langle \mathbf{nd}\ \boldsymbol{A} \rangle$. If we tried to turn the first two arguments into a pair, then the type would be $(\exists \boldsymbol{A} \in \mathbf{o}.\ \langle \mathbf{nd}\ \boldsymbol{A} \rangle) \supset \langle \mathbf{nd}\ ? \rangle$. We use **?** to indicate that we cannot even express the type we desire as the scope of $\boldsymbol{A}$ is inside the pair and does not exist in scope of the return type. More precisely, the return type wants to be dependent on the first element of the input, which is a computation-level pair. However, recall that we do not want to allow dependencies on computation as it would render type checking undecidable.

Therefore, our solution is to generalize function types to specify a list of arguments, which we express as $\forall \overline{\alpha \in \delta}.\ \tau$. As a function takes a list of declarations, we need to extend the introduction of $\forall$ to take a list of patterns, $\overline{e} \mapsto f$. Additionally, application will now take a list of arguments, $e\ \overline{f}$.

Recall that we write $X; Y$ to refer to a two element list containing X and Y. The type of an identity function that allows us to look at both arguments together is $\forall (\boldsymbol{A} \in \mathbf{o};\ u \in \langle \mathbf{nd}\ \boldsymbol{A} \rangle).\ \langle \mathbf{nd}\ \boldsymbol{A} \rangle$. This function type specifies that the function will be called with two arguments and the result is dependent on the first. A function of this type can only be called when supplied with a list of two arguments, i.e. there

is no partial evaluation possible by applying it to just the first argument. The fact that there is no partial evaluation on the first argument makes it appear similar to specifying a function with one argument which is a pair. However, there are significant differences. Computationally, a function that takes a list of two arguments differs from a function that takes a pair of two arguments in that the former is always applied with two arguments and the latter is applied to one argument which evaluates to a pair of two elements. This distinction is necessary as we want to be able to express case analysis on multiple arguments while only permitting dependencies on LF objects.

We next show the actual identity function id2:

Example 4.1.2 (Actual Version of Identity Function (id2))**.**

$$
\begin{array}{l}
\mu\mathsf{id2} \in \forall(\boldsymbol{A}{\in}\mathbf{o};\ u{\in}\langle\mathbf{nd}\ \boldsymbol{A}\rangle).\ \langle\mathbf{nd}\ \boldsymbol{A}\rangle. \\
\quad \mathrm{fn}\ \ (\boldsymbol{(A \Rightarrow B)};\ \langle\mathbf{impi}\ (\boldsymbol{\lambda d}{:}(\mathbf{nd}\ \boldsymbol{A}).\ \boldsymbol{D}\ \boldsymbol{d})\rangle) \ \ \mapsto \\
\qquad\quad (\mathrm{case}\ (\nu\boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A})^{\#}.\ \mathsf{id2}\ (\mathbf{B};\ \langle\boldsymbol{D}\ \boldsymbol{d}\rangle)) \\
\qquad\qquad\quad \mathrm{of}\ (\nu\boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A})^{\#}.\ \langle\boldsymbol{D'}\ \boldsymbol{d}\rangle)\ \mapsto\ \langle\mathbf{impi}\ (\boldsymbol{\lambda d}.\ \boldsymbol{D'}\ \boldsymbol{d})\rangle) \\
\\
\quad \mid\ (\boldsymbol{B};\ \langle\mathbf{impe}\ \boldsymbol{D_1}\ \boldsymbol{D_2}\rangle)\ \mapsto \\
\qquad\quad \mathrm{let}\ \langle\boldsymbol{D'_1}\rangle = \mathsf{id2}\ (\boldsymbol{(A \Rightarrow B)};\ \langle\boldsymbol{D_1}\rangle)\ \mathrm{in} \\
\qquad\quad \mathrm{let}\ \langle\boldsymbol{D'_2}\rangle = \mathsf{id2}\ (\boldsymbol{A};\ \langle\boldsymbol{D_2}\rangle)\ \mathrm{in} \\
\qquad\quad \langle\mathbf{impe}\ \boldsymbol{D'_1}\ \boldsymbol{D'_2}\rangle \\
\\
\quad \mid\ (\boldsymbol{A};\ \langle\boldsymbol{d}^{\#}\rangle)\ \ \mapsto\ \ \langle\boldsymbol{d}\rangle
\end{array}
$$

Example 4.1.2 defines id2 which differs from id1 (Example 4.1.1) by specifying the first two arguments together. Recursive calls to id2 are supplied with two-element lists. By looking at the first two arguments together, this function will always exhibit the operational behavior desired. Furthermore, Delphin can reason that this function is total. ML and Haskell provide convenient syntax to specify a curried function which examines its arguments together by turning it into a function over pairs. Similarly in Delphin, one can write a curried function that examines its arguments

together by turning it into a function over lists. For example, we can write an identity function of type $\forall \boldsymbol{A}{\in}\mathbf{o}.\ \langle \mathbf{nd}\ \boldsymbol{A}\rangle \supset \langle \mathbf{nd}\ \boldsymbol{A}\rangle$ by doing fn $\boldsymbol{A} \mapsto$ fn $\langle \boldsymbol{D}\rangle \mapsto \mathsf{id2}\ (\boldsymbol{A};\ \langle \boldsymbol{D}\rangle)$.

4.1.3 Implicitness

Recall that the type of $\mathsf{id2}$ is $\forall(\boldsymbol{A}{\in}\mathbf{o};\ u{\in}\langle \mathbf{nd}\ \boldsymbol{A}\rangle).\ \langle \mathbf{nd}\ \boldsymbol{A}\rangle$. One important observation is that the formula (first argument) is easily inferable from the second argument as the type of the second argument tells us exactly what the first argument must be. Additionally, as application requires the first two arguments to be supplied together, it is redundant to explicitly specify the first argument. Therefore, just as we omitted leading Π's from LF constants, we will allow the omission of arguments that occur indexed in other inputs as they can be inferred.

With this convention, we can simplify our presentation of $\mathsf{id2}$ by writing:

Example 4.1.3 (Sugared Version of Identity Function ($\mathsf{id3}$))**.**

$$
\begin{array}{l}
\mu\mathsf{id3} \in \langle \mathbf{nd}\ \boldsymbol{A}\rangle \supset \langle \mathbf{nd}\ \boldsymbol{A}\rangle. \\
\quad \text{fn}\ \ \langle \mathbf{impi}\ (\boldsymbol{\lambda d}{:}(\mathbf{nd}\ \boldsymbol{A}).\ \boldsymbol{D}\ \boldsymbol{d})\rangle \ \ \mapsto \\
\qquad\quad (\text{case}\ (\nu \boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A})^{\#}.\ \mathsf{id3}\ \langle \boldsymbol{D}\ \boldsymbol{d}\rangle) \\
\qquad\qquad \text{of}\ (\nu \boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A})^{\#}.\ \langle \boldsymbol{D'}\ \boldsymbol{d}\rangle) \ \mapsto\ \langle \mathbf{impi}\ (\boldsymbol{\lambda d}.\ \boldsymbol{D'}\ \boldsymbol{d})\rangle) \\
\quad \mid\ \langle \mathbf{impe}\ \boldsymbol{D_1}\ \boldsymbol{D_2}\rangle \ \mapsto \\
\qquad\quad \text{let}\ \langle \boldsymbol{D_1'}\rangle = \mathsf{id3}\ \langle \boldsymbol{D_1}\rangle\ \text{in} \\
\qquad\quad \text{let}\ \langle \boldsymbol{D_2'}\rangle = \mathsf{id3}\ \langle \boldsymbol{D_2}\rangle\ \text{in} \\
\qquad\quad \langle \mathbf{impe}\ \boldsymbol{D_1'}\ \boldsymbol{D_2'}\rangle \\
\quad \mid\ \langle \boldsymbol{d}^{\#}\rangle \ \ \mapsto \ \ \langle \boldsymbol{d}\rangle
\end{array}
$$

Example 4.1.3 shows the function $\mathsf{id3}$, which is declared to have type $\langle \mathbf{nd}\ \boldsymbol{A}\rangle \supset \langle \mathbf{nd}\ \boldsymbol{A}\rangle$, where the variable $\boldsymbol{A}$ occurs free. Free variables in types are implicitly attached (using lists) to the argument where they first occur. Therefore, under-the-hood, $\mathsf{id3}$ is exactly the same as $\mathsf{id2}$ and has the type $\forall(\boldsymbol{A}{\in}\mathbf{o};\ u{\in}\langle \mathbf{nd}\ \boldsymbol{A}\rangle).\ \langle \mathbf{nd}\ \boldsymbol{A}\rangle$. Note that this function is truly identical to $\mathsf{id2}$. The implicit argument exists in all

patterns and in all applications, i.e. it is just a frontend convenience. For example, if $\boldsymbol{D}\in\mathbf{(nd}\ \boldsymbol{B}\mathbf{)}$, then the expression $\mathsf{id3}\ \langle \boldsymbol{D} \rangle$ is actually referring under-the-hood to $\mathsf{id3}\ (\boldsymbol{B};\ \langle \boldsymbol{D} \rangle)$.

It is also important to note that our generalization to lists is only necessary for performing case analysis over multiple arguments when one is indexed in another. If the arguments are not related, then we can use pairs. However, our above convention allows us to omit the dependent argument and specify the type using free variables. Therefore, throughout this chapter, none of our remaining examples will contain lists but they will specify types with free variables. However, the reader should keep in mind that the actual types do not contain free variables as every free variable is implicitly quantified (using lists) with the argument where they first occur.

4.1.4 Comparison with SimpleDelphin

All other constructs in Figure 4.1 are exactly the same as in the simply-typed version of Chapter 3, Figure 3.1. We also employ the same abbreviations from Figure 3.2 which defines the types $\langle \boldsymbol{A} \rangle$ and $\langle \boldsymbol{A}^{\#} \rangle$, as well as case statements, let statements, and the "with" syntax.

Delphin's most important feature is the dynamic creation of parameters via $\nu \boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e$. The type of this construct is $\nabla \boldsymbol{x}\in\boldsymbol{A}^{\#}.\ \tau$. Note that the only difference between this dependent version and the simply-typed version is that now the $\boldsymbol{x}$ can occur in τ.

Delphin truly is a conservative extension of the simply-typed variant from Chapter 3. The typing and execution of our simply-typed examples does not change. Hence, we will assume that the reader is familiar with writing programs in the simply-typed framework and focus on our extensions to handle dependencies.

4.2 Static Semantics

Before showing some advanced examples using dependencies, we will discuss the static semantics of Delphin. In extending SimpleLF to LF (Chapter 2) we needed to address that the introduction of dependencies entails that some types may not be valid. Similarly, in Section 4.2.1, we will discuss the well-formedness of types δ and contexts Ω. The typing of expressions now utilizes substitutions, so we will first expand our substitutions to dependencies in Section 4.2.2. Finally, we will present the new typing rules in Section 4.2.3.

When appealing to LF judgments, we will need to convert computation-level contexts Ω into LF contexts $\mathbf{\Gamma}$. As in the simply-typed variant, we define $\|\Omega\|$ (Definition 3.4.1) to perform this operation by throwing out all declarations $u{\in}\tau$ and mapping $\boldsymbol{x}{\in}\boldsymbol{A}$ and $\boldsymbol{x}{\in}\boldsymbol{A}^{\#}$ and $\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ all into $\boldsymbol{x}{:}\boldsymbol{A}$.

4.2.1 Valid Types and Contexts

As we allow dependencies, not all types are valid. Figure 4.2 presents the judgment $\Omega \vdash \delta\ \mathsf{wff}$ for determining the well-formedness of types. If the type δ can be shown to be well-formed following this judgment, then we also say δ is a valid type. The well-formedness of types δ only depends on the LF part of the context.

The rule $\mathsf{LF\_wff}$ handles the type $\boldsymbol{A}$ by appealing to the LF well-formedness judgment $\mathbf{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{A} : \boldsymbol{type}$ from Chapter 2. We use Casting, $\|\Omega\|$, to convert Ω to an appropriate $\mathbf{\Gamma}$. The rule $\mathsf{param\_wff}$ specifies that any valid type $\boldsymbol{A}$ is also a valid type $\boldsymbol{A}^{\#}$. The rest of the rules handle computational types τ by inductively traversing the type and hence we see that the actual validity of types can only depend on the LF portion of the context used in $\mathsf{LF\_wff}$.

We will consider typing only in well-formed contexts Ω, which means that all types

$$\frac{\|\Omega\| \vdash^{\mathrm{lf}} \boldsymbol{A} : \boldsymbol{type}}{\Omega \vdash \boldsymbol{A}\ \mathsf{wff}}\ \mathsf{LF\_wff} \qquad \frac{\Omega \vdash \boldsymbol{A}\ \mathsf{wff}}{\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}}\ \mathsf{param\_wff}$$

$$\frac{}{\Omega \vdash \mathrm{unit}\ \mathsf{wff}}\ \mathrm{unit\_wff} \qquad \frac{\Omega \vdash \delta\ \mathsf{wff} \quad \Omega, \alpha{\in}\delta \vdash \tau\ \mathsf{wff}}{\Omega \vdash \exists \alpha{\in}\delta.\ \tau\ \mathsf{wff}}\ \exists\mathsf{wff}$$

$$\frac{\Omega \vdash \tau\ \mathsf{wff}}{\Omega \vdash \forall nil.\ \tau\ \mathsf{wff}}\ \forall_b\mathsf{wff} \qquad \frac{\Omega \vdash \delta_1\ \mathsf{wff} \quad \Omega, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}{\Omega \vdash \forall \alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}\ \forall_i\mathsf{wff}$$

$$\frac{\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff} \quad \Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}}{\Omega \vdash \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}}\ \nabla\mathsf{wff}$$

Figure 4.2: Well-Formedness of Delphin Types

$$\frac{}{\cdot\ \mathsf{ctx}}\ \mathsf{ctxEmpty} \qquad \frac{\Omega\ \mathsf{ctx} \quad \Omega \vdash \delta\ \mathsf{wff}}{\Omega, \alpha{\in}\delta\ \mathsf{ctx}}\ \mathsf{ctxAdd} \qquad \frac{\Omega\ \mathsf{ctx} \quad \Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}}{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}\ \mathsf{ctx}}\ \mathsf{ctxAddNew}$$

Figure 4.3: Well-Formed Contexts

in Ω are well-formed. We write $\Omega\ \mathsf{ctx}$ for the judgment that checks that Ω is well-formed. This judgment just makes a straightforward appeal to our well-formedness judgment, and is presented in Figure 4.3.

We will next discuss the typing rules for substitutions (Section 4.2.2) and expressions (Section 4.2.3). For the sake of brevity and clarity we simply stipulate that all types are valid. This is guaranteed by implicit appeals to the validity judgments defined in this section. We make this explicit in Appendix A.2 as well as in the corresponding Technical Report (Poswolsky and Schürmann 2007).

$$\frac{}{\cdot \vdash \cdot : \cdot} \qquad \frac{\Omega' \vdash \omega : \Omega \quad \Omega' \vdash e \in \delta[\omega]}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}$$

$$\frac{\Omega' \vdash \omega : \Omega}{(\Omega', \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})} * \qquad \frac{\Omega' \vdash \omega : \Omega}{\Omega', \alpha{\in}\delta \vdash \uparrow_{\alpha} \omega : \Omega}$$

. . . .

$$\frac{}{\cdot \vdash^{\mathbf{lf}} \cdot : \cdot} \qquad \frac{\boldsymbol{\Gamma'} \vdash^{\mathbf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma} \quad \boldsymbol{\Gamma'} \vdash^{\mathbf{lf}} \boldsymbol{M} \in \boldsymbol{A}[\boldsymbol{\gamma}]}{\boldsymbol{\Gamma'} \vdash^{\mathbf{lf}} (\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) : (\boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A})} \qquad \frac{\boldsymbol{\Gamma'} \vdash^{\mathbf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}}{\boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathbf{lf}} \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}}$$

Figure 4.4: Substitution Typing (Delphin and LF)

4.2.2 Substitutions

We now turn to substitutions. Substitutions will appeal to our main typing judgment $\Omega \vdash e \in \delta$ (to be discussed in Section 4.2.3), which also uses substitutions.

Substitutions in our dependently-typed version are virtually identical to the simply-typed variant in Section 3.4.2, except we now apply the substitution to the types.

The abstract syntax of substitutions does not change (Def. 3.4.3) and the typing rules are presented in Figure 4.4. It may be useful to compare these rules to the simply-typed variant in Figure 3.4. The judgment $\Omega' \vdash \omega : \Omega$ states that the substitution ω brings one from the domain Ω to the codomain Ω'. Substitution application is expressed as $e[\omega]$ and $\delta[\omega]$ which will be discussed in Section 4.3. The substitution can instantiate pattern variables ($\alpha{\in}\delta$) inside Ω, but extensions to the signature $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$ can only be renamed. Additionally, we also extend the LF judgment $\boldsymbol{\Gamma'} \vdash^{\mathbf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}$ to include dependencies.

The interesting rule handles extensions to the signature, which is marked with an $*$. The parameter $\boldsymbol{x}$ is mapped to a parameter $\boldsymbol{x'}$, but now its type is also refined as to make sense in Ω'.

Just as in the simply-typed variant, we will need to convert computation-level substitutions ω into LF substitutions $\boldsymbol{\gamma}$. Analogous to our operation on contexts, we call this operation casting and define $\|\omega\|$ to throw out all computation-level aspects of the substitution yielding an appropriate LF substitution $\boldsymbol{\gamma}$. The definition of casting does not change from the simply-typed variant (Def. 3.4.4).

We write id_Ω and id_Γ for identity substitutions, whose definition does not change (Def. 3.4.5). Recall that $\Omega \vdash \mathrm{id}_\Omega : \Omega$ and $\boldsymbol{\Gamma} \Vdash^{\mathbf{f}} \mathbf{id}_{\boldsymbol{\Gamma}} : \boldsymbol{\Gamma}$.

Additionally, substitution composition $\omega_1 \circ \omega_2$, and $\gamma_1 \circ \gamma_2$ does not change from our simply-typed version (Def. 3.4.6). Recall that substitution composition is reversed from the standard function composition. If $\Omega_1 \vdash \omega_1 : \Omega_0$ and $\Omega_2 \vdash \omega_2 : \Omega_1$, then $\Omega_2 \vdash \omega_1 \circ \omega_2 : \Omega_0$.

4.2.3 Type System

We write $\Omega \vdash e \in \delta$ for the central derivability judgment, which we present in Figure 4.5. We will summarize the rules and emphasize those that have changed from SimpleDelphin (Chapter 3).

As in the simply-typed variant, the rule isLF states that the only expressions of type $\boldsymbol{A}$ are LF objects $\boldsymbol{M}$, which are typed using the LF typing judgment under $\|\Omega\|$ (Definition 3.4.1).

The variable rules $\tau\mathsf{var}$ and $\mathsf{var}^{\#}$ allow one to use assumptions in the context of types τ and $\boldsymbol{A}^{\#}$, respectively. The only expression of type $\boldsymbol{A}^{\#}$ is a variable $\boldsymbol{x}$. If $\boldsymbol{x}$ has type $\boldsymbol{A}^{\#}$, then it also has type $\boldsymbol{A}$ by isLF, reflecting that all parameters are also objects.

We now turn to the introduction and elimination of computational types τ. Function types are introduced via a list of cases $\overline{c}$. The introduction rule impl simply states that all branches must have the same type, which is the same as in SimpleDelphin

$$\frac{\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\,\mathsf{isLF} \qquad \frac{}{\Omega, u{\in}\tau, \Omega_2 \vdash u \in \tau}\,\tau\mathsf{var} \qquad \frac{((\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\,\mathsf{var}^{\#}$$

$$\frac{\text{for all } {}_{c_i \in \overline{c}}(\Omega \vdash c_i \in \tau)}{\Omega \vdash \text{fn } \overline{c} \in \tau}\,\mathsf{impI} \qquad \frac{\Omega \vdash e \in \forall\overline{\alpha{\in}\delta}.\ \tau \quad \Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta}}{\Omega \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]}\,\mathsf{impE}$$

$$\frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\,\mathsf{new} \qquad \frac{\Omega \vdash e \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]}\,\mathsf{pop}$$

$$\frac{\Omega \vdash e \in \delta \quad \Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]}{\Omega \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\,\mathsf{pairI} \qquad \frac{\Omega, u{\in}\tau \vdash e \in \tau}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\,\mathsf{fix} \qquad \frac{}{\Omega \vdash () \in \text{unit}}\,\mathsf{top}$$

. . . . . .

$$\frac{\Omega, \alpha{\in}\delta \vdash c \in \tau}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\,\mathsf{cEps} \qquad \frac{\Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}] \quad \Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta}}{\Omega \vdash \overline{e} \mapsto f \in \forall\overline{\alpha{\in}\delta}.\ \tau}\,\mathsf{cMatch}$$

$$\frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\,\mathsf{cNew} \qquad \frac{\Omega \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]}\,\mathsf{cPop}$$

Figure 4.5: Delphin Typing Rules

except that we employ our list notation. A function that ranges over an empty list of cases can be used to build up parameter functions (as in SimpleDelphin) but may also express impossibility when a function ranges over an empty type. For example, we defined a judgment to determine equality of λ-expressions, which was represented using the type family **equiv** (Example 2.2.6). For any τ, a reasonable function of type $\langle \mathbf{equiv}\ \boldsymbol{P}\ (\mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2})\ (\mathbf{var'}\ \mathbf{one}) \rangle \supset \tau$ is "fn $\cdot$" as the input type is empty since an application should not be equivalent to a variable.

Functions are eliminated through impE. We will first discuss this rule considering application on just one argument, $e\ f$. By inversion, we can see that if e has type $\forall\alpha{\in}\delta.\ \tau$ and f has type δ, then $e\ f$ has type $\delta[\mathrm{id}_\Omega, f/\alpha]$. The elimination refines τ under a substitution replacing all occurrences of α by f. How-

ever, as functions are applied to lists, the actual rule is generalized to lists. If e has type $\forall\overline{\alpha\in\delta}.\ \tau$ and it is applied to $\overline{f}$, then we need to perform an application for all elements in $\overline{f}$. We check that $\overline{f}$ contains appropriate instantiations for $\overline{\alpha\in\delta}$, by appealing to the judgment $\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha\in\delta}$, which was defined in Section 4.2.2. An alternative way of capturing applications of multiple arguments is to utilize a spine notation . However, as we already have a mechanism to instantiate multiple arguments at once, we reuse the substitution-typing judgment. For example, consider the $\mathsf{id2}$ function from Example 4.1.2. The type of this function is $\forall(\boldsymbol{A}\in\mathbf{o};\ u\in\langle\mathbf{nd}\ \boldsymbol{A}\rangle).\ \langle\mathbf{nd}\ \boldsymbol{A}\rangle$. In the context Ω, the application $\mathsf{id2}\ (\boldsymbol{(B_1 \Rightarrow B_2)};\ \langle\boldsymbol{D_1}\rangle)$ checks $\Omega \vdash \mathrm{id}_\Omega, \frac{\boldsymbol{B_1 \Rightarrow B_2}}{\boldsymbol{A}}, \frac{\langle\boldsymbol{D_1}\rangle}{u} : \Omega, \boldsymbol{A}\in\mathbf{o}, u\in\langle\mathbf{nd}\ \boldsymbol{A}\rangle$. This in turn behaves as if we performed two single applications as the return type is $\langle\mathbf{nd}\ \boldsymbol{(B_1 \Rightarrow B_2)}\rangle$ and it checks (1) $\Omega \vdash \boldsymbol{(B_1 \Rightarrow B_2)} \in \mathbf{o}$ and (2) $\Omega \vdash \langle\boldsymbol{D_1}\rangle \in \langle\mathbf{nd}\ \boldsymbol{(B_1 \Rightarrow B_2)}\rangle$.

The introduction of ∇ is called new which has not changed from its simply-typed counterpart. However, when we consider the type $\nabla\boldsymbol{x'}\in\boldsymbol{A}^{\#}.\ \tau$, it is now possible for $\boldsymbol{x'}$ to occur free in τ. Therefore, the elimination of this type via $e\backslash\boldsymbol{x}$ (pop) refines τ by renaming the hypothetical parameter $\boldsymbol{x'}$ with the actual $\boldsymbol{x}$ in the context.

Pairs $(e,\ f)$ have type $\exists\alpha\in\delta.\ \tau$, indicating that the type of the second argument depends on the first. Therefore, pairI checks that f has type τ refined with the substitution $(\mathrm{id}_\Omega, e/\alpha)$. Pairs are eliminated via case analysis. It is important to note that dependent pairs cannot be eliminated with `fst` and `snd` computation-level constructs as the type of f would then be dependent on a computation.

The typing rules for cases are updated appropriately. The introduction of pattern variables via cEps has not changed. The use of ν over cases (cNew) has not changed, but the elimination form (cPop) has been updated to refine the type as was done with its counterpart over expressions (pop). Just as application needed to handle a

list of arguments, we analogously handle a list of patterns in cMatch.

Finally, the typing rules for recursion (fix) and unit (top) are standard and do not change from their simply-typed counterparts.

4.3 Dynamic Semantics

The operational behavior of Delphin is virtually identical to its simply-typed counterpart SimpleDelphin (Chapter 3).

Definition 4.3.1 (Values)**.** *The set of values are:*

$$\textit{Values:}\quad v \quad ::= \quad () \mid \mathrm{fn}\ \overline{c} \mid \nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ v \mid (v_1,\ v_2) \mid \boldsymbol{M}$$

The values in Delphin (Def. 4.3.1) do not change from the simply-typed variant (Def 3.5.1). The only difference is that we formally express a list of case branches as $\overline{c}$ instead of $(c_1 \mid \ldots \mid c_n)$.

We present the small-step operational semantics, $\Omega \vdash e \rightarrow f$, in Figure 4.6. We execute programs in a context Ω which indicates the extensions to the signature in which evaluation is occurring. In other words, Ω is empty with respect to pattern variable definitions $\alpha {\in} \delta$.

The evaluation of ν does not change. The expression $\nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ e$ evaluates e under the context extended with $\boldsymbol{x}$, indicating an extension to the signature. Additionally, the evaluation of $e' \backslash \boldsymbol{x}$ is much like an application. We first evaluate e' down to $\nu \boldsymbol{x'} {\in} \boldsymbol{A}^{\#}.\ e$ and then substitute $\boldsymbol{x}$ for $\boldsymbol{x'}$. Although we show a versatile small-step semantics, we could follow an evaluation strategy that first evaluates e' all the way to a value. With this evaluation strategy, we see that this operation differs from function application in that we would not need to continue the evaluation of $e[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega}, \boldsymbol{x}/\boldsymbol{x'}]$

$$\frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \to f}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \to \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ f} \qquad \frac{\Omega \vdash e \to e'}{\Omega \vdash (e,\ f) \to (e',\ f)} \qquad \frac{\Omega \vdash f \to f'}{\Omega \vdash (e,\ f) \to (e,\ f')}$$

$$\frac{\Omega \vdash e \to e'}{\Omega \vdash e\ \overline{f} \to e'\ \overline{f}} \qquad \frac{\Omega \vdash f_i \to f_i'}{\Omega \vdash e\ (\overline{f_1}; f_i; \overline{f_n}) \to e\ (\overline{f_1}; f_i'; \overline{f_n})} \qquad \frac{\Omega \vdash e \to f}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \to f\backslash\boldsymbol{x}}$$

$$\frac{}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ e)\backslash\boldsymbol{x} \to e[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]}$$

$$\frac{}{\Omega \vdash (\mathrm{fn}\ \overline{c})\backslash\boldsymbol{x} \to \mathrm{fn}\ (\overline{c}\backslash\boldsymbol{x})}$$

$$\frac{\Omega \vdash c_i \overset{*}{\to} (\overline{v} \mapsto e)}{\Omega \vdash (\mathrm{fn}\ (\overline{c_1}; c_i; \overline{c_n}))\ \overline{v} \to e}\ * \qquad \frac{}{\Omega \vdash \mu u{\in}\tau.\ e \to e[\mathrm{id}_\Omega, \mu u{\in}\tau.\ e/u]}$$

. . . . . .

$$\frac{\Omega \vdash v \in \delta}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \to c[\mathrm{id}_\Omega, v/\alpha]} \qquad \frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \to c'}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \to \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c'} \qquad \frac{\Omega \vdash c \to c'}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \to c'\backslash\boldsymbol{x}}$$

$$\frac{}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ c)\backslash\boldsymbol{x} \to c[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]} \qquad \frac{\Omega \vdash e_i \to e_i'}{\Omega \vdash ((\overline{e_1}; e_i; \overline{e_n}) \mapsto f) \to ((\overline{e_1}; e_i'; \overline{e_n}) \mapsto f)}$$

Figure 4.6: Delphin Small-Step Operational Semantics

as it will already be a value.

The only difference with the operational semantics from the simply-typed counterpart is in its handling of lists. As the application $e\ \overline{f}$ takes a list of arguments, we allow the evaluation of any argument f_i in $\overline{f}$. As the ∇ type can be introduced in cases (via cNew), we may encounter an expression fn $(c_1 \mid \ldots \mid c_n)\backslash\boldsymbol{x}$, which we evaluate by pushing the $\backslash$ into the cases, i.e. it evaluates to fn $(c_1\backslash\boldsymbol{x} \mid \ldots \mid c_n\backslash\boldsymbol{x})$. As we represent a list of cases as $\overline{c}$, we use the syntax $\overline{c}\backslash\boldsymbol{x}$ to refer to this operation of pushing the $\backslash$ into all elements of the list.

The small-step operational semantics for cases, $\Omega \vdash c \to c'$, is also shown in Figure 4.6. These rules do not change from their simply-typed versions except that we now have lists of patterns and hence the evaluation of patterns allows one to evaluate any pattern in the list. Although we allow patterns to be any arbitrary expression e, we will restrict their form in Chapter 5 when we prove coverage. Generally, patterns

will be variables or values, except that they may also use the $\backslash$ construct as we saw in Chapter 3, Lemma 3.4.2.

The instantiation of pattern variables is expressed in the first rule by nondeterministically choosing an arbitrary v. In the implementation, the choice of v is delayed until matching by using an eigenvariable.

The actual execution of functions occurs in the rule marked with * which also has not changed except for utilizing a list notation. The judgment $\Omega \vdash c \xrightarrow{*} c'$ refers to the transitive closure (0 or more steps) of the $\rightarrow$ judgment. Therefore, given a list of cases $\overline{c}$, if we can reduce $c_i \in \overline{c}$ so that the patterns match the arguments, we can then continue with the body of the resulting case branch.

Finally, since we have dependent types, we define substitution application on types in Figure 4.7. When applying a substitution ω to $\exists \alpha{\in}\delta.\ \tau$, we must respect that τ makes sense with the domain extended by $\alpha{\in}\delta$. Therefore, τ is applied to the substitution $(\uparrow_\alpha \omega, \alpha/\alpha)$ which adds an α to the codomain and handles the additional α in the domain by mapping α to α. In the type $\forall \overline{\alpha{\in}\delta}.\ \tau$, the type τ makes sense where the domain is extended by all the declarations in $\overline{\alpha{\in}\delta}$. Therefore, we define $\omega + \overline{\alpha{\in}\delta}$ which extends the substitution in exactly the same fashion for all elements in the declaration list. Additionally, substitution application on expressions is given in Figure 4.8, which is very similar to its simply-typed variant except that it applies the substitution to types and is extended with our list generalization. We write $\overline{e}[\omega]$ and $\overline{c}[\omega]$ for the operation of applying the substitution ω to each element in the list.

4.4 Meta-Theoretic Results

We summarize some meta-theoretic results of the Delphin system. In Chapter 5 we will give a rigorous account of the following properties, but we will first extend the

Types

$$\begin{array}{lcl}
\boldsymbol{A}^{\#}[\omega] & = & \boldsymbol{A}[\omega]^{\#} \\
\boldsymbol{A}[\omega] & = & \boldsymbol{A}[\|\omega\|] \\
\mathrm{unit}[\omega] & = & \mathrm{unit} \\
(\forall\overline{\alpha{\in}\delta}.\ \tau)[\omega] & = & \forall\overline{\alpha{\in}\delta}[\omega].\ \tau[\omega + \overline{\alpha{\in}\delta}] \\
(\exists\alpha{\in}\delta.\ \tau)[\omega] & = & \exists\alpha{\in}\delta[\omega].\ \tau[\uparrow_{\alpha}\omega, \alpha/\alpha] \\
(\nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)[\omega] & = & \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega].\ \tau[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}]
\end{array}$$

LF Types

$$\begin{array}{lcl}
\boldsymbol{a}[\gamma] & = & \boldsymbol{a} \\
(\boldsymbol{\Pi x{:}A.\ B})[\gamma] & = & \boldsymbol{\Pi x{:}A}[\gamma].\ \boldsymbol{B}[\uparrow_{\boldsymbol{x}}\gamma, \boldsymbol{x}/\boldsymbol{x}] \\
(\boldsymbol{A\ M})[\gamma] & = & (\boldsymbol{A}[\gamma])\ (\boldsymbol{M}[\gamma])
\end{array}$$

LF Kinds

$$\begin{array}{lcl}
\boldsymbol{type}[\gamma] & = & \boldsymbol{type} \\
(\boldsymbol{\Pi x{:}A.\ K})[\gamma] & = & \boldsymbol{\Pi x{:}A}[\gamma].\ \boldsymbol{K}[\uparrow_{\boldsymbol{x}}\gamma, \boldsymbol{x}/\boldsymbol{x}]
\end{array}$$

Declaration Lists

$$\begin{array}{lcl}
\mathit{nil}[\omega] & = & \mathit{nil} \\
(\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta})[\omega] & = & \alpha_1{\in}\delta_1[\omega]; \\
 & & (\overline{\alpha{\in}\delta}[\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1])
\end{array}$$

Extending Substitutions by Decs.

$$\begin{array}{lcl}
\omega + \mathit{nil} & = & \omega \\
\omega + (\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}) & = & (\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1) + \overline{\alpha{\in}\delta}
\end{array}$$

Figure 4.7: Delphin Substitution Application on Types

system so that coverage is built into the typing rules. In other words, we will make explicit what it means for a list of cases to be exhaustive. The Delphin implementation (Chapter 6) initially types programs using the system described in this Chapter, but subsequently it checks if programs conform to the further restrictions we make for coverage. If a list of cases is not exhaustive, then Delphin returns an appropriate warning message, i.e. *Match Non-Exhaustive Warning*.

Type preservation (Theorem 4.4.3) holds irregardless of coverage, but progress (Theorem 4.4.5) does not hold as programs will get stuck when no cases match. The interested reader may see the Delphin Technical Report (Poswolsky and Schürmann 2007) for a detailed account of the meta-theory of the following properties without coverage.[1]

Lemma 4.4.1 (Weakening)**.**

If $\Omega \vdash e \in \delta$*, then* $\Omega, \Omega_2 \vdash e \in \delta$*.*

Proof. We have built-in weakening into our rules and this lemma is straightforward except that we generalize the induction hypothesis with an appropriate definition of

[1]In the Technical Report, we also supported computation-level parameters, $\tau^{\#}$, whose possible uses is left for future work.

Expressions

$$
\begin{array}{lcl}
u[\omega, e/u] & = & e \\
u[\omega, e/\alpha] \text{ where } u \neq \alpha & = & u[\omega] \\
u[\uparrow_{\alpha'} \omega] & = & u[\omega] \\
\boldsymbol{M}[\omega] & = & \boldsymbol{M}[\|\omega\|] \\
()[\omega] & = & () \\
(e\ \overline{f})[\omega] & = & (e[\omega])\ (\overline{f}[\omega]) \\
(e,\ f)[\omega] & = & (e[\omega],\ f[\omega]) \\
(\nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ e)[\omega] & = & \nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}[\omega].\ e[\uparrow_{\boldsymbol{x}} \omega, \boldsymbol{x}/\boldsymbol{x}] \\
(e \backslash \boldsymbol{x})[\omega] & = & e[\omega] \backslash \boldsymbol{x}[\omega] \\
(\mu u {\in} \tau.\ e)[\omega] & = & \mu u {\in} \tau[\omega].\ e[\uparrow_u \omega, u/u] \\
(\text{fn } \overline{c})[\omega] & = & \text{fn } (\overline{c}[\omega])
\end{array}
$$

Lists

$$
\begin{array}{lcl}
\mathit{nil}[\omega] & = & \mathit{nil} \\
(c_1; \overline{c})[\omega] & = & c_1[\omega]; \overline{c}[\omega] \\
(e_1; \overline{e})[\omega] & = & e_1[\omega]; \overline{e}[\omega]
\end{array}
$$

Cases

$$
\begin{array}{lcl}
(\epsilon \alpha {\in} \delta.\ c)[\omega] & = & \epsilon \alpha {\in} \delta[\omega].\ c[\uparrow_\alpha \omega, \alpha/\alpha] \\
(\nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ c)[\omega] & = & \nu \boldsymbol{x} {\in} \boldsymbol{A}^{\#}[\omega].\ c[\uparrow_{\boldsymbol{x}} \omega, \boldsymbol{x}/\boldsymbol{x}] \\
(c \backslash \boldsymbol{x})[\omega] & = & c[\omega] \backslash \boldsymbol{x}[\omega] \\
(\overline{e} \mapsto f)[\omega] & = & (\overline{e}[\omega] \mapsto f[\omega])
\end{array}
$$

LF Objects

$$
\begin{array}{lcl}
\boldsymbol{c}[\boldsymbol{\gamma}] & = & \boldsymbol{c} \\
\boldsymbol{x}[\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}] & = & \boldsymbol{M} \\
\boldsymbol{x}[\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x'}] \text{ where } \boldsymbol{x} \neq \boldsymbol{x'} & = & \boldsymbol{x}[\boldsymbol{\gamma}] \\
\boldsymbol{x}[\uparrow_{\boldsymbol{x'}} \boldsymbol{\gamma}] & = & \boldsymbol{x}[\boldsymbol{\gamma}] \\
(\boldsymbol{\lambda x{:}A.\ M})[\boldsymbol{\gamma}] & = & \boldsymbol{\lambda x{:}A}[\boldsymbol{\gamma}].\ \boldsymbol{M}[\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}, \boldsymbol{x}/\boldsymbol{x}] \\
(\boldsymbol{M}\ \boldsymbol{N})[\boldsymbol{\gamma}] & = & (\boldsymbol{M}[\boldsymbol{\gamma}])\ (\boldsymbol{N}[\boldsymbol{\gamma}])
\end{array}
$$

Figure 4.8: Delphin Substitution Application on Expressions

$\leq$ such that $\Omega \leq \Omega'$ means that Ω' contains everything in Ω. Details of this proof can be found in the Technical Report (Poswolsky and Schürmann 2007).

However, when we are concerned with coverage, we do not allow weakening by an arbitrary Ω_2 since a program may not be covered with respect to arbitrary extensions to the signature. Therefore, Chapter 5 will restrict weakening. □

Lemma 4.4.2 (Substitution)**.**

If $\Omega \vdash e \in \delta$ *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash e[\omega] \in \delta[\omega]$.

Proof. Chapter 5 provides details of this proof with built-in coverage. One can also see the Technical Report (Poswolsky and Schürmann 2007) for the version without coverage. However, for this Lemma, both versions are virtually identical. □

Theorem 4.4.3 (Type Preservation)**.**

If $\Omega \vdash e \in \tau$ *and* $\Omega \vdash e \to f$*, then* $\Omega \vdash f \in \tau$.

Proof. By induction on $\mathcal{E} :: \Omega \vdash e \to f$ and $\mathcal{F} :: \Omega \vdash c \to c'$. See Chapter 5 or the Technical Report (Poswolsky and Schürmann 2007) for details. Type preservation holds irregardless of coverage. □

Corollary 4.4.4 (Soundness)**.** *Parameters cannot escape their scope. If* $\Omega \vdash e \in \tau$ *and* $\Omega \vdash e \to e'$*, then all parameters in* e *and* e' *are declared in* Ω*.*

Proof. Follows directly from type preservation noting that typing in a context Ω guarantees that all encountered parameters are in Ω (easily proved by induction on the typing rules). □

Theorem 4.4.5 (Progress)**.**
Under the condition that all cases in e *are exhaustive, if* $\Omega \vdash e \in \tau$ *and* Ω *only contains declarations of the form* $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$*, then* $\Omega \vdash e \to f$ *or* e *is a value.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash e \in \tau$. In matching (rule *****) we assume that cases are exhaustive. □

This final theorem will be made rigorous by formalizing what it means for all cases to be covered. This is the most complicated part of our meta-theory and is the focus of Chapter 5.

4.5 Natural Deduction vs. Combinators

In this section we use Delphin to translate between derivations of a natural deduction calculus and derivations of an axiomatic Hilbert-style calculus, which we also called typed combinators. Formulas are represented as objects of type **o** (Example 2.2.2). Given a formula $\boldsymbol{A}$: **o**, objects of type **nd** $\boldsymbol{A}$ (Example 2.2.3) represent derivations of $\boldsymbol{A}$ in the natural deduction calculus. Similarly, objects of type **comb** $\boldsymbol{A}$ represent derivations of the same formula in our Hilbert-style calculus (Example 2.2.4).

We will write functions that translate between derivations in both calculi. Furthermore, Delphin will be able to reason that these functions are total (Chapter 5) and hence we have also proved that whatever can be proven in one calculus can also be proven in the other.

In the simply-typed variant from Section 3.7, we converted between untyped λ-expressions and untyped combinators. In this Section, one may similarly view our translations as converting between the simply-typed λ-calculus and typed combinators. In other words, our representation of derivations in the natural deduction calculus is isomorphic to a representation of the simply-typed λ-calculus where **impi** corresponds to λ-abstraction and **impe** corresponds to application.

An important observation throughout this section is that our new code is very similar to the non-dependent version from Section 3.7. However, in the previous version there was nothing stopping us from writing incorrect programs, i.e. a convert function that converted every λ-expression into the identity combinator. Dependencies serve to statically guarantee that we write meaningful translations, which is a property that could not be statically guaranteed in our earlier version.

4.5.1 From Combinators

The difficult direction is converting from natural deduction derivations into combinators as we need to get rid of variable abstractions. However, we first show the reverse direction as it is straightforward, just as in Chapter 3. The reader should keep in mind that with respect to the simply-typed λ-calculus, **impi** corresponds to function abstraction and **impe** corresponds to application. We call this function hil2nd and it is written as:

Example 4.5.1 (Combinators to Natural Deduction Derivations)**.**

μhil2nd $\in \langle \mathbf{comb}\ \boldsymbol{A} \rangle \supset \langle \mathbf{nd}\ \boldsymbol{A} \rangle$.

$$
\begin{array}{lll}
\text{fn } \langle \mathbf{K} \rangle & \mapsto & \langle \mathbf{impi}\ (\boldsymbol{\lambda x}{:}(\mathbf{nd}\ \boldsymbol{A}).\ \mathbf{impi}\ (\boldsymbol{\lambda y}{:}(\mathbf{nd}\ \boldsymbol{B}).\ \boldsymbol{x}))\rangle \\
\mid \langle \mathbf{S} \rangle & \mapsto & \langle \mathbf{impi}\ (\boldsymbol{\lambda x}{:}(\mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B} \Rightarrow \boldsymbol{C})). \\
 & & \quad \mathbf{impi}\ (\boldsymbol{\lambda y}{:}(\mathbf{nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})). \\
 & & \quad\quad \mathbf{impi}\ (\boldsymbol{\lambda z}{:}(\mathbf{nd}\ \boldsymbol{A}). \\
 & & \quad\quad\quad \mathbf{impe}\ (\mathbf{impe}\ \boldsymbol{x}\ \boldsymbol{z})\ (\mathbf{impe}\ \boldsymbol{y}\ \boldsymbol{z})))))\rangle \\
\mid \langle \mathbf{MP}\ \boldsymbol{H_1}\ \boldsymbol{H_2} \rangle & \mapsto & \text{let } \langle \boldsymbol{D_1} \rangle = \mathsf{hil2nd}\ \langle \boldsymbol{H_1} \rangle \text{ in} \\
 & & \text{let } \langle \boldsymbol{D_2} \rangle = \mathsf{hil2nd}\ \langle \boldsymbol{H_2} \rangle \text{ in} \\
 & & \quad \langle \mathbf{impe}\ \boldsymbol{D_1}\ \boldsymbol{D_2} \rangle
\end{array}
$$

This function is straightforward. The **K** and **S** combinators are mapped to equivalent natural deduction derivations and the correctness of this translation is guaranteed by the type system. For example, both **K** and its mapping represent derivations of the same formula $\boldsymbol{A} \Rightarrow \boldsymbol{B} \Rightarrow \boldsymbol{A}$, and it would not type check otherwise.

4.5.2 To Combinators

Translating from natural deduction derivations to combinators is complicated as it requires the elimination of abstractions. Our translation will follow a two-step algorithm.

The first step is bracket abstraction, or ba, which internalizes abstraction with respect to combinators. If $\boldsymbol{M}$ has type $(\mathbf{comb}\ \boldsymbol{A} \rightarrow \mathbf{comb}\ \boldsymbol{B})$, then we can use ba to get a combinator $\boldsymbol{M'}$ of type $\mathbf{comb}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})$. The use of dependent types really illustrates what this function is doing. We are converting an LF abstraction down to a combinator that is equivalent to the original abstraction in combinator logic. Subsequently, if we have a combinator $\boldsymbol{C}$ of type $\mathbf{comb}\ \boldsymbol{A}$, then the combinator $\mathbf{MP}\ \boldsymbol{M'}\ \boldsymbol{C}$ proves the same formula as the LF application $\boldsymbol{M}\ \boldsymbol{C}$. Formally, ba is

written as:

Example 4.5.2 (Bracket Abstraction)**.**

$$
\begin{array}{lll}
\mu\mathsf{ba} \in \langle \mathbf{comb}\ \boldsymbol{A} \rightarrow \mathbf{comb}\ \boldsymbol{B} \rangle \supset \langle \mathbf{comb}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \rangle. & & \\
\quad \text{fn}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{x} \rangle & \mapsto & \langle \mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \mathbf{K}) \\
& & \qquad (\mathbf{K} : \mathbf{comb}\ (\boldsymbol{A} \Rightarrow \boldsymbol{A} \Rightarrow \boldsymbol{A})) \rangle \\
\quad \mid \langle \boldsymbol{\lambda x.}\ \mathbf{MP}\ (\boldsymbol{H_1}\ \boldsymbol{x})\ (\boldsymbol{H_2}\ \boldsymbol{x}) \rangle & \mapsto & \text{let}\ \langle \boldsymbol{H_1'} \rangle = \mathsf{ba}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{H_1}\ \boldsymbol{x} \rangle\ \text{in} \\
& & \text{let}\ \langle \boldsymbol{H_2'} \rangle = \mathsf{ba}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{H_2}\ \boldsymbol{x} \rangle\ \text{in} \\
& & \quad \langle \mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \boldsymbol{H_1'})\ \boldsymbol{H_2'} \rangle \\
\quad \mid \langle \boldsymbol{\lambda x.}\ \boldsymbol{H} \rangle & \mapsto & \langle \mathbf{MP}\ \mathbf{K}\ \boldsymbol{H} \rangle
\end{array}
$$

This function appears virtually identical to the one from Chapter 3. However, our code is greatly simplified by frontend conveniences that allow us to keep things *implicit*. With respect to the LF encodings (Example 2.2.4), the combinator constructs **MP**, **S**, and **K** all contain (implicit) arguments for the formulas that they are indexed by. Similarly, the type of ba has an implicit quantification of the free variables $\boldsymbol{A}$ and $\mathbf{B}$ as discussed in Section 4.1.3.

As discussed in Example 2.2.4, the object **MP (MP S K) K** does not represent a unique derivation of $\boldsymbol{A} \Rightarrow \boldsymbol{A}$. Therefore, although we are treating formulas implicitly, we use a type ascription to make the type of the second **K** explicit so that it corresponds to an actual derivation. If we omit the type ascription, then the object under-the-hood would contain a free variable as information tagged as implicit cannot be automatically inferred, and hence Delphin would return an appropriate error message.

Next we write the function nd2hil which traverses natural deduction derivations and uses ba to convert them into combinators. In this function, we will need to introduce new parameters of **nd** $\boldsymbol{A}$ and **comb** $\boldsymbol{A}$ together. In order to hold onto the relationship between these parameters, we pass around a parameter function W of

type $\langle \mathbf{nd}\ \boldsymbol{A}^{\#} \rangle \supset \langle \mathbf{comb}\ \boldsymbol{A} \rangle$. For readability, we will employ type aliasing:

$$\text{type } \mathsf{ndParamFun} = \langle (\mathbf{nd}\ \boldsymbol{A})^{\#} \rangle \supset \langle \mathbf{comb}\ \boldsymbol{A} \rangle$$

Example 4.5.3 (Natural Deduction Derivations to Combinators)**.**

$\mu\mathsf{nd2hil} \in \mathsf{ndParamFun} \supset \langle \mathbf{nd}\ \boldsymbol{A} \rangle \supset \langle \mathbf{comb}\ \boldsymbol{A} \rangle$.
fn $W \mapsto$
fn $\langle \mathbf{impi}\ (\boldsymbol{\lambda d.}\ \boldsymbol{D}\ \boldsymbol{d}) \rangle \mapsto$
(case $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \mathsf{nd2hil}\ (W \text{ with } \langle \boldsymbol{d} \rangle \mapsto \langle \boldsymbol{h} \rangle)\ \langle \boldsymbol{D}\ \boldsymbol{d} \rangle)$
of $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \langle \boldsymbol{H}\ \boldsymbol{h} \rangle) \mapsto \mathsf{ba}\ \langle \boldsymbol{\lambda h.}\ \boldsymbol{H}\ \boldsymbol{h} \rangle)$

| $\langle \mathbf{impe}\ \boldsymbol{D_1}\ \boldsymbol{D_2} \rangle \mapsto$
let $\langle \boldsymbol{H_1} \rangle = \mathsf{nd2hil}\ W\ \langle \boldsymbol{D_1} \rangle$ in
let $\langle \boldsymbol{H_2} \rangle = \mathsf{nd2hil}\ W\ \langle \boldsymbol{D_2} \rangle$ in
$\langle \mathbf{MP}\ \boldsymbol{H_1}\ \boldsymbol{H_2} \rangle$

| $\langle \boldsymbol{d}^{\#} \rangle \mapsto W\ \langle \boldsymbol{d} \rangle$

Example 4.5.3 shows us the nd2hil function which is very much the same as the simply-typed variant exp2comb (Example 3.7.3). The first argument to nd2hil is the parameter function W, and we will need to update this function when we create new parameters. W will be called when we encounter a dynamically created parameter. In the **impi** case, we recurse on $\boldsymbol{D}$ by first creating a parameter $\boldsymbol{d}$ to which $\boldsymbol{D}$ can be applied. However, W is not defined with respect to the new $\boldsymbol{d}$. Therefore, before recursing, we create a fresh combinator $\boldsymbol{h}$ and write $(W \text{ with } \langle \boldsymbol{d} \rangle \mapsto \langle \boldsymbol{h} \rangle))$ which extends W by mapping $\boldsymbol{d}$ to $\boldsymbol{h}$. The "with" syntax has not changed from the simply-typed variant (Section 3.7.2). We omitted a good amount of type information as it is inferable, but one should keep in mind that $\boldsymbol{d}$ and $\boldsymbol{h}$ are indexed by the same formula. The expression $(\nu \boldsymbol{d} \in (\mathbf{nd}\ \boldsymbol{A})^{\#}.\ \nu \boldsymbol{h} \in (\mathbf{comb}\ \boldsymbol{A})^{\#}.\ \mathsf{nd2hil}\ (W \text{ with } \langle \boldsymbol{d} \rangle \mapsto \langle \boldsymbol{h} \rangle)\ \langle \boldsymbol{D}\ \boldsymbol{d} \rangle)$ contains a recursive call in the presence of the new $\boldsymbol{d}$ and $\boldsymbol{h}$. During the recursion, the $\boldsymbol{d}$ will be translated to $\boldsymbol{h}$ by the last case of nd2hil. Because the $\boldsymbol{h}$ may occur in

the result, we utilize higher-order matching by matching against $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \langle \boldsymbol{H}\ \boldsymbol{h}\rangle)$. The variable $\boldsymbol{H}$ is an LF function resulting from abstracting away all occurrences of $\boldsymbol{h}$ in the result. The call to ba internalizes this abstraction to create an equivalent combinator. The **impe** case recurses on both components and glues the results together with **MP**. Finally, the last case handles parameters by applying the parameter function W.

The behavior of nd2hil is very similar to our earlier exp2comb and we conclude this section with a similar sample execution.

Example 4.5.4 (Sample nd2hil Execution)**.**

nd2hil (fn $\cdot$) $\langle$**impi (**$\boldsymbol{\lambda d}$**:(nd** $\boldsymbol{p}$**).** $\boldsymbol{d}$**)**$\rangle$
$\ldots \rightarrow$ case $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.$ nd2hil $(W$ with $\langle \boldsymbol{d}\rangle \mapsto \langle \boldsymbol{h}\rangle)\ \langle \boldsymbol{d}\rangle)$
of $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \langle \boldsymbol{H}\ \boldsymbol{h}\rangle) \mapsto$ ba $\langle \boldsymbol{\lambda d}$**:(nd** $\boldsymbol{p}$**).** $\boldsymbol{H}\ \boldsymbol{d}\rangle$
$\ldots \rightarrow$ case $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ (W$ with $\langle \boldsymbol{d}\rangle \mapsto \langle \boldsymbol{h}\rangle)\ \langle \boldsymbol{d}\rangle)$
of $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \langle \boldsymbol{H}\ \boldsymbol{h}\rangle) \mapsto$ ba $\langle \boldsymbol{\lambda d}$**:(nd** $\boldsymbol{p}$**).** $\boldsymbol{H}\ \boldsymbol{d}\rangle$
$\ldots \rightarrow$ case $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \langle \boldsymbol{h}\rangle)$
of $(\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \langle \boldsymbol{H}\ \boldsymbol{h}\rangle) \mapsto$ ba $\langle \boldsymbol{\lambda d}$**:(nd** $\boldsymbol{p}$**).** $\boldsymbol{H}\ \boldsymbol{d}\rangle$
$\ldots \rightarrow$ ba $\langle \boldsymbol{\lambda h}$**:(comb** $\boldsymbol{p}$**).** $\boldsymbol{h}\rangle$
$\ldots \rightarrow \langle$**MP (MP S K) K**$\rangle$

4.6 HOAS vs. de Bruijn with Proofs

In Chapter 3 we converted between a HOAS and a de Bruijn representation of λ-expressions. In this Section, we augment our translations by returning a proof that our translation is correct. The type **exp** (Example 2.1.2) refers to our encoding utilizing HOAS. For our de Bruijn representation (Example 2.2.5) we first encode the context of free variables Φ as a natural number **nat** (Example 2.1.1). We then explicitly encode variables in Φ as objects of type **variable** $\ulcorner \Phi \urcorner$. Similarly, λ-expressions in Φ are represented as objects of type **term** $\ulcorner \Phi \urcorner$.

A de Bruijn encoding represents a variable as a number pointing to its binding. The complication of this notation is that the same variable can be represented differently depending on where it occurs. In Example 2.1.4, our representation function needed to pass around a function that mapped object-level variables to de Bruijn indices. This function needed to be updated whenever we encounter a new binding so that the newest binding was mapped to 1 and all other mappings were incremented. Similarly, our translation algorithm will need to pass around an analogous function.

In order to make sure our translation function translates λ-expressions correctly, our functions will return a proof that the input is *equivalent* to the output by producing derivations in our equivalence judgment $\Phi \vdash e \equiv f$ represented as objects of type **equiv** $\ulcorner \Phi \urcorner \ulcorner e \urcorner \ulcorner f \urcorner$ (Example 2.2.6). This also highlights the usefulness of dependent types in guaranteeing properties of programs. Our dependently-typed version here has the same structure as the simply-typed version, but the requirement to return a proof that the λ-expressions are equal disallows us from writing meaningless functions such as ones that always return the identity function.

4.6.1 Converting HOAS to de Bruijn

Example 4.6.1 (HOAS to de Bruijn)**.**

$$
\begin{array}{l}
\mu\mathsf{toDebruijn} \in \forall \boldsymbol{P}{\in}\mathbf{nat}.\ (\forall \boldsymbol{E}{\in}\mathbf{exp}^{\#}.\ \exists \boldsymbol{X}{\in}(\mathbf{variable}\ \boldsymbol{P}).\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ (\mathbf{var'}\ \boldsymbol{X})\rangle) \\
\qquad\qquad\qquad\qquad \supset \forall \boldsymbol{E}{\in}\mathbf{exp}.\ \exists \boldsymbol{T}{\in}(\mathbf{term}\ \boldsymbol{P}).\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ \boldsymbol{T}\rangle. \\
\quad \mathrm{fn}\ \boldsymbol{P} \mapsto \mathrm{fn}\ W \mapsto \\
\qquad \mathrm{fn}\ \ \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x})\rangle \ \ \mapsto \\
\qquad\qquad \mathrm{let}\ W' = (\mathrm{fn}\ \langle \boldsymbol{x}^{\#}\rangle \mapsto \mathrm{let}\ (\langle \boldsymbol{Y}\rangle,\ \langle \boldsymbol{D}\rangle) = W\ \langle \boldsymbol{x}\rangle\ \mathrm{in}\ (\langle \mathbf{succ}\ \boldsymbol{Y}\rangle, \langle \mathbf{eqVar}\ \boldsymbol{D}\rangle)) \\
\qquad\qquad\quad \mathrm{in}\ (\mathrm{case}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \nu \boldsymbol{d}{\in}(\mathbf{equiv}\ (\mathbf{s}\ \boldsymbol{P})\ \boldsymbol{x}\ (\mathbf{var'}\ \mathbf{one}))^{\#}. \\
\qquad\qquad\qquad\qquad \mathsf{toDebruijn}\ (\mathbf{s}\ \boldsymbol{P})\ (W'\ \mathrm{with}\ \langle \boldsymbol{x}\rangle \mapsto (\langle \mathbf{one}\rangle,\ \langle \boldsymbol{d}\rangle))\ \langle \boldsymbol{E}\ \boldsymbol{x}\rangle) \\
\qquad\qquad\qquad\quad \mathrm{of}\ (\nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\langle \boldsymbol{T}\rangle,\ \langle \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d}\rangle)) \ \ \mapsto \\
\qquad\qquad\qquad\qquad (\langle \mathbf{lam'}\ \boldsymbol{T}\rangle,\ \langle \mathbf{eqLam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{\lambda d.}\ \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d})\rangle))) \\
\\
\qquad \mid\ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2}\rangle \ \ \mapsto \\
\qquad\qquad \mathrm{let}\ (\langle \boldsymbol{T_1}\rangle,\ \langle \boldsymbol{D_1}\rangle) = \mathsf{toDebruijn}\ \boldsymbol{P}\ W\ \langle \boldsymbol{E_1}\rangle\ \mathrm{in} \\
\qquad\qquad \mathrm{let}\ (\langle \boldsymbol{T_2}\rangle,\ \langle \boldsymbol{D_2}\rangle) = \mathsf{toDebruijn}\ \boldsymbol{P}\ W\ \langle \boldsymbol{E_2}\rangle\ \mathrm{in} \\
\qquad\qquad\quad (\langle \mathbf{app'}\ \boldsymbol{T_1}\ \boldsymbol{T_2}\rangle,\ \langle \mathbf{eqApp}\ \boldsymbol{D_1}\ \boldsymbol{D_2}\rangle) \\
\\
\qquad \mid\ \langle \boldsymbol{x}^{\#}\rangle \ \ \mapsto \\
\qquad\qquad \mathrm{let}\ (\langle \boldsymbol{Y}\rangle,\ \langle \boldsymbol{D}\rangle) = W\ \langle \boldsymbol{x}\rangle\ \mathrm{in} \\
\qquad\qquad\quad (\langle \mathbf{var'}\ \boldsymbol{Y}\rangle,\ \langle \boldsymbol{D}\rangle)
\end{array}
$$

Example 4.6.1 gives all the code for our function toDebruijn which converts λ-expressions in a context Φ encoded as **exp** into a first-order representation encoded as **term** $\ulcorner \Phi \urcorner$. This function takes three arguments and returns a pair containing the translation as well as a proof that the translation is equivalent to the input. The first argument $\boldsymbol{P}$ is the context which could have been made implicit, but we chose to make it explicit for illustration. The second argument is a parameter function mapping parameters to variables with proofs that they are indeed equivalent. Finally, the third (and last) argument contains the actual λ-expression we are translating, which must be a value **exp** instead of computation $\langle \mathbf{exp}\rangle$ as the result is dependent on this input.

The interesting case is in our handling of $\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x})\rangle$. We create a new $\boldsymbol{x}$ so that we can recurse on $\langle \boldsymbol{E}\ \boldsymbol{x}\rangle$. However, in order to update the parameter function

we need to also create a new $\boldsymbol{d}$ which contains a proof in our equality judgment that $\boldsymbol{x}$ is mapped to **one** in the extended context **s** $\boldsymbol{P}$. We use the "with" syntax to express this update on W', where W' increments the mappings of all parameters in W by using **succ** on the variable portion of the mapping, and **eqVar** on the derivation portion. Finally, the result of the recursive call returns the translation of $\langle \boldsymbol{E}\ \boldsymbol{x} \rangle$ with a proof that the translation is correct. Notice that the $\boldsymbol{x}$ and $\boldsymbol{d}$ cannot occur in the translation but may occur in the proof. Therefore, we utilize higher-order matching on the proof portion of the result, which then allows us to finish the **lam** case by using **lam'** on the translation and **eqLam** on the proof.

4.6.2 Converting de Bruijn to HOAS

Example 4.6.2 (de Bruijn to HOAS)**.**

$$
\begin{array}{l}
\mu\mathsf{toHOAS} \in \forall \boldsymbol{P}{\in}\mathbf{nat}.\ (\forall \boldsymbol{X}{\in}(\mathbf{variable}\ \boldsymbol{P}).\ \exists \boldsymbol{E}{\in}\mathbf{exp}.\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ (\mathbf{var'}\ \boldsymbol{X})\rangle) \\
\qquad\qquad\qquad\qquad \supset \forall \boldsymbol{T}{\in}(\mathbf{term}\ \boldsymbol{P}).\ \exists \boldsymbol{E}{\in}\mathbf{exp}.\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ \boldsymbol{T}\rangle. \\
\quad \mathrm{fn}\ \boldsymbol{P} \mapsto \mathrm{fn}\ W \mapsto \\
\qquad \mathrm{fn}\ \langle \mathbf{lam'}\ \boldsymbol{T}\rangle \ \mapsto \\
\qquad\qquad (\mathrm{case}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \nu \boldsymbol{d}{\in}(\mathbf{equiv}\ (\mathbf{s}\ \boldsymbol{P})\ \boldsymbol{x}\ (\mathbf{var'}\ \mathbf{one}))^{\#}. \\
\qquad\qquad\qquad \mathsf{toHOAS} \\
\qquad\qquad\qquad\quad (\mathbf{s}\ \boldsymbol{P}) \\
\qquad\qquad\qquad\quad ((\mathrm{fn}\ \ \nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\ \langle \mathbf{one}\rangle \ \mapsto\ (\langle \boldsymbol{x}\rangle,\ \langle \boldsymbol{d}\rangle)\ ) \\
\qquad\qquad\qquad\qquad \mid\ \nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\ \langle \mathbf{succ}\ \boldsymbol{X}\rangle \mapsto \\
\qquad\qquad\qquad\qquad\quad (\mathrm{let}\ (\langle \boldsymbol{E}\rangle,\ \langle \boldsymbol{D}\rangle) = W\ \langle \boldsymbol{X}\rangle \\
\qquad\qquad\qquad\qquad\quad\ \mathrm{in}\ \nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\langle \boldsymbol{E}\rangle,\ \langle \mathbf{eqVar}\ \boldsymbol{D}\rangle))\ \backslash \boldsymbol{x} \backslash \boldsymbol{d}\ )) \\
\qquad\qquad\qquad\qquad \backslash \boldsymbol{x} \backslash \boldsymbol{d}) \\
\qquad\qquad\qquad\quad \langle \boldsymbol{T}\rangle \\
\qquad\qquad\qquad) \\
\qquad\qquad\quad \mathrm{of}\ (\nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\langle \boldsymbol{E}\ \boldsymbol{x}\rangle,\ \langle \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d}\rangle)) \ \mapsto \\
\qquad\qquad\qquad\qquad\quad (\langle \mathbf{lam}\ (\boldsymbol{\lambda x}.\ \boldsymbol{E}\ \boldsymbol{x})\rangle,\ \langle \mathbf{eqLam}\ (\boldsymbol{\lambda x}.\ \boldsymbol{\lambda d}.\ \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d})\rangle))) \\
\\
\qquad \mid\ \langle \mathbf{app'}\ \boldsymbol{T_1}\ \boldsymbol{T_2}\rangle \ \mapsto \\
\qquad\qquad \mathrm{let}\ (\langle \boldsymbol{E_1}\rangle,\ \langle \boldsymbol{D_1}\rangle) = \mathsf{toHOAS}\ \boldsymbol{P}\ W\ \langle \boldsymbol{T_1}\rangle\ \mathrm{in} \\
\qquad\qquad \mathrm{let}\ (\langle \boldsymbol{E_2}\rangle,\ \langle \boldsymbol{D_2}\rangle) = \mathsf{toHOAS}\ \boldsymbol{P}\ W\ \langle \boldsymbol{T_2}\rangle\ \mathrm{in} \\
\qquad\qquad\quad (\langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2}\rangle,\ \langle \mathbf{eqApp}\ \boldsymbol{D_1}\ \boldsymbol{D_2}\rangle) \\
\\
\qquad \mid\ \langle \mathbf{var'}\ \boldsymbol{X}\rangle \ \mapsto\ W\ \langle \boldsymbol{X}\rangle
\end{array}
$$

Example 4.6.2 gives all the code for our function toHOAS which converts λ-expressions encoded in a context Φ a la de Bruijn into λ-expressions encoded in a representation utilizing HOAS. Similar to our last function, the first argument $\boldsymbol{P}$ is the context which we could also make implicit. The second argument W is a function that tells us how to handle variables in $\boldsymbol{P}$. Finally, the third argument is the λ-expression we want to convert to HOAS. As in the last function, the result is a pair containing the translation as well as a proof that the translation is correct.

The most interesting case is $\langle \mathbf{lam'}\ \boldsymbol{T}\rangle$. In this situation we do not need to apply $\boldsymbol{T}$ to anything, but we still need to create a new parameter $\boldsymbol{x}$ as in the body of $\boldsymbol{T}$, the

variable **one** is meant to correspond to a fresh object-level variable. Additionally, in order to update W, we need to create a new parameter $\boldsymbol{d}$ containing a proof that $\boldsymbol{x}$ is mapped to **one** in the extended context (**s** $\boldsymbol{P}$). Additionally, variable **succ X** now corresponds to W's mapping for **X**. Although W is not a parameter function and we cannot use the "with" syntax, our method of updating W still follows the same methodology and looks similar to the desugared form of the "with" construct. Finally, the result of the recursive call gives us a translation and proof, where the $\boldsymbol{x}$ can occur in both parts while the $\boldsymbol{d}$ can only occur in the latter part. We therefore utilize higher-order matching appropriately and finish this case using **lam** on the translation and **eqLam** on the proof.

4.6.3 Example Execution

We now conclude this section with a sample of converting the object-level function $\lambda x.\ \lambda y.\ x\ y$ between its representation utilizing HOAS and its representation a la de Bruijn. This is the same example we used in Chapter 3 except we now also return a proof that the translation is correct. The first argument to both functions will be **z**, indicating that the λ-expression makes sense in the empty context.

Example 4.6.3 (Sample Execution of toDebruijn and toHOAS)**.**

toDebruijn **z** (fn $\cdot$) $\langle$**lam (**$\boldsymbol{\lambda x}$**. lam (**$\boldsymbol{\lambda y}$**. app** $\boldsymbol{x}$ $\boldsymbol{y}$**))**$\rangle$
$\ldots \rightarrow$ (**lam' (lam' (app' (var' (succ one)) (var' one)))**
, $\langle$**eqLam (**$\boldsymbol{\lambda x}$**.** $\boldsymbol{\lambda d_x}$**. eqLam (**$\boldsymbol{\lambda y}$**.** $\boldsymbol{\lambda d_y}$**. eqApp (eqVar** $\boldsymbol{d_x}$**)** $\boldsymbol{d_y}$**))**$\rangle$)

toHOAS (fn $\cdot$) $\langle$**lam' (lam' (app' (var' (succ one)) (var' one)))**$\rangle$
$\ldots \rightarrow$ (**lam (**$\boldsymbol{\lambda x}$**. lam (**$\boldsymbol{\lambda y}$**. app** $\boldsymbol{x}$ $\boldsymbol{y}$**))**
, $\langle$**eqLam (**$\boldsymbol{\lambda x}$**.** $\boldsymbol{\lambda d_x}$**. eqLam (**$\boldsymbol{\lambda y}$**.** $\boldsymbol{\lambda d_y}$**. eqApp (eqVar** $\boldsymbol{d_x}$**)** $\boldsymbol{d_y}$**))**$\rangle$)

4.7 Conclusion

In this chapter we have discussed the core of the Delphin system and illustrated how it can be used to write functions over derivations in various judgments. This is achieved by extending SimpleDelphin (Chapter 3) with dependent types as to support functions over arbitrary LF encodings.

Just as we introduced implicit Π quantifications in LF, we have also introduced implicit quantifications in computation-level functions, which leads to concise code.

Finally, to show that Delphin is a type safe language we will define what it means for a list of cases to be covered in the next chapter, Chapter 5. In this chapter, we will briefly briefly talk about termination. When functions are total, i.e. covered and terminating, we can interpret our functions as proofs. We will see that the examples of this section do indeed correspond to proofs and hence we have shown that any formula derivable in our natural deduction calculus can be derived in the combinator calculus, and vice versa. Additionally, we have also proved that for any λ-expression encoded in HOAS we can convert it into an equivalent λ-expression a la de Bruijn, and vice versa.

Our implementation is discussed in Chapter 6 and the concrete code for all the examples in this chapter are available in Section 6.10.2.

Chapter 5

Meta-Theory

We have discussed the underlying type theory of the Delphin programming language. This chapter focuses on the meta-theory of Delphin, whose main contribution is twofold. First, we show that Delphin is a type safe language by proving type preservation and progress. Second, we will discuss how total functions naturally correspond to proofs. Therefore, Delphin is not only well-suited for programming/manipulating complex data, but also proves to be a valuable tool for reasoning.

In order to show progress, we present a novel *coverage* algorithm that statically checks if a list of cases is exhaustive. In first-order simply-typed languages, the issue of coverage analysis is straightforward as we may simply stipulate that a case must exist for every constructor of a given type. However, complications arise in determining coverage over dependently-typed higher-order data, as found in Delphin.

In Section 4.2.3 we discussed how an empty function can express impossibility when the function ranges over an empty (or uninhabited) type. However, determining if a type is empty may be undecidable (McBride 1999). For example, determining if a type is empty may entail proving the consistency of arbitrary logics. The Delphin website contains an example proving cut elimination for a sequent calculus whose

derivations of $\boldsymbol{F}$ are represented as **conc** $\boldsymbol{F}$. It is not obvious that **conc** $\bot$ is an empty type, as the emptiness is a result of a nontrivial cut-elimination proof.

Not surprisingly, the most novel feature of our coverage-checking algorithm is with respect to handling HOAS. We permit case analysis over any LF (representation-level) type $\boldsymbol{A}$, including functions. This is crucial for our examples of bracket abstraction (Examples 3.7.2 and 4.5.2). Although we cannot determine if an arbitrary type is inhabited, we can always generate an exhaustive list of cases to inspect terms of any type. Therefore, one may view our coverage algorithm as providing a complete elimination form for LF objects. Determining coverage for closed LF types has been studied (Schürmann and Pfenning 2003) and extended for types open with respect to special *blocks* in Twelf (Schürmann 2000). Additionally, recent work (Dunfield and Pientka 2008) defines coverage in the Beluga system (Pientka and Dunfield 2008) whose representation level is LF extended with explicit contexts and substitutions allowing the programmer to only handle closed objects. As our representation level is LF, we define coverage over LF objects open with respect to parameters, and additionally define coverage over computation-level pairs and the ∇ type.

We begin this chapter in Section 5.1 where we distinguish functions by the parameters they are expected to handle. We proceed in Section 5.2 with an update of the type system and operational semantics that reflect these distinctions. Section 5.3 gives the coverage algorithm that generates a list of exhaustive cases for any type. We discuss how Delphin can be used for reasoning (writing proofs) in Section 5.4, which we illustrate with the examples from Chapter 4 extended with world information. Finally, we summarize the proof of type safety in Section 5.5 and conclude this chapter in Section 5.6.

5.1 Worlds

The ability to dynamically create parameters, via $\nu\boldsymbol{x}.\ e$, requires us to carefully consider how we perform case analysis over a type. Rather than just consider all constructors of the desired type, we must also consider parameters. As a motivating example we will return to our examples from Chapter 3 using Delphin to implement various evaluation strategies over λ-expressions of the untyped λ-calculus encoded (using HOAS) as objects of type **exp** (Example 2.1.2).

5.1.1 Motivation and Definitions

Example 5.1.1 (Closed and Open Evaluators)**.** *To refresh the reader we just repeat Examples 3.3.4 and 3.3.5.*

$$
\begin{array}{l}
\mu\mathsf{eval} \in \langle \mathbf{exp} \rangle \supset \langle \mathbf{exp} \rangle. \\
\quad \text{fn}\ \ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle \quad \mapsto \text{case}\ (\mathsf{eval}\ \langle \boldsymbol{E_1} \rangle,\ \mathsf{eval}\ \langle \boldsymbol{E_2} \rangle) \\
\qquad\qquad\qquad\qquad\quad \text{of}\ (\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{F}\ \boldsymbol{x}) \rangle,\ \langle \boldsymbol{V} \rangle) \mapsto \mathsf{eval}\ \langle \boldsymbol{F}\ \boldsymbol{V} \rangle \\
\quad \mid\ \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle \ \mapsto \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mu\mathsf{evalBeta} \in \langle \mathbf{exp} \rangle \supset \langle \mathbf{exp} \rangle. \\
\quad \text{fn}\ \ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2} \rangle \quad \mapsto \text{case}\ (\mathsf{evalBeta}\ \langle \boldsymbol{E_1} \rangle,\ \mathsf{evalBeta}\ \langle \boldsymbol{E_2} \rangle) \\
\qquad\qquad\qquad\qquad\quad \text{of}\ (\langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{F}\ \boldsymbol{x}) \rangle,\ \langle \boldsymbol{V} \rangle) \mapsto \mathsf{evalBeta}\ \langle \boldsymbol{F}\ \boldsymbol{V} \rangle \\
\qquad\qquad\qquad\qquad\quad \mid\ (\langle \boldsymbol{F} \rangle,\ \langle \boldsymbol{V} \rangle) \qquad\quad \mapsto \langle \mathbf{app}\ \boldsymbol{F}\ \boldsymbol{V} \rangle \\
\quad \mid\ \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x}) \rangle \ \mapsto \text{case}\ (\nu\boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \mathsf{evalBeta}\ \langle \boldsymbol{E}\ \boldsymbol{x} \rangle) \\
\qquad\qquad\qquad\qquad\quad \text{of}\ \ (\nu\boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \langle \boldsymbol{E'}\ \boldsymbol{x} \rangle)\ \mapsto \langle \mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E'}\ \boldsymbol{x} \rangle \\
\quad \mid\ \langle \boldsymbol{x}^{\#} \rangle \qquad\qquad\quad \mapsto \langle \boldsymbol{x} \rangle
\end{array}
$$

Example 5.1.1 gives the code for the closed evaluator eval and the open evaluator evalBeta. Both functions have type $\langle \mathbf{exp} \rangle \supset \langle \mathbf{exp} \rangle$ even though only evalBeta can handle parameters. As discussed in Section 3.2, the current type of a function does not indicate which parameters it is intended to handle. Consider the expressions "$\nu\boldsymbol{x}.$ eval $\boldsymbol{x}$" and "$\nu\boldsymbol{x}.$ evalBeta $\boldsymbol{x}$". Both expressions are well-typed, but the first

will get stuck as eval is not intended to handle parameters. Therefore, in order to determine if a list of cases is exhaustive, we first need to distinguish functions based on the types of parameters they are expected to handle, which we call its *world*.

Definition 5.1.2 (World)**.** *A function's* world *indicates the types of parameters it is intended to handle and is defined as follows:*

$$\textit{Worlds} \quad \mathcal{W} \quad ::= \quad \cdot \mid \mathcal{W}, (\Omega, \boldsymbol{A}) \mid *$$

It is best to view a function's world as a schema describing the extensions to the signature which it is intended to handle. The world $*$ stands for the *any* world permitting any parameters, while the world $\cdot$ is the empty world indicating that no parameters (extensions to the signature) are permitted. For example, functions in ML and Haskell all make sense with respect to the empty world. Finally, one can build up a *set* of types, $(\Omega, \boldsymbol{A})$, where Ω contains the free variables in $\boldsymbol{A}$. For example, the evalBeta function needs to be declared in a world containing $(\cdot, \mathbf{exp})$. If a function is intended to handle parameters of arbitrary combinators, then the world must contain $((\cdot, \boldsymbol{A} \in \mathbf{o}),\ \mathbf{comb}\ \boldsymbol{A})$.

It is important to emphasize that a world simply provides a set of types describing the parameters a function is intended to handle. The order does not matter. Throughout this chapter, all of our examples utilize the dependent-typed signature summarized in Section 2.4.2. For simplicity, our examples will only use the following two worlds:

$$
\begin{array}{rcl}
\textsc{closed} & = & \cdot \\
\textsc{open} & = & \cdot,\ (\cdot,\ \mathbf{exp}), \\
 & & ((\cdot, \boldsymbol{A}{\in}\mathbf{o}),\ \mathbf{comb}\ \boldsymbol{A}), \\
 & & ((\cdot, \boldsymbol{A}{\in}\mathbf{o}),\ \mathbf{nd}\ \boldsymbol{A}), \\
 & & ((\cdot, \boldsymbol{P}{\in}\mathbf{nat}, \boldsymbol{E}{\in}\mathbf{exp}^{\#}),\ \mathbf{equiv\ (s}\ \boldsymbol{P}\mathbf{)}\ \boldsymbol{E}\ \mathbf{(var'\ one)})
\end{array}
$$

The world CLOSED indicates that no parameters are allowed and eval respects this world. A function that respects the world OPEN supports parameters of type **exp**, **comb** $\boldsymbol{A}$, **nd** $\boldsymbol{A}$, and **equiv (s** $\boldsymbol{P}$**)** $\boldsymbol{E}$ **(var' one)**. We declare $\boldsymbol{E}$ to be a parameter in **equiv (s** $\boldsymbol{P}$**)** $\boldsymbol{E}$ **(var' one)**, but for the examples in this chapter it would have also sufficed to allow it to be any arbitrary **exp**. We will see an advanced example (proving our translation functions are deterministic) in Section 6.8 where this distinction is necessary.

The function evalBeta will be declared to have world OPEN. The **lam** case of evalBeta makes a recursive call in scope of a new parameter, which means that the function does not respect the world CLOSED. Note that we could also declare evalBeta to have the world "$(\cdot,\ \mathbf{exp}), \cdot$", but we instead make a stronger claim by declaring it to respect OPEN. All of our examples will be declared to have worlds CLOSED or OPEN, but the reader should keep in mind that the same function may respect multiple worlds.

As eval has world CLOSED, our coverage checker will verify that a case exists for **lam** and **app**. Similarly, as evalBeta has world OPEN, it will check that a case exists for **lam**, **app**, and compatible parameters, i.e. of type $\mathbf{exp}^{\#}$. In Section 5.2 we will modify the type system to distinguish functions by their respective worlds. Our modifications will restrict weakening to take a function's world into account, which will statically guarantee that the eval function never encounters a parameter,

and hence will never get stuck.

Twelf worlds: For the Twelf (Pfenning and Schürmann 1998) enthusiast, we will briefly compare Delphin worlds to Twelf worlds. Twelf worlds provide a set of *blocks* that describe the parameters that a function[1] respects while Delphin worlds provide a set of $(\Omega, \boldsymbol{A})$. One can think of $(\Omega, \boldsymbol{A})$ as a one-element block where Ω is analogous to Twelf's `some` variables. In Twelf, blocks are often greater than one element as they are used to describe the shape of the context. In Example 4.5.3 we wrote nd2hil using a parameter function to save a map between parameters. Since Twelf is not higher-order, this technique is not possible and one may instead declare a function to make sense in a world with a multi-element block that states that every element in the context has its mapping directly to the right of it. However, in Example 4.6.1 we wrote toDebruijn which did more than simply maintain a mapping as it also changed mappings, which cannot be done with Twelf blocks. The Delphin technique of passing around a parameter function offers much more flexibility in what one may express as well as alleviate concerns with respect to the order of elements in a block.

We will next discuss the well-formedness of worlds, world inclusion, and world subsumption.

5.1.2 World Judgments

Figure 5.1 gives the well-formedness rules for worlds. The only interesting rule is worldAdd which checks that all types are well-formed in their provided contexts. As in our development of the core (Chapter 4), we make an implicit appeal that the context Ω is well-formed, which is made explicit in Appendix A.2. Additionally, we further restrict Ω to only contain declarations of the form $\boldsymbol{x}{:}\boldsymbol{A}$ and $\boldsymbol{x}{:}\boldsymbol{A}^{\#}$ as

[1] More specifically, it is a relation.

$$\frac{}{\cdot\ \mathsf{world}}\ \mathsf{worldEmpty} \qquad \frac{}{*\ \mathsf{world}}\ \mathsf{worldAny} \qquad \frac{\mathcal{W}\ \mathsf{world} \quad \Omega \vdash \boldsymbol{A}\ \mathsf{wff}}{\mathcal{W}, (\Omega, \boldsymbol{A})\ \mathsf{world}}\ \mathsf{worldAdd}$$

Figure 5.1: Well-Formed Worlds

$$\frac{\Omega \vdash (\mathrm{id}_\Omega, \omega) : \Omega, \Omega'}{(\Omega, \boldsymbol{A}[\mathrm{id}_\Omega, \omega]) \in (\mathcal{W}, (\Omega', \boldsymbol{A}))} \qquad \frac{(\Omega, \boldsymbol{A}) \in \mathcal{W}}{(\Omega, \boldsymbol{A}) \in (\mathcal{W}, (\Omega_2, \boldsymbol{A_2}))} \qquad \frac{}{(\Omega, \boldsymbol{A}) \in *}$$

Figure 5.2: World Inclusion

variables of type τ cannot possibly occur as free variables in types.

The judgment $(\Omega, \boldsymbol{A}) \in \mathcal{W}$ states that a parameter of type $\boldsymbol{A}$, which is well-formed in Ω, is supported by world $\mathcal{W}$. This world inclusion judgment may be seen as analogous to set inclusion and is given in Figure 5.2. Recall that if a world contains $(\Omega', \boldsymbol{A})$, then Ω' is a quantification of all the free variables in $\boldsymbol{A}$. Therefore, the first two rules state that if $(\Omega', \boldsymbol{A})$ is in a world, then the world supports parameters of type $\boldsymbol{A}$ modulo any (well-typed) instantiations of the free variables. Notice that we have introduced a notation of concatenation of contexts and substitutions. The concatenation of contexts, Ω, Ω', is straightforward. The concatenation of substitutions in (ω_0, ω) only makes sense if ω is of the form $\cdot, e_1/\alpha_1, \ldots, e_n/\alpha_n$ where the resulting substitution is analogous to extending a substitution by a list as in Section 4.1. The final rule states that the world $*$ supports parameters of any type.

We say $\mathcal{W}_1$ is subsumed by $\mathcal{W}_2$, or $\mathcal{W}_1 \leq \mathcal{W}_2$, if all parameters supported by $\mathcal{W}_1$ are also supported by $\mathcal{W}_2$. If a function respects $\mathcal{W}_2$, then it may be called in scope of any arbitrary collection of parameters that is supported by $\mathcal{W}_2$. Therefore, the same function may also be called in any arbitrary collection of parameters that is supported by $\mathcal{W}_1$. We formally present world subsumption in Figure 5.3. The interesting rule is the last one which formalizes that $\mathcal{W}_1 \leq \mathcal{W}_2$ if every type supported by $\mathcal{W}_1$ is also supported by $\mathcal{W}_2$ by appealing to world inclusion.

$$\frac{}{\cdot \leq \mathcal{W}} \qquad \frac{}{\mathcal{W} \leq *} \qquad \frac{\mathcal{W}_1 \leq \mathcal{W}_2 \quad (\Omega, \boldsymbol{A}) \in \mathcal{W}_2}{\mathcal{W}_1, (\Omega, \boldsymbol{A}) \leq \mathcal{W}_2}$$

Figure 5.3: World Subsumption

World subsumption is crucial for allowing functions to call other functions that are declared to have different worlds. If a function f has world $\mathcal{W}_1$ and a function g has world $\mathcal{W}_2$, then f may only call g if $\mathcal{W}_1 \leq \mathcal{W}_2$.

5.2 Semantics

In this section we discuss the formal extensions to the Delphin type system from Chapter 4 to distinguish functions by their respective worlds. Under these extensions programs are well-typed only if they are also covered and hence guaranteed not to get stuck. In our implementation (Chapter 6), programs will be first typed with respect to the typing rules in Chapter 4 and then they will be annotated with user specified world information and typed with respect to the typing rules in this section. If a program is typeable in these latter semantics, then we are guaranteed that programs will never get stuck.

We will only show the changes necessary to build coverage into the system, but the entire system is summarized in Appendix A.2. We will motivate parts of this section with our eval and evalBeta functions from Example 5.1.1.

5.2.1 Computation-Level Types τ

We start by adding a type that allows us to distinguish functions by their respective worlds. Figure 5.4 defines the extensions to the grammar from Figure 4.1 to support worlds.

If an expression has type $\nabla\mathcal{W}.\ \tau$ in Ω, then we are stating that the expression has

Computational Types	τ, σ	::=	$\ldots \mid \nabla\mathcal{W}.\ \tau$
Expressions	e, f	::=	$\ldots \mid \nu u{\in}\mathcal{W}.\ e \mid e\backslash u$
Context	Ω	::=	$\ldots \mid \Omega, u\overset{\nabla}{\in}\mathcal{W}$

Figure 5.4: Syntactic Extensions to Delphin Supporting Worlds

type τ in any extension of Ω which is supported by world $\mathcal{W}$. One should note the similarities between the types $\nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$ and $\nabla\mathcal{W}.\ \tau$. The former states that we have an expression of type τ in a context extended by exactly one parameter, while the latter states that we have an expression of type τ in a context extended by an arbitrary number of parameters supported by world $\mathcal{W}$. Therefore, this latter type can be thought of as a generalization of the former to support an arbitrary number of parameters.

The introduction and elimination forms for $\nabla\mathcal{W}.\ \tau$ are similar to those of the other ∇ type. The introduction form is $\nu u{\in}\mathcal{W}.\ e$, and the type is eliminated via $e\backslash u$. Recall that the evaluation of "$\nu\boldsymbol{x}.$ eval $\boldsymbol{x}$" would get stuck as eval has world CLOSED and does not support the parameter $\boldsymbol{x}$. Previously, this expression was well-typed because we allowed the weakening of eval over the new $\boldsymbol{x}$. To rectify this, weakening (Lemma 4.4.1) over parameters will no longer hold and one must handle weakening explicitly via the introduction and elimination forms of this new type. Therefore, eval will not be accessible in scope of any parameters. More specifically, functions can only be accessed in scope of parameters that are supported by their respective worlds.

The type $\nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$ can be used to work with a specific parameter, while our type $\nabla\mathcal{W}.\ \tau$ will instead be used to control when functions can be accessed. Therefore, the differences between eval and evalBeta will now be captured by their respective types. We will change the type of eval to be ∇CLOSED. ($\langle\mathbf{exp}\rangle \supset \langle\mathbf{exp}\rangle$) and the type of evalBeta to be ∇OPEN. ($\langle\mathbf{exp}\rangle \supset \langle\mathbf{exp}\rangle$).

One important extension is with respect to contexts:

$$\text{Context } \Omega ::= \cdot \mid \Omega, \alpha{\in}\delta \mid \Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega, u \overset{\triangledown}{\in} \mathcal{W}$$

Previously, evaluation of Delphin programs occurred in a context empty with respect to pattern-variable declarations $\alpha{\in}\delta$ and open with respect to signature extensions $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$. Now, the evaluation of programs is also open with respect to weakening declarations $u \overset{\triangledown}{\in} \mathcal{W}$. Weakening declarations are syntactic markers indicating where, and by what parameters, contexts may be extended; they will be explained further in Section 5.2.4.

In Section 5.2.2 we will discuss the extensions to our well-formedness judgments from Chapter 4. The extensions for substitutions are discussed in Section 5.2.3 while the changes to our type system and operational semantics are given in Sections 5.2.4 and 5.2.5, respectively. The details of checking if a list of cases is exhaustive will be subsequently discussed in Section 5.3.

5.2.2 Valid Types and Contexts

The extensions of our well-formedness judgments from Figures 4.2 and 4.3 are presented in Figure 5.5. These extensions are straightforward and simply appeal to our well-formed worlds judgment from Figure 5.1. The type $\nabla\mathcal{W}.\ \tau$ is well-formed if τ is well-formed and $\mathcal{W}$ is a valid world. Similarly, the context $(\Omega, u \overset{\triangledown}{\in} \mathcal{W})$ is well-formed if Ω is well-formed and the weakening declaration $u \overset{\triangledown}{\in} \mathcal{W}$ is over a well-formed world $\mathcal{W}$.

$$\frac{\mathcal{W}\ \mathsf{world} \quad \Omega \vdash \tau\ \mathsf{wff}}{\Omega \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}}\ \nabla_{\text{world}}\mathsf{wff} \qquad \frac{\Omega\ \mathsf{ctx} \quad \mathcal{W}\ \mathsf{world}}{\Omega, u{:}\overset{\triangledown}{\in}\mathcal{W}\ \mathsf{ctx}}\ \mathsf{ctxAddWorld}$$

Figure 5.5: Well-Formedness of Types and Contexts for Worlds

$$\frac{\Omega' \vdash \omega : \Omega \quad \mathcal{W}\ \mathsf{world}}{(\Omega', u'{:}\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'}\omega, u'/u) : (\Omega, u{:}\overset{\triangledown}{\in}\mathcal{W})}$$

Figure 5.6: Substitution Typing Extension for Worlds

5.2.3 Substitutions and Substitution Application

The typing rules for substitutions are extended with one rule to handle weakening declarations $u{:}\overset{\triangledown}{\in}\mathcal{W}$, which is presented in Figure 5.6. The rest of the rules from Figure 4.4 remain unchanged. We handle weakening declarations in exactly the same manner as we handle signature-extension declarations $\boldsymbol{x}{:}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$. Declarations of the form $u{:}\overset{\triangledown}{\in}\mathcal{W}$ can only be renamed to $u'{:}\overset{\triangledown}{\in}\mathcal{W}$. There is no need to ever refine $\mathcal{W}$ with respect to a substitution as the well-formedness of worlds does not depend on the context. Therefore, substitutions allow one to instantiate pattern-variable declarations $\alpha{:}\in\delta$, but they preserve the structure of the rest of the context.

The definition of identity substitutions (Definition 3.4.5) remained unchanged from SimpleDelphin (Chapter 3) to full Delphin (Chapter 4). However, we now extend the definition to handle weakening declarations. The full definition of identity substitutions (for Delphin and LF) is given in Definition 5.2.1, but the only extension is defining $\mathrm{id}_{\Omega, u{:}\overset{\triangledown}{\in}\mathcal{W}}$ as $\uparrow_u \mathrm{id}_\Omega, u/u$.

Definition 5.2.1 (Identity Substitutions)**.**

$$\begin{array}{lcl|lcl} \mathrm{id}_{\cdot} & = & \cdot & \mathbf{id}_{\cdot} & = & \cdot \\ \mathrm{id}_{\Omega,\alpha\in\delta} & = & \uparrow_\alpha \mathrm{id}_\Omega, \alpha/\alpha & \mathbf{id}_{\boldsymbol{\Gamma},\boldsymbol{x}:\boldsymbol{A}} & = & \uparrow_{\boldsymbol{x}} \mathbf{id}_{\boldsymbol{\Gamma}}, \boldsymbol{x}/\boldsymbol{x} \\ \mathrm{id}_{\Omega,\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}} & = & \uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x} & & & \\ \mathrm{id}_{\Omega,u\overset{\triangledown}{\in}\mathcal{W}} & = & \uparrow_u \mathrm{id}_\Omega, u/u & & & \end{array}$$

Substitution composition $\omega_1 \circ \omega_2$, and $\gamma_1 \circ \gamma_2$ does not change (Def. 3.4.6) from the

Types

$$(\nabla\mathcal{W}.\ \tau)[\omega] \quad = \quad \nabla\mathcal{W}.\ (\tau[\omega])$$

Expressions

$$(\nu u{\in}\mathcal{W}.\ e)[\omega] \quad = \quad \nu u{\in}\mathcal{W}.\ e[\uparrow_u \omega, u/u]$$
$$(e\backslash u)[\omega] \quad = \quad e[\omega]\backslash u[\omega]$$

Figure 5.7: Delphin Substitution Application Extension for Worlds

simply-typed or dependently-typed version. Recall that substitution composition is reversed from the standard function composition. If $\Omega_1 \vdash \omega_1 : \Omega_0$ and $\Omega_2 \vdash \omega_2 : \Omega_1$, then $\Omega_2 \vdash \omega_1 \circ \omega_2 : \Omega_0$.

Finally, Figure 5.7 defines substitution application on our new types and expressions. Substitution Application on all other types and expressions remain unchanged from Figures 4.7 and 4.8.

5.2.4 Type System

The Delphin type system from Figure 4.5 is both extended to handle $\nabla\mathcal{W}.\ \tau$ and modified to restrict the arbitrary weakening of expressions of type τ. Figure 5.8 summarizes the changes in the type system to ensure that all functions are covered in their respective worlds and that they are only accessed in scope of parameters compatible with said world.

Function introduction (impl) is now restricted to only allow for an exhaustive list of cases, guaranteed by the premise $\Omega \vdash \overline{c}$ covers τ, whose details are deferred to Section 5.3. Intuitively, this judgment checks that $\overline{c}$ provides a pattern for every well-typed expression of type τ in Ω.

As motivated via our eval and evalBeta example, the arbitrary weakening of expressions over parameters (Lemma 4.4.1) is now disabled. In Chapter 4, the admissibility of weakening was built-in by not restricting the Ω_2 found in τvar, pop, and

$$\frac{\text{for all }{}_{c_i \in \overline{c}}(\Omega \vdash c_i \in \tau) \quad \Omega \vdash \overline{c} \text{ covers } \tau}{\Omega \vdash \text{fn } \overline{c} \in \tau}\ \mathsf{impl}$$

$$\frac{\Omega_2 \text{ closed}}{\Omega, u{\in}\tau, \Omega_2 \vdash u \in \tau}\ \tau\mathsf{var} \qquad \frac{\Omega \vdash e \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \quad \Omega_2 \text{ closed}}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$$

$$\frac{\Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash e \in \tau}{\Omega \vdash \nu u{\in}\mathcal{W}.\ e \in \nabla\mathcal{W}.\ \tau}\ \mathsf{newW}$$

$$\frac{\Omega \vdash e \in \nabla\mathcal{W}_2.\ \tau \quad \mathcal{W} \leq \mathcal{W}_2 \quad \Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash \mathcal{W}_2 \text{ supports } \Omega_2}{\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \in \tau}\ \mathsf{popW}$$

. . . . . .

$$\frac{\Omega \vdash c \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \quad \Omega_2 \text{ closed}}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$$

Figure 5.8: Delphin Typing Rules Update for Worlds

cPop. We now refine these rules to restrict Ω_2 such that it only contains pattern-variable declarations $\alpha{\in}\delta$ via the premise "Ω_2 closed". In Appendix A.2.5, we define the judgment $\Omega_A \leq \Omega_B$ to determine if Ω_B is a "closed" extension of Ω_A and hence we equivalently check if $(\Omega, u{\in}\tau) \leq (\Omega, u{\in}\tau, \Omega_2)$.

Aside from our motivating example, one can see that the weakening over parameters cannot be admissible due to the coverage condition added to impl, which makes sure that only covered functions are well-typed. In order for the weakening over parameters to be admissible, every function covered in Ω must also be covered in $\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ for any $\boldsymbol{A}$. In order for this to hold, it would necessitate that all functions respect the world $*$, which is too restrictive to do anything interesting.

As weakening over parameters is disallowed, when one introduces a new parameter (extends the signature), no computation-level functions are accessible. However, note that the isLF rule does not change as extensions to the signature can never invalidate

data.

We now turn to the introduction of $\nabla\mathcal{W}.\ \tau$ given in newW. Our weakenability type is introduced via $\nu u{\in}\mathcal{W}.\ e$, which types e in the context extended with the weakening declaration $u\overset{\nabla}{\in}\mathcal{W}$. Weakening declarations allow us to syntactically annotate the context indicating where and how it may be extended such that the typing (and hence coverage) of expressions is preserved. To provide intuition, the meaning of $\Omega, u\overset{\nabla}{\in}\mathcal{W} \vdash e \in \tau$ can be seen as stating that for all Ω_2 supported by $\mathcal{W}$, $\Omega, \Omega_2 \vdash e \in \tau$. This marker in the context will be used in the coverage judgment to determine which parameters a function must take into account. Additionally, we will see in Lemma 5.5.11 (Section 5.5) that our usage of weakening declarations does indeed entail that the weakening of functions is only permitted when such weakenings respect their worlds.

The introduction form of $\nabla\mathcal{W}.\ \tau$ declares that an expression (usually a function) is weakenable by contexts whose parameters are supported by $\mathcal{W}$. The actual weakening occurs in the elimination $e\backslash u$, from the rule popW. Reading this rule top-down, one can see that if we have a function in Ω, which is weakenable by $\mathcal{W}$, then we can access the function in any extension to Ω that is supported by $\mathcal{W}$. The judgment "$\Omega_1 \vdash \mathcal{W}$ supports Ω_2" checks (1) for all $\boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}$ in Ω_2, $((\Omega_1, \Omega_2),\ \boldsymbol{A}) \in \mathcal{W}$ and (2) for all $u\overset{\nabla}{\in}\mathcal{W}'$, $\mathcal{W}' \leq \mathcal{W}$. In Appendix A.2.5, we define a more general judgment $\Omega_A \leq_{\mathcal{W}} \Omega_B$ to determine if Ω_B is an extension of Ω_A supported by $\mathcal{W}$ and therefore we can alternatively check $(\Omega, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\nabla}{\in}\mathcal{W}, \Omega_2)$.

The elimination rule popW is very similar to our elimination rule pop for the other ∇ type. In Section 5.1.2 we discussed that a function can call another function if the caller's world is subsumed by the callee's. In the expression $e\backslash u$ the callee is e and this support of world subsumption is built in by allowing e to have type $\nabla\mathcal{W}_2.\ \tau$ and u to have world $\mathcal{W}$, as long as $\mathcal{W} \leq \mathcal{W}_2$ (Figure 5.3).

The uses of this new type are best illustrated with an example. With our new restrictions, the evalBeta function from Example 5.1.1 is not typeable as the recursive call in the **lam** case is not well-typed as it would require accessing (weakening) evalBeta in scope of a new parameter **x**. The new version of our function is as follows:

Example 5.2.2 (Open Evaluator with Worlds)**.** *This example updates our* evalBeta *function from Example 5.1.1 such that it is typeable in our updated system with worlds.*

$$
\begin{array}{l}
\mu\mathsf{evalBeta} \in \nabla\textsc{open}.\ (\langle\mathbf{exp}\rangle \supset \langle\mathbf{exp}\rangle). \\
\quad \nu u{\in}\textsc{open}. \\
\qquad \begin{array}{lll}
\text{fn}\ \langle\mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2}\rangle & \mapsto \text{case}\ ((\mathsf{evalBeta}\backslash u)\ \langle\boldsymbol{E_1}\rangle,\ (\mathsf{evalBeta}\backslash u)\ \langle\boldsymbol{E_2}\rangle) & \\
\qquad\qquad & \quad \text{of}\ (\langle\mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{F}\ \boldsymbol{x})\rangle,\ \langle\boldsymbol{V}\rangle) & \\
\qquad\qquad & \qquad\qquad \mapsto (\mathsf{evalBeta}\backslash u)\ \langle\boldsymbol{F}\ \boldsymbol{V}\rangle & \\
\qquad\qquad & \quad \mid (\langle\boldsymbol{F}\rangle,\ \langle\boldsymbol{V}\rangle) \qquad\quad \mapsto \langle\mathbf{app}\ \boldsymbol{F}\ \boldsymbol{V}\rangle & \\
\mid\ \langle\mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x})\rangle & \mapsto \text{case}\ (\nu\boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ (\mathsf{evalBeta}\backslash u)\ \langle\boldsymbol{E}\ \boldsymbol{x}\rangle) & \\
\qquad\qquad & \quad \text{of}\ \ (\nu\boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \langle\boldsymbol{E'}\ \boldsymbol{x}\rangle)\ \mapsto \langle\mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E'}\ \boldsymbol{x})\rangle & \\
\mid\ \langle\boldsymbol{x}^{\#}\rangle & \mapsto \langle\boldsymbol{x}\rangle &
\end{array}
\end{array}
$$

Example 5.2.2 updates the older evalBeta function by declaring it weakenable by parameters supported by OPEN. This update only required adding $\nu u{\in}\textsc{open}$ to introduce the ∇ type and then replace every call to evalBeta with $(\mathsf{evalBeta}\backslash u)$. The **lam** case is the most interesting as $(\mathsf{evalBeta}\backslash u)$ verifies that the new $\boldsymbol{x}$ is supported by world OPEN.

Implementation Note 1: The implementation (Chapter 6) hides the details of the type $\nabla\mathcal{W}.\ \tau$ by allowing the user to declare the world that a function is intended to respect and automatically make the same update that we have shown above. More specifically, given a function $\mu\mathsf{foo}{\in}\tau.\ e$ and a declaration that foo respects $\mathcal{W}$, the implementation will automatically covert this to $\mu\mathsf{foo}{\in}(\nabla\mathcal{W}.\ \tau).\ \nu u\overset{\nabla}{\in}\mathcal{W}.\ e$ where e

is updated such that every access to a g of type $\nabla\mathcal{W}_2.\ \tau$ is replaced with $g\backslash u$. In the case of recursive calls, g will be foo, but it may be other functions as well.

We next turn to the eval function which respects the world CLOSED. One important observation is that the eval function from Example 5.1.1 is still well-typed as it never made a recursive call in scope of new parameters. Therefore, the type $\langle\mathbf{exp}\rangle \supset \langle\mathbf{exp}\rangle$ indicates a function on λ-expressions, which cannot be weakened by any parameters. Similarly, a function of type ∇CLOSED. $(\langle\mathbf{exp}\rangle \supset \langle\mathbf{exp}\rangle)$ also indicates a function on λ-expressions that cannot be weakened by any parameters. In other words, a type τ alone does not permit any weakening over parameters, while the type ∇CLOSED. τ permits weakening over parameters supported by CLOSED, which happen to be none.

Example 5.2.3 (Closed Evaluator with Worlds)**.** *This example updates our* eval *function from Example 5.1.1 with a declaration that it has world* CLOSED.

μeval $\in \nabla$CLOSED. $(\langle\mathbf{exp}\rangle \supset \langle\mathbf{exp}\rangle)$.
  $\nu u{\in}$CLOSED.
    fn $\langle\mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2}\rangle \mapsto$ case $((\mathsf{eval}\backslash u)\ \langle\boldsymbol{E_1}\rangle,\ (\mathsf{eval}\backslash u)\ \langle\boldsymbol{E_2}\rangle)$
      of $(\langle\mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{F}\ \boldsymbol{x})\rangle,\ \langle\boldsymbol{V}\rangle) \mapsto (\mathsf{eval}\backslash u)\ \langle\boldsymbol{F}\ \boldsymbol{V}\rangle$
    | $\langle\mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x})\rangle \mapsto \langle\mathbf{lam}\ (\boldsymbol{\lambda x.}\ \boldsymbol{E}\ \boldsymbol{x})\rangle$

Example 5.2.3 defines our closed evaluator eval with world CLOSED. The update from our older version proceeded identically to the update of evalBeta. As CLOSED $\leq$ OPEN and world subsumption is built into popW, our new eval function would still be well-typed if we replace recursive calls to $(\mathsf{eval}\backslash u)$ with $(\mathsf{evalBeta}\backslash u)$ although that would not have the desired operational behavior.

$$\frac{\Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash e \to f}{\Omega \vdash \nu u{\in}\mathcal{W}.\ e \to \nu u{\in}\mathcal{W}.\ f} \qquad \frac{\Omega \vdash e \to f}{\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \to f\backslash u}$$

$$\frac{}{\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash (\nu u'{\in}\mathcal{W}_2.\ e)\backslash u \to e[\uparrow_u \mathrm{id}_\Omega, u/u']}$$

Figure 5.9: Delphin Operational Semantics Extension for Worlds)

5.2.5 Operational Semantics

We end this section with the formal operational semantics and a sample execution. Figure 5.9 gives the extensions of the operational semantics in Figure 4.6 to handle the introduction and elimination forms of $\nabla\mathcal{W}.\ \tau$. The operational behavior of all other expressions do not change. These three new rules work in exactly the same manner as our other ∇ type. The first rule evaluates the e in $\nu u{\in}\mathcal{W}.\ e$ under the weakening declaration u. The second rule allows for the evaluation of e in $e\backslash u$ by shifting evaluation to the shrunk context where e is well-typed. The expression $e\backslash u$ will eventually evaluate to $(\nu u'{\in}\mathcal{W}_2.\ e')\backslash u$ which behaves as application replacing u for u' in the body e'. Note that if the expression is well-typed, then we are guaranteed that $\mathcal{W} \leq \mathcal{W}_2$ by rule popW.

Example 3.3.6 showed a sample evaluation of our older version of evalBeta. We now show the same sample execution with worlds. We can access the function by doing (evalBeta$\backslash w$) for some w. Our execution is intended to occur at the top level where no parameters exist.

Example 5.2.4 (Sample evalBeta Execution with Worlds)**.**

$\nu w{\in}\textsc{closed}.\ (\mathsf{evalBeta}\backslash w)\ \langle \mathbf{lam\ (\lambda}\boldsymbol{y}\mathbf{:exp.\ }\boldsymbol{y}\mathbf{)}\rangle$
$\ldots \rightarrow \nu w{\in}\textsc{closed}.\ ((\nu u{\in}\textsc{open}.\ \text{fn}\ \ldots)\backslash w)\ \langle \mathbf{lam\ (\lambda}\boldsymbol{y}\mathbf{:exp.\ }\boldsymbol{y}\mathbf{)}\rangle$
$\ldots \rightarrow \nu w{\in}\textsc{closed}.\ \text{case}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ (\mathsf{evalBeta}\backslash w)\ \langle \boldsymbol{x}\rangle)$
$\qquad\qquad\qquad\qquad \text{of}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \langle \boldsymbol{E'}\ \boldsymbol{x}\rangle)\ \mapsto \langle \mathbf{lam\ (\lambda}\boldsymbol{x}\mathbf{.\ }\boldsymbol{E'}\ \boldsymbol{x}\mathbf{)}\rangle$
$\ldots \rightarrow \nu w{\in}\textsc{closed}.\ \text{case}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \langle \boldsymbol{x}\rangle)$
$\qquad\qquad\qquad\qquad \text{of}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \langle \boldsymbol{E'}\ \boldsymbol{x}\rangle)\ \mapsto \langle \mathbf{lam\ (\lambda}\boldsymbol{x}\mathbf{.\ }\boldsymbol{E'}\ \boldsymbol{x}\mathbf{)}\rangle$
$\ldots \rightarrow \nu w{\in}\textsc{closed}.\ \langle \mathbf{lam\ (\lambda}\boldsymbol{x}\mathbf{:exp.\ }\boldsymbol{x}\mathbf{)}\rangle$

Example 5.2.4 uses evalBeta to evaluate $\langle \mathbf{lam\ (\lambda}\boldsymbol{y}\mathbf{:exp.\ }\boldsymbol{y}\mathbf{)}\rangle$. The execution is done under a weakening declaration $w{\in}\textsc{closed}$. The expression $\mathsf{evalBeta}\backslash w$ causes the concrete w to take the role of the hypothetical $u{\in}\textsc{open}$ in the body of evalBeta. Note that this illustrates an example of world subsumption. The execution behaves just as before where we create a new parameter $\boldsymbol{x}$ and recursively call the function on $\mathbf{(\lambda}\boldsymbol{y}\mathbf{:exp.\ }\boldsymbol{y}\mathbf{)}\ \boldsymbol{x}$, which is equivalent to $\boldsymbol{x}$. The recursive call results in $\boldsymbol{x}$ and thus the case analysis causes the function to result in the identity function.

Implementation Note 2: The implementation (Chapter 6) hides the details of the type $\nabla\mathcal{W}.\ \tau$. Therefore, the user can just write "evalBeta $\langle \mathbf{(\lambda}\boldsymbol{y}\mathbf{:exp.\ }\boldsymbol{y}\mathbf{)}\ \boldsymbol{x}\rangle$" as in Example 3.3.6, and it is automatically converted to the one in Example 5.2.4 by prefacing the expression with $\nu w{\in}\textsc{closed}$ and updating every access to a g of type $\nabla\mathcal{W}.\ \tau$ with $g\backslash w$. It is important to emphasize that since the top level has no parameters, world subsumption allows one to call functions respecting any world, i.e. for all worlds $\mathcal{W}$, $\textsc{closed} \leq \mathcal{W}$.

5.2.6 Advanced Discussion of "with"

We presented the "with" syntax to extend parameter functions in Section 3.7.2. This function extension contains a subterm of the form "let $\langle \boldsymbol{R}\rangle = e$ in $\nu\boldsymbol{\Gamma}.\ \langle \boldsymbol{R}\rangle$". The pattern $\langle \boldsymbol{R}\rangle$ matches the result of an execution of the parameter function. If the

result of the parameter function had type $\langle \boldsymbol{A} \rangle \star \langle \boldsymbol{B} \rangle$, then the pattern would be $(\langle \boldsymbol{R_1} \rangle, \langle \boldsymbol{R_2} \rangle)$.

Observe that we *cannot* replace the pattern with a computation-level pattern variable u and write "let $(u{\in}\tau) = e$ in $\nu\boldsymbol{\Gamma}.\ u$" as this expression is not well-typed. Technically this is due to the typing rule $\tau\mathsf{var}$. If τ is a $\forall$, then this rule expresses our invariant that a function cannot be weakened by the arbitrary parameters in $\boldsymbol{\Gamma}$ as it may not be covered with respect to these new parameters. This is why we stipulated (Section 3.7.2) that the "with" construct creates an appropriate pattern instead of using a computation-level variable.

However, if we restricted ourselves to an eager operational semantics, we could also generalize the rule $\tau\mathsf{var}$ to allow the weakening of computation-level types as long as the type does not contain any functional types.[2] However, this generalization (1) is not necessary (if a type τ does not contain any $\forall$'s, then the programmer can always deconstruct the type before creating the new parameters) and (2) the substitution lemma (Lemma 5.5.16) would break on $\tau\mathsf{var}$ unless we are guaranteed to only be substituting variables, values, or fixpoints; hence the *eager* semantics requirement.

5.3 Coverage Algorithm

The type system guarantees that during an application there will always exist a case whose pattern matches the argument being applied. We have seen in the previous section how our type system enforces that a function declared to respect world $\mathcal{W}$ can only be accessed in scope of parameters supported by $\mathcal{W}$. The final step of actually checking a list of cases is done in impl by appealing to the judgment $\Omega \vdash \overline{c}$ covers τ, which is the focus of this section.

[2]Our implementation has an eager operational semantics and indeed makes this generalization.

The judgment $\Omega \vdash \overline{c}$ covers τ checks if, during execution, $\overline{c}$ can be successfully applied to every possible well-typed argument. Typically, τ will be of the form $\forall\overline{\alpha\in\delta}.\ \sigma$, but recall that the type may also be prefaced with an arbitrary number of $\nabla\boldsymbol{x}$'s as ν may range over cases (an example of this can be found in Lemma 3.4.2 as well as in the "with" syntax defined in Section 3.7.2).

A key observation is that the context Ω may contain weakening declarations $u\overset{\triangledown}{\in}\mathcal{W}$ which indicates that the type system will allow this function to be accessed in extensions to Ω that are supported by $\mathcal{W}$. Additionally, Ω may contain individual parameters $\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ if the function is being defined under ν's. For example, in our new evalBeta function (Examples 5.2.2), the typing of its list of cases is in scope of a weakening declaration $u\overset{\triangledown}{\in}\textsc{open}$ which tells us that it must provide a case for all parameters supported by OPEN.

5.3.1 Preliminaries and Judgments

Not surprisingly, the main complication of our coverage algorithm is with respect to the handling of higher-order data. Recall that bracket abstraction (Example 4.5.2) required case analysis over *functions* of type $\langle\mathbf{comb}\ \boldsymbol{A} \longrightarrow \mathbf{comb}\ \boldsymbol{B}\rangle$.

The idea of case analysis is that it allows one to inspect how data was constructed and thus the issue of coverage corresponds to providing a list of cases that handles all possible ways an object can be constructed. Case analysis over first-order simply-typed data is straightforward as we just need to guarantee there is a case for every constructor in the signature that *results* in an object of the type we are analyzing. The key observation in understanding case analysis over higher-order data is that the actual analysis is done over the *result* type. For example, consider objects of type $\boldsymbol{A_1} \longrightarrow \boldsymbol{A_2} \longrightarrow \mathbf{exp}$. All objects of this type have the form $\boldsymbol{\lambda(x_1{:}A_1).\ \lambda(x_2{:}A_2).\ M}$ where $\boldsymbol{M}$ is an $\mathbf{exp}$ and the variables $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ may occur free in $\boldsymbol{M}$. One should

observe that this is due to the existence of canonical forms (Chapter 2); in other words, LF functions are sufficiently weak that no other possibilities exist. Returning to our example, the possibilities for $\boldsymbol{M}$ depend on:

- *Signature Coverage:* All constants (constructors) in the signature that end in type **exp**. For example, since **app** takes two arguments and returns an **exp**, we we will need a case for $\boldsymbol{\lambda(x_1{:}A_1).\ \lambda(x_2{:}A_2).\ \mathbf{app}\ M_1\ M_2}$ where $\boldsymbol{M_1}$ and $\boldsymbol{M_2}$ are pattern variables.

- *Local Parameter Coverage:* Since variables $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ may occur free in $\boldsymbol{M}$, we perform the same analysis on them as we did for constants in the signature. For example, if $\boldsymbol{x_1}$ has type $\boldsymbol{B \longrightarrow \mathbf{exp}}$, then we will need a case for $\boldsymbol{\lambda(x_1{:}A_1).\ \lambda(x_2{:}A_2).\ x_1\ M}$ where $\boldsymbol{M}$ is a pattern variable of type $\boldsymbol{B}$. For the scope of $\boldsymbol{M}$, the variables $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ behave as constants in the signature, just as our parameters created by ν. Therefore, we refer to these variables as *local parameters*.

- *Global Parameter Coverage:* Besides constants and local parameters, $\boldsymbol{M}$ may contain parameters introduced by the ν construct. In other words, when we perform case analysis over an object of type $\boldsymbol{A_1 \longrightarrow A_2 \longrightarrow \mathbf{exp}}$, we are doing it in a possibly nonempty context Ω. This is the context in which the function is being typed. We must therefore provide a case for every $\boldsymbol{y \overset{\triangledown}{\in} B^{\#}}$ in Ω that can construct an **exp**. Rather than specifying a *unique* case for every such parameter, we instead allow the user to provide a case matching *any* parameter, i.e. if there exists a parameter $\boldsymbol{y \overset{\triangledown}{\in} \mathbf{exp}^{\#}}$ and $\boldsymbol{x \overset{\triangledown}{\in} \mathbf{exp}^{\#}}$ in Ω, then the user can provide one case to handle both of these parameters as $\boldsymbol{\lambda(x_1{:}A_1).\ \lambda(x_2{:}A_2).\ p}$ where $\boldsymbol{p}$ is a pattern variable of type $\mathbf{exp}^{\#}$.

  If Ω contains a weakening declaration $\boldsymbol{u \overset{\triangledown}{\in} \mathcal{W}}$, then our type system will allow

the function to be accessed in extensions to the context containing parameters supported by $\mathcal{W}$. Therefore, in order to guarantee that the function is covered (and hence well-typed) in such extended contexts, we must provide cases for all parameters supported by $\mathcal{W}$. Therefore, Global Parameter Coverage checks that there are cases for all compatible parameters supported by $\mathcal{W}$ via *Schematic Parameter Coverage*.

Global Parameter Coverage is the most important one as it guarantees that there exists a case for every parameter that can be in scope of the function when it is called. This is done via Schematic Parameter Coverage which takes a world $\mathcal{W}$ and makes sure a case exists for every type of parameter supported by $\mathcal{W}$. One should notice the careful interplay between typing and coverage. Our typing rules use weakening declarations $u \overset{\nabla}{\in} \mathcal{W}$ to determine when a function can be accessed and the coverage checker uses the same weakening declarations to determine the type of parameters that a function must handle such that Global Parameter Coverage will hold over all possible weakenings.

Figure 5.10 summarizes the judgments used for coverage. The main coverage judgment (used in the typing rules) is $\Omega \vdash \overline{c}$ covers τ. When performing case analysis over LF types, this judgment will appeal to Signature Coverage, Local Parameter Coverage, and Global Parameter Coverage. As mentioned, Global Parameter Coverage will appeal to Schematic Parameter Coverage. Finally, *Parameter Function Coverage* will be used to extend parameter functions (e.g. the desugared "with" statement).

Abbreviations / Operations

Definition 5.3.1 (Abbreviations / Operations)**.**
In Figure 4.7 we defined $\omega + \overline{\alpha \in \delta}$ *and we similarly define* $\omega + \mathbf{\Gamma}$ *to extend a substitution*

Signature Coverage	$\boldsymbol{\Sigma} \gg \overline{c}$ covers τ
Local Parameter Coverage	$\boldsymbol{\Gamma} \gg \overline{c}$ covers τ
Schematic Parameter Coverage	$\mathcal{W} \gg \overline{c}$ covers τ
Global Parameter Coverage	$\Omega \gg^{\text{world}} \overline{c}$ covers τ
Parameter Function Coverage	$\boldsymbol{\Gamma} \gg^{\nabla} \overline{c}$ covers τ
Complete Coverage	$\Omega \vdash \overline{c}$ covers τ

Figure 5.10: Coverage Judgments

ω with identities on everything in Γ.

$$\begin{aligned} \omega + \cdot &= \omega \\ \omega + (\boldsymbol{x_1{:}A_1}, \boldsymbol{\Gamma}) &= (\uparrow_{\boldsymbol{x_1}} \omega, \boldsymbol{x_1}/\boldsymbol{x_1}) + \boldsymbol{\Gamma} \end{aligned}$$

For any (possibly empty) context $\boldsymbol{\Gamma} = \cdot, \boldsymbol{x_1{:}A_1}, \ldots, \boldsymbol{x_m{:}A_m}$, we define the following:

$$\begin{array}{lcl} \boldsymbol{\Pi\Gamma.\ B} & \equiv & \boldsymbol{\Pi x_1{:}A_1.\ \ldots\ \Pi x_m{:}A_m.\ B}, \text{ where } \boldsymbol{B \neq \Pi x{:}B_2.\ B_3} \\ \boldsymbol{M\ \Gamma} & \equiv & \boldsymbol{M\ x_1\ \ldots\ x_m} \\ \nabla\boldsymbol{\Gamma}.\ \tau & \equiv & \nabla\boldsymbol{x_1}{\in}\boldsymbol{A_1}^{\#}.\ \ldots\ \nabla\boldsymbol{x_m}{\in}\boldsymbol{A_m}^{\#}.\ \tau, \text{ where } \tau \neq \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \sigma_2 \\ e\ \backslash\boldsymbol{\Gamma} & \equiv & e\ \backslash\boldsymbol{x_1}\backslash\ldots\backslash\boldsymbol{x_m} \\ \nu\boldsymbol{\Gamma}.\ c & \equiv & \nu\boldsymbol{x_1}{\in}\boldsymbol{A_1}^{\#}.\ \ldots\ \nu\boldsymbol{x_m}{\in}\boldsymbol{A_m}^{\#}.\ c \\ \Omega, \boldsymbol{\Gamma} & \equiv & \Omega, \boldsymbol{x_1}{\in}\boldsymbol{A_1}, \ldots, \boldsymbol{x_m}{\in}\boldsymbol{A_m} \\ \boldsymbol{\Gamma}[\omega] & \equiv & \boldsymbol{x_1{:}A_1}[\|\omega\|], \ldots, \boldsymbol{x_m{:}A_m}[\|\omega + (\cdot, \boldsymbol{x_1{:}A_1}, \ldots, \boldsymbol{x_{m-1}{:}A_{m-1}})\|] \\ \mathrm{id}_{\boldsymbol{\Gamma}} & \equiv & \cdot, \boldsymbol{x_1}/\boldsymbol{x_1}, \ldots, \boldsymbol{x_m}/\boldsymbol{x_m} \end{array}$$

For any (possibly empty) context $\Omega = \cdot, \alpha_1{\in}\delta_1, \ldots, \alpha_m{\in}\delta_m$, we define the following:

$$\begin{array}{lcl} \forall\Omega.\ \tau & \equiv & \forall(\alpha_1{\in}\delta_1; \ldots; \alpha_m{\in}\delta_m).\ \tau \\ \epsilon\Omega.\ c & \equiv & \epsilon\alpha_1{\in}\delta_1.\ \ldots\ \epsilon\alpha_m{\in}\delta_m.\ c \\ \Omega[\omega]; f & \equiv & \alpha_1[\omega]; \ldots; \alpha_m[\omega]; f \\ \Omega; f & \equiv & \alpha_1; \ldots; \alpha_m; f \\ \uparrow_{\Omega}\omega & \equiv & \uparrow_{\alpha_m} \ldots \uparrow_{\alpha_1}\omega \end{array}$$

Definition 5.3.1 gives some abbreviations/operations to aid in the presentation of the rules. We will first dicuss the abbreviations utilizing an LF context $\boldsymbol{\Gamma}$. All LF types can be written as $\boldsymbol{\Pi\Gamma.\ B}$ where $\boldsymbol{\Gamma}$ may be empty and $\boldsymbol{B}$ is not a functional type. This abbreviation is useful as we discussed that case analysis over an object of type $\boldsymbol{\Pi\Gamma.\ B}$ analyzes how an object of type $\boldsymbol{B}$ is constructed where the variables in $\boldsymbol{\Gamma}$ are treated as extensions to the signature, i.e. local parameters. The expression

$\boldsymbol{M}\ \boldsymbol{\Gamma}$ refers to multiple applications where $\boldsymbol{M}$ is applied to every variable in $\boldsymbol{\Gamma}$. The expressions $\nabla\boldsymbol{\Gamma}.\ \tau$ and $e\backslash\boldsymbol{\Gamma}$ are analogous. The case $\nu\boldsymbol{\Gamma}.\ c$ expresses a case c under possibly many new parameters. We write $\Omega, \boldsymbol{\Gamma}$ to express the extension of Ω by $\boldsymbol{\Gamma}$. We can apply a substitution to a context by writing $\boldsymbol{\Gamma}[\omega]$, which applies ω to the first declaration $\boldsymbol{x_1{:}A_1}$ and then continues with $\boldsymbol{(\uparrow_{x_1} \omega, x_1/x_1)}$ on the rest of the context. This definition of substitution allows us to express $\boldsymbol{(\Pi\Gamma.\ B)}[\omega]$ as $\boldsymbol{\Pi(\Gamma[\omega]).\ B}[\omega + \boldsymbol{\Gamma}]$ where $+$ is defined to extend a substitution with identities for everything in $\boldsymbol{\Gamma}$. Finally, we allow computation-level identity substitutions over LF contexts via $\mathrm{id}_{\boldsymbol{\Gamma}}$, which is straightforward.

We also have abbreviations utilizing computation-level contexts Ω restricted to pattern-variable declarations $\alpha{\in}\delta$. Recall that functions can have the form $\forall\overline{\alpha{\in}\delta}.\ \tau$. We will also express such functions as $\forall\Omega.\ \tau$ for conciseness. Our rules will be explicit with respect to pattern variables and hence we define $\epsilon\Omega.\ c$ to express the introduction of possibly many of them. As functions range over a list of arguments, so do patterns and we write $\Omega[\omega]; f$ to express a pattern ending with f and prefaced with all variables in Ω under the *same* substitution. The expression $\Omega; f$ is similar but the variables do not undergo any substitution. Finally, $\uparrow_\Omega \omega$ adds $\uparrow_\alpha$'s to the substitution ω in order to extend the codomain by Ω.

Unification of LF Types

In Chapter 4 we presented an identity function over $\langle\mathbf{nd}\ \boldsymbol{A}\rangle$ and demonstrated that case analysis over such a type required the refinement of $\boldsymbol{A}$. Although this refinement can be done implicitly (Example 4.1.3), the actual function is given in Example 4.1.2 and has type $\forall(\boldsymbol{A}{\in}\mathbf{o};\ u{\in}\langle\mathbf{nd}\ \boldsymbol{A}\rangle).\ \langle\mathbf{nd}\ \boldsymbol{A}\rangle$. When performing case analysis over simple types, we need to check for the existence of a case that handles every compatible (1) constant in the signature, (2) local parameter (LF $\boldsymbol{\lambda}$-

bound variable), and (3) parameter introduced via ν. Due to dependencies, the compatibility of a constant (or parameter) depends on the result type under possible refinements of indices. For example, consider the identity function of type $\forall(\boldsymbol{A}{\in}\mathbf{o};\ u{\in}\langle\mathbf{nd}\ \boldsymbol{A}\rangle).\ \langle\mathbf{nd}\ \boldsymbol{A}\rangle$. This function type indicates that our case analysis may refine the $\boldsymbol{A}$ and hence this permits the matching of the constant **impi**, which has type $(\mathbf{nd}\ \boldsymbol{B_1} \to \mathbf{nd}\ \boldsymbol{B_2}) \to \mathbf{nd}\ (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})$. In order to see that **impi** was a possible match we need to argue that $\mathbf{nd}\ (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})$ can match $\mathbf{nd}\ \boldsymbol{A}$ via refinements of the type $\boldsymbol{A}$. This is a *unification* problem, which we express as "$\mathbf{nd}\ (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2}) \approx \mathbf{nd}\ \boldsymbol{A}$". The result of a unification problem is a substitution ω that expresses refinements to the free variables such that both sides of the equation are equal. For example, assuming (1) $\Omega_{\text{domain}} = \cdot, \boldsymbol{B_1}{\in}\mathbf{o}, \boldsymbol{B_2}{\in}\mathbf{o}, \boldsymbol{D}{\in}(\mathbf{nd}\ \boldsymbol{B_1} \to \mathbf{nd}\ \boldsymbol{B_2}), \boldsymbol{A}{\in}\mathbf{o}$ and (2) $\Omega_{\text{ex}} = \cdot, \boldsymbol{B_1}{\in}\mathbf{o}, \boldsymbol{B_2}{\in}\mathbf{o}, \boldsymbol{D}{\in}(\mathbf{nd}\ \boldsymbol{B_1} \to \mathbf{nd}\ \boldsymbol{B_2})$ and (3) $\omega_{\text{ex}} = (\mathrm{id}_{\Omega_{\text{ex}}}, (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})/\boldsymbol{A})$, then $\Omega_{\text{ex}} \vdash \omega_{\text{ex}} : \Omega_{\text{domain}}$ and ω_{ex} is a unifier since it makes both sides equal:

$$(\mathbf{nd}\ (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2}))[\mathrm{id}_{\Omega_{\text{ex}}}, (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})/\boldsymbol{A}] \ \equiv_{\boldsymbol{\alpha\beta\eta}}\ \mathbf{nd}\ (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})$$

$$(\mathbf{nd}\ \boldsymbol{A})[\mathrm{id}_{\Omega_{\text{ex}}}(\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})/\boldsymbol{A}] \ \equiv_{\boldsymbol{\alpha\beta\eta}}\ \mathbf{nd}\ (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})$$

Definition 5.3.2 (Unifiers)**.** *We write $\omega \in (\boldsymbol{A} \approx \boldsymbol{B})$ to express that ω is a substitution such that $\boldsymbol{A}[\omega] \equiv_{\boldsymbol{\alpha\beta\eta}} \boldsymbol{B}[\omega]$. We call ω a unifier of $\boldsymbol{A}$ and $\boldsymbol{B}$.*

There can be many unifiers to the same problem. Recall that we have a base formula $\mathbf{p}$ (Example 2.2.2), and hence another possible substitution which unifies these equations is $\omega_{\text{base}} = \cdot, \mathbf{p}/\boldsymbol{B_1}, \mathbf{p}/\boldsymbol{B_2}, (\boldsymbol{\lambda x.\ x})/\boldsymbol{D}, (\mathbf{p} \Rightarrow \mathbf{p})/\boldsymbol{A}$. The codomain of the substitution is different than in ω_{ex}, i.e. $\cdot \vdash \omega_{\text{base}} : \Omega_{\text{domain}}$.

If $\omega \in (\boldsymbol{A} \approx \boldsymbol{B})$ then, by definition, $\boldsymbol{A}[\omega] \equiv_{\boldsymbol{\alpha\beta\eta}} \boldsymbol{B}[\omega]$. Additionally, now that both sides are equal, we can also argue that for all ω_2 that are well-typed, $\boldsymbol{A}[\omega][\omega_2] \equiv_{\boldsymbol{\alpha\beta\eta}} \boldsymbol{B}[\omega][\omega_2]$. For all well-typed ω_2, this last statement is equivalent to

stating $\boldsymbol{A}[\omega \circ \omega_2] \equiv_{\boldsymbol{\alpha\beta\eta}} \boldsymbol{B}[\omega \circ \omega_2]$ (Lemma 5.5.18).

We say that a unifier ω is a *most-general unifier*, if any other unifier ω' can be expressed as $\omega \circ \omega_2$, for some ω_2. In other words, the most-general unifier sets both sides of the equation equal in such a way that *any* other solution can be expressed as applying additional substitutions to both sides of the equation. As an example, ω_{ex} is a most-general unifier and hence there must exist a substitution ω_2 such that $\omega_{\text{base}} = \omega_{\text{ex}} \circ \omega_2$. In this case, $\omega_2 = \cdot, \mathbf{p}/\boldsymbol{B_1}, \mathbf{p}/\boldsymbol{B_2}, (\boldsymbol{\lambda x.\ x})/\boldsymbol{D}$. It is useful to think of ω_2 as expressing refinements on the codomain of ω_{ex}. The codomain of ω_{ex} will correspond to pattern variables and the existence of ω_2 guarantees that during runtime, we will always be able to find an assignment of pattern variables for any use of the constant **impi**.

Definition 5.3.3 (Most-General Unifiers)**.** *We write $\omega = mgu\ (\boldsymbol{A} \approx \boldsymbol{B})$ if ω is a most-general unifier which means that for all other unifiers ω', there must exist an ω_2 such that $\omega' = \omega \circ \omega_2$.*

We will use most-general unifiers to construct a list of cases we need to match. The codomain of the unifier determines the pattern (ϵ-quantified) variables. For example, we can use the unifier ω_{ex} to state that a function of type $\forall(\boldsymbol{A}{\in}\mathbf{o};\ u{\in}\langle\mathbf{nd}\ \boldsymbol{A}\rangle).\ \tau$ must contain a case of the form "$\epsilon\boldsymbol{B_1}.\ \epsilon\boldsymbol{B_2}.\ \epsilon\boldsymbol{D}.\ ((\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2}); \langle\mathbf{impi}\ \boldsymbol{D}\rangle) \mapsto f$" for any f. Additionally, we can use our abbreviations and also write this same case as "$\epsilon\Omega_{\text{ex}}.\ \ (\boldsymbol{A}[\omega_{\text{ex}}]; \langle\mathbf{impi}\ (\boldsymbol{D}[\omega_{\mathbf{ex}}])\rangle) \mapsto f$". Since we used a most-general unifier, we are guaranteed during execution that any object constructed with **impi** will match this case via some refinements of the pattern variables.

Our coverage algorithm is designed to work with respect to any unification algorithm and our rules abstract away from the details of any such algorithm. In the implementation we follow the design by Dowek et al. (1998) which defines how to do

unification over LF objects. Note that we will only employ the unification algorithm to unify LF types $\boldsymbol{A}$ and the context will not contain any computation-level types τ. For computation-level types τ, our coverage algorithm generates a pattern without any use of unification.

We have motivated our use of most-general unifiers, and now turn to discussing the actual coverage judgments summarized in Figure 5.10. All judgments take a type τ indicating the type of the function we are constructing. Therefore, τ is always of the form $\nabla\boldsymbol{\Gamma}.\ \forall\Omega.\ \sigma$. A function that performs case analysis over an LF type has the form $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{(\Pi\Gamma_x.\ B_x)}).\ \tau$ and we construct a list of cases by looking at how objects of type $\boldsymbol{B_x}$ can be constructed, possibly with refinements of the variables in Ω_A. Therefore, we *split* our list of cases based on $\boldsymbol{B_x}$. The type of the function uniquely determines exactly what we are splitting. Recall from Chapter 4 that Ω_A is needed to express case analysis over dependent types, but it is typically treated implicitly via a frontend convenience.

5.3.2 Rules for Signature Coverage

The judgment $\boldsymbol{\Sigma} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{(\Pi\Gamma_x.\ B_x)}).\ \tau$ traverses through the signature $\boldsymbol{\Sigma}$ checking that $\overline{c}$ contains a case for every constant that can be used to construct an object of type $\boldsymbol{B_x}$ (with possible refinements to Ω_A). Note that we assume that all free variables in $\boldsymbol{(\Pi\Gamma_x.\ B_x)}$ are quantified in Ω_A, which is guaranteed in the main judgment before this judgment is used. We define Signature Coverage as follows:

$$\frac{}{\cdot \gg \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{scEmpty}$$

$$\frac{\begin{array}{l}\boldsymbol{\Sigma} \gg \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \text{mgu } (\boldsymbol{B_x} \approx \boldsymbol{B_c}) : \Omega_A, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_c} \\ \omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_c \\ g = \epsilon\Omega'.\ \Omega_A[\omega]; (\boldsymbol{\lambda(\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A])\boldsymbol{.\ (c\ \Gamma_c)}[\omega]) \mapsto f \\ g \text{ in } \bar{c}\end{array}}{\boldsymbol{\Sigma, c{:}(\Pi\Gamma_c.\ B_c)} \gg \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{scUnify}$$

$$\frac{\begin{array}{l}\boldsymbol{\Sigma} \gg \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_c}) \text{ do not unify}\end{array}}{\boldsymbol{\Sigma, c{:}(\Pi\Gamma_c.\ B_c)} \gg \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{scSkip}$$

These rules may appear overwhelming, but it is best to explain them operationally. For every constant $\boldsymbol{c}$ in the signature, we need to see if it is applicable. For example, similar to our preceding example explaining unifiers, consider that we are constructing a function of type $\forall(\boldsymbol{A}{\in}\mathbf{o};\ \boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A})).\ \tau$.[3] The constants for λ-expressions, **app** and **lam** will be ignored via scSkip. However, the constant **impi** cannot be skipped with scSkip as the result types do unify. Therefore, we must process **impi** with scUnify which uses a most-general unifier to create a case that will match all objects whose top-level constructor is **impi**.

The rule scUnify follows from our preceding discussion of unifiers, but with a generalization to handle local parameters $\boldsymbol{\Gamma_x}$. We are splitting on objects of type $\boldsymbol{\Pi\Gamma_x.\ B_x}$. Recall that objects of this type have the form $\boldsymbol{\lambda\Gamma_x}[\omega']\boldsymbol{.\ M}$. The substitution ω' is needed to reflect that the variables in Ω_A can be refined, which $\boldsymbol{\Gamma_x}$ may be dependent upon. If $\boldsymbol{c{:}(\Pi\Gamma_c.\ B_c)}$ may be used in constructing an $\boldsymbol{M}$, then objects can have the form $\boldsymbol{\lambda\Gamma_x}[\omega]\boldsymbol{.\ c\ N_1 \ldots N_n}$. The important thing to notice is that $\boldsymbol{N_i}$ may make use of the local parameters in $\boldsymbol{\Gamma_x}$. Therefore, it is a crucial that

[3]Note that here we are splitting on the LF type **(nd *A*)** instead of the computation-level type $\langle\mathbf{nd}\ \boldsymbol{A}\rangle$ as we did in the earlier example.

the domain of ω is specified as $\Omega_A, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_c}$ instead of $\Omega_A, \boldsymbol{\Gamma_c}, \boldsymbol{\Gamma_x}$. When we unify $\boldsymbol{B_x}$ with $\boldsymbol{B_c}$ we only allow refinements of the variables in Ω_A and $\boldsymbol{\Gamma_c}$. Since $\boldsymbol{\Gamma_x}$ is in the domain of the substitution, we stipulate that the resulting ω must be the identity on all elements in $\boldsymbol{\Gamma_x}$ by stating that $\omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_c$. Therefore, (1) Ω' expresses the resulting pattern variables, (2) ω_A expresses the refinements needed to variables in Ω_A, and (3) ω_c expresses the refinements needed to variables in $\boldsymbol{\Gamma_c}$. We then construct g, which provides a pattern that will always match objects whose top-level constructor is $\boldsymbol{c}$. As long as a case of the form g is in $\overline{c}$, then we have adequately accounted for the constant $\boldsymbol{c}$.

One observation to be made is that the codomain of the substitution places all pattern variables to the left of $\boldsymbol{\Gamma_x}$, which is of the desired form we need to generate g. We mentioned that there may be many most-general unifiers with different codomains. At first glance it may seem overly restrictive to restrict the codomain to have this particular form, but it is not. For example, if an object $\boldsymbol{M}$ is well-typed in $\boldsymbol{(\Gamma, x{:}A, y{:}B)}$, then we can always move $\boldsymbol{y}$ to the left of $\boldsymbol{x}$ as $\boldsymbol{M}$ is equivalent to to $\boldsymbol{M}[\uparrow_{\boldsymbol{x}}\uparrow_{\boldsymbol{y}} \mathbf{id}_{\boldsymbol{\Gamma}}, \boldsymbol{x/x}, \boldsymbol{(y\ x)/y}]$ in $\boldsymbol{(\Gamma, y{:}(\Pi x{:}A.\ B), x{:}A)}$. In other words, pattern variables can always be moved to the left by changing their type to be Π-quantified by the variables we move them over. This process is known as *raising* and was used heavily in Twelf (Schürmann 2000), but we have been able to improve and abstract away from raising in the work here.

It is straightforward that the rule scUnify would fail if a most-general unifier exists but no compatible case is found in $\overline{c}$. One subtle observation is that it is possible for scSkip not to apply, and the rule scUnify fail in the construction of the most-general unifier. One such example would be if we were constructing a function of type $\forall(\boldsymbol{F}{\in}\mathbf{(o \to o)}; \boldsymbol{A}{\in}\mathbf{o};\ u{\in}\mathbf{(nd\ (}\boldsymbol{F}\ \boldsymbol{A}\mathbf{)}\mathbf{)}).\ \tau$. This type indicates that we are splitting on objects of type $\mathbf{nd\ (}\boldsymbol{F}\ \boldsymbol{A}\mathbf{)}$, and creating a unifier for **impi** will

correspond to solving when $(\boldsymbol{F}\ \boldsymbol{A}) \approx (\boldsymbol{B_1} \Rightarrow \boldsymbol{B_2})$. Unfortunately, there is no unique solution to this unification problem, so the algorithm will fail. We are guaranteed that unification will succeed if we stay within the decidable pattern fragment of higher-order unification (Dowek et al. 1998). Formally, this means that we only allow unification problems over objects of the form $\boldsymbol{E}\ \boldsymbol{x_1}\ \ldots\ \boldsymbol{x_n}$ where $\boldsymbol{x_i}$ is a fresh parameter (with respect to $\boldsymbol{E}$) and all $\boldsymbol{x_i}$'s are distinct.

5.3.3 Rules for Local Parameter Coverage

The judgment $\boldsymbol{\Gamma} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.}\ \boldsymbol{B_x}))$. τ is used to traverse through the local parameters $\boldsymbol{\Gamma_x}$ and check that there is a case for every compatible local parameter, in exactly the same manner as we checked for every constant in the signature. Therefore, the $\boldsymbol{\Gamma}$ in this judgment is initialized with $\boldsymbol{\Gamma_x}$, and the rules are as follows:

$$\frac{}{\cdot \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.}\ \boldsymbol{B_x}))\text{. } \tau}\ \mathsf{IcEmpty}$$

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.}\ \boldsymbol{B_x}))\text{. } \tau \\ \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \text{mgu } (\boldsymbol{B_x} \approx \boldsymbol{B_p}) : \Omega_A, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_p} \\ \omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_p \\ g = \epsilon\Omega'.\ \Omega_A[\omega]; (\boldsymbol{\lambda(\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A])\boldsymbol{.}\ (\boldsymbol{p}\ \boldsymbol{\Gamma_p})[\omega]) \mapsto f \\ g \text{ in } \overline{c}\end{array}}{\boldsymbol{\Gamma, p{:}(\Pi\Gamma_p.}\ \boldsymbol{B_p}) \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.}\ \boldsymbol{B_x}))\text{. } \tau}\ \mathsf{IcUnify}$$

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.}\ \boldsymbol{B_x}))\text{. } \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_p}) \text{ do not unify}\end{array}}{\boldsymbol{\Gamma, p{:}(\Pi\Gamma_p.}\ \boldsymbol{B_p}) \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.}\ \boldsymbol{B_x}))\text{. } \tau}\ \mathsf{IcSkip}$$

The rules IcEmpty, IcUnify, and IcSkip are virtually identical to their Signature Coverage counterparts. This highlights that we treat local parameters exactly the same as constants in the signature. It is in Local Parameter Coverage that we

determine the need for a case handling $\boldsymbol{\lambda x.\ x}$ in bracket abstraction (Example 4.5.2) as $\boldsymbol{x}$ is a local parameter.

5.3.4 Rules for Schematic Parameter Coverage

The judgment $\mathcal{W} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x}))$. τ is used to traverse the world $\mathcal{W}$ and check that $\overline{c}$ contains a case for every parameter supported by $\mathcal{W}$. We call this Schematic Parameter Coverage, and the rules are as follows:

$$\frac{}{\cdot \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x}))\text{. } \tau}\ \textsf{wcEmpty}$$

$$\frac{\begin{array}{l}\mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x}))\text{. } \tau \\ \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \text{mgu } (\boldsymbol{B_x} \approx \boldsymbol{B_b}) : \Omega_b, \Omega_A, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_b} \\ \omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_b \\ g = \epsilon\Omega'.\ \epsilon x_b{\in}(\boldsymbol{\Pi\Gamma_b.\ B_b})[\omega]^{\#}.\ \Omega_A[\omega]; (\boldsymbol{\lambda(\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A])\boldsymbol{.\ (x_b\ \Gamma_b)}[\omega]) \mapsto f \\ g \text{ in } \overline{c}\end{array}}{\mathcal{W}, (\Omega_b,\ (\boldsymbol{\Pi\Gamma_b.\ B_b})) \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x}))\text{. } \tau}\ \textsf{wcUnify}$$

$$\frac{\begin{array}{l}\mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x}))\text{. } \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_b}) \text{ do not unify}\end{array}}{\mathcal{W}, (\Omega_b,\ (\boldsymbol{\Pi\Gamma_b.\ B_b})) \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x}))\text{. } \tau}\ \textsf{wcSkip}$$

The rules for Schematic Parameter Coverage are also similar to those of Signature Coverage and Local Parameter Coverage. The rule wcUnify is slightly more complicated than in our previous versions. Recall that Ω_b contains the free variables in $\boldsymbol{(\Pi\Gamma_b.\ B_b)}$ and our definition of support (Figure 5.2) allows for any refinements of these variables. Therefore, the unifier ω is also allowed to refine these variables. In our previous judgments we had a concrete constant (scUnify) or local parameter (lcUnify) when building up our case g; we now we build up g using a pattern variable $\boldsymbol{x_b}$ that ranges over parameters.

5.3.5 Rules for Global Parameter Coverage

Functions are typed (and hence coverage-checked) in a context Ω. We check that all compatible parameters $\boldsymbol{y}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ in Ω are accounted for in our Global Parameter Coverage judgment, $\Omega \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x}))$. τ, which follows:

$$\frac{}{\cdot \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcEmpty}$$

$$\frac{\begin{array}{l}\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_c}) \text{ do not unify}\end{array}}{\Omega_g, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcSkipMismatch}$$

$$\frac{\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}{\Omega_g, \alpha{\in}\delta \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcSkipNonParameter}$$

$$\frac{\begin{array}{l}\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \mathcal{W} \text{ world} \\ \mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau \\ (\Omega_g, (\boldsymbol{\Pi\Gamma_c.\ B_c})) \in \mathcal{W}\end{array}}{\Omega_g, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcSkipWorld}$$

$$\frac{\begin{array}{l}\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ \mathcal{W}_2 \text{ world} \\ \mathcal{W}_2 \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau\end{array}}{\Omega_g, u\overset{\triangledown}{\in}\mathcal{W} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcCheckWorld}$$

The goal of our coverage checker is to ensure that programs never get stuck during runtime. During runtime, contexts do not contain any pattern-variable declarations $\alpha{\in}\delta$ and hence we ignore them in gcSkipNonParameter. Additionally, incompatible parameters can be ignored via gcSkipMismatch. We could have presented a rule similar to scUnify to allow for the handling of an explicit parameter in Ω, but we instead stipulate in gcSkipWorld that the parameter is supported by some world $\mathcal{W}$,

for which $\bar{c}$ is covered (via an appeal to Schematic Parameter Coverage). Finally, if we encounter a weakening declaration $u \overset{\nabla}{\in} \mathcal{W}$, then the type system will allow for the weakening of this function with respect to parameters supported by $\mathcal{W}$, and we check that $\bar{c}$ passes Schematic Parameter Coverage for some world $\mathcal{W}_2$ such that $\mathcal{W} \leq \mathcal{W}_2$. We need the $\leq$ generalization to handle world subsumption.

Example 5.3.4 (Global Parameter Coverage of evalBeta)**.**
In this example we explain the role of Global Parameter Coverage during the typing and executing of our evalBeta *function from Example 5.2.2.*

- *During typing, the cases in* evalBeta *are checked in the context* $(\cdot, u \overset{\nabla}{\in} \text{OPEN})$. *This passes Global Parameter Coverage using* gcCheckWorld *to call Schematic Parameter Coverage on* OPEN.
- *The function is recursively accessed in the* **lam** *case in scope of a new parameter* $\boldsymbol{x}$ *supported by* OPEN. *During execution, the same cases are still covered in the context* $(\cdot, u \overset{\nabla}{\in} \text{OPEN}, \boldsymbol{x} \overset{\nabla}{\in} \mathbf{exp}^{\#})$ *as* $\boldsymbol{x}$ *passes via* gcSkipWorld. *This same methodology allows us to argue that all extensions by supported parameters will still be covered.*
- *The sample execution of* evalBeta *in Example 5.2.4 executes the function by doing* $\nu w \in \text{CLOSED}.\ (\text{evalBeta} \backslash w)$. *This means during execution that the cases are evaluated in the context* $(\cdot, w \overset{\nabla}{\in} \text{CLOSED})$. *In this context, the cases are still covered by* gcCheckWorld *using Schematic Parameter Coverage on* OPEN *since* CLOSED $\leq$ OPEN, *which is enforced by our type system.*

5.3.6 Parameter Function Coverage

Our last auxiliary coverage judgment deals with the coverage of parameter functions. Recall that a function fn $\bar{c}$ may have type $(\nabla \boldsymbol{\Gamma}_{\boldsymbol{g}}.\ \forall \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau)$ as ν may range over cases. This is mostly used to extend a parameter function $(\forall \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau)$ to handle the parameters in $\boldsymbol{\Gamma}_{\boldsymbol{g}}$ (see "with" from Section 3.7.2).

In Ω, we may construct a covered parameter function of type $(\nabla \boldsymbol{\Gamma}_{\boldsymbol{g}}.\ \forall \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau)$ by providing (1) a case for every compatible parameter in $\boldsymbol{\Gamma}_{\boldsymbol{g}}$ and (2) providing a base

case for the parameters in Ω. The judgment $\mathbf{\Gamma} \gg^{\nabla} \overline{c}$ covers $\nabla\mathbf{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau$ handles the construction of cases to handle (1). The coverage of the entire parameter function will be checked in the rule coverNewLF$^{\#}$ which also checks for (2). Our judgment is initialized with $\mathbf{\Gamma_g}$ and is as follows:

$$\frac{}{\cdot \gg^{\nabla} \overline{c} \text{ covers } \nabla\mathbf{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau}\ \textsf{pfEmpty}$$

$$\frac{\begin{array}{l}\mathbf{\Gamma} \gg^{\nabla} \overline{c} \text{ covers } \nabla\mathbf{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau \\ \Omega', \mathbf{\Gamma_g} \vdash \omega = \text{mgu } (\boldsymbol{A_p} \approx \boldsymbol{A}) : \mathbf{\Gamma_g}, \Omega_A \\ \omega = ((\uparrow_{\Omega'} \cdot) + \mathbf{\Gamma_g}), \omega_A \\ g = \epsilon\Omega'.\ \nu\mathbf{\Gamma_g}.\ (\Omega_A[\omega]; \boldsymbol{p} \mapsto f) \\ g \text{ in } \overline{c}\end{array}}{\mathbf{\Gamma}, \boldsymbol{p}{:}\boldsymbol{A_p} \gg^{\nabla} \overline{c} \text{ covers } \nabla\mathbf{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau}\ \textsf{pfUnify}$$

$$\frac{\begin{array}{l}\mathbf{\Gamma} \gg^{\nabla} \overline{c} \text{ covers } \nabla\mathbf{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau \\ (\boldsymbol{A_p} \approx \boldsymbol{A}) \text{ do not unify}\end{array}}{\mathbf{\Gamma}, \boldsymbol{p}{:}\boldsymbol{A_p} \gg^{\nabla} \overline{c} \text{ covers } \nabla\mathbf{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau}\ \textsf{pfSkip}$$

Parameters of type $\boldsymbol{A}^{\#}$ are variables $\boldsymbol{x}$. Note that $\boldsymbol{A}$ may be a functional type as one may create parameters of any arbitrary type. However, unlike in our previous judgments we do not treat functional types different from base (atomic) types. The interesting rule is pfUnify which checks if a parameter in $\mathbf{\Gamma_g}$ is compatible with respect to refinements of the variables in Ω_A. The resulting case g is also similar to our previous constructions except that it is constructed with $\nu\mathbf{\Gamma_g}$.

5.3.7 Complete Coverage

We have discussed the auxiliary judgments and now turn to the main coverage judgment used in typing, $\Omega \vdash \overline{c}$ covers τ, which determines if $\overline{c}$ is covered in Ω. This is the main heart of the algorithm, and we present each rule in turn:

$$\frac{c = \epsilon\alpha{\in}\delta.\ \alpha \mapsto f}{\Omega \vdash c \text{ covers } \forall\alpha{\in}\delta.\ \tau}\ \mathsf{coverSimple}$$

The first rule, coverSimple expresses the simplest function one may write by binding the argument to a pattern variable of the appropriate type. More specifically, this rule states that the function (fn $(\alpha{\in}\delta) \mapsto f$) is covered.

$$\frac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau\ \mathsf{wff} \\ c = \epsilon\boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y}\ \boldsymbol{\Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y}\ \boldsymbol{\Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}$$

The function (fn $(\boldsymbol{M},\ U) \mapsto f$) is a covered function performing case analysis over pairs of the form $\exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau$ by decomposing the pair into its two parts: $\boldsymbol{M}$ and U. If we instead wished to perform case analysis over pairs of the form $\nabla\boldsymbol{y}{\in}\boldsymbol{B}^{\#}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau$, then the parameter $\boldsymbol{y}$ may occur in both elements of the pair, and we can write a covered function as (fn $(\nu\boldsymbol{y}.\ ((\boldsymbol{M}\ \boldsymbol{y}),\ (U\backslash\boldsymbol{y}))) \mapsto f$). The important observation here is that the pattern variables ($\boldsymbol{M}$ and U) express abstractions as the parameter $\boldsymbol{y}$ may occur in both elements of the pair. The LF object $(\boldsymbol{M}\ \boldsymbol{y})$ expresses an instance of higher-order matching where $\boldsymbol{M}$ is the function resulting from the abstraction of all occurrences of $\boldsymbol{y}$ from the first element of the pair. The computation-level expression $(U\backslash\boldsymbol{y})$ is analogous to higher-order matching but at the computation level. This illustrates an example where the patterns we express are not just values but may also be computation-level expressions using the \ construct. Finally, the rule coverPairLF generalizes this discussion from one parameter into an arbitrary $\boldsymbol{\Gamma}$.

As an example of coverPairLF recall that the type $\langle\boldsymbol{A}\rangle$ is sugar for $\exists\boldsymbol{x}{\in}\boldsymbol{A}.$ unit and hence the recursive call in the **lam** case of evalBeta (Example 5.2.2) uses this rule to match the result of the recursive call (which has type $\nabla\boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \langle\mathbf{exp}\rangle$) with the pattern $(\nu\boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \langle\boldsymbol{E'}\ \boldsymbol{x}\rangle)$. It is worthwhile to note that if we desugar the $\langle\rangle$, we

see that we are technically matching against $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ ((\boldsymbol{E'}\ \boldsymbol{x}),\ ()))$ but the rule specifies that we should really match against $(\nu \boldsymbol{x} \in \mathbf{exp}^{\#}.\ ((\boldsymbol{E'}\ \boldsymbol{x}),\ (u \backslash \boldsymbol{x})))$ where u is a pattern variable of type $\nabla \boldsymbol{x} \in \mathbf{exp}^{\#}.$ unit. However, (1) we do not actually care about the second element of the pair and (2) the pattern () is indeed covered as the only value of type unit is () irregardless of being in scope of any parameters. This is an example of utilizing *subordination*, which we discuss in Section 5.3.8.

$$\frac{\begin{array}{l}\cdot \vdash \exists \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff} \\ c = \epsilon \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \epsilon u \in \tau.\ (\boldsymbol{x},\ u) \mapsto f\end{array}}{\Omega \vdash c\ \text{covers}\ (\exists \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}^{\#}$$

We can also decompose a pair whose first element is a parameter by using $\mathsf{coverPairLF}^{\#}$. The interesting observation for this rule is that we do not allow the decomposition of a pair under parameters $\boldsymbol{\Gamma}$, although one can imagine extending this rule with support of such cases. Notice that the support of such decompositions would require an auxiliary judgment to traverse $\boldsymbol{\Gamma}$ and check that a case exists for all compatible cases, similar to our Parameter Function Coverage judgment. We have not found any need for such decompositions.

$$\frac{\begin{array}{l}\cdot \vdash \nabla \boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)\ \mathsf{wff} \\ c = \epsilon u_1 \in (\nabla \boldsymbol{\Gamma}.\ \tau_1).\ \epsilon u_2 \in (\nabla \boldsymbol{\Gamma}.\ \tau_2).\ (\nu \boldsymbol{\Gamma}.\ (u_1 \backslash \boldsymbol{\Gamma},\ u_2 \backslash \boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c\ \text{covers}\ (\nabla \boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma}\ \mathsf{coverPairMeta}$$

The rule $\mathsf{coverPairMeta}$ allows us to decompose computation-level pairs $(\nabla \boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2))$ in the same manner as $\mathsf{coverPairLF}$. The function $(\text{fn}\ (\nu \boldsymbol{\Gamma}.\ (u_1 \backslash \boldsymbol{\Gamma},\ u_2 \backslash \boldsymbol{\Gamma})) \mapsto f)$ is a covered function performing this decomposition.

$$\frac{\begin{array}{l}(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})\ \mathsf{ctx} \\ \Omega_A \text{ only contains declarations of type } \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#} \\ \boldsymbol{\Sigma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau \\ \boldsymbol{\Gamma_x} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau \\ \Omega \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau\end{array}}{\Omega \vdash \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau}\ \mathsf{coverLF}$$

We have seen that our coverage algorithm can decompose pairs into their individual components, but the rule coverLF is the one that actually considers multiple cases based on the different ways objects of an LF type can be constructed. We can write a function of type $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by splitting on objects of type $\boldsymbol{B_x}$. We check that the list of cases handles every compatible (1) constant in the signature $\boldsymbol{\Sigma}$ (Signature Coverage), (2) variable in $\boldsymbol{\Gamma_x}$ (Local Parameter Coverage), and (3) parameter in Ω (Global Parameter Coverage). Since Ω_A is used to express variables that are refined during case analysis and we do not permit dependencies on computation-level variables, there is no reason for Ω_A to contain variables of type τ.

The restriction that $(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})$ ctx allows us to guarantee that all free variables in $\boldsymbol{B_x}$ are quantified either in $\boldsymbol{\Gamma_x}$ or Ω_A. As an example why this is necessary, consider the function (fn $(\boldsymbol{A}{\in}\mathbf{o}) \mapsto$ fn $\overline{c}$) of type $\forall\boldsymbol{A}{\in}\mathbf{o}.\ \forall\boldsymbol{d}{\in}(\mathbf{nd}\ \boldsymbol{A}).\ \tau$ for any τ. Now the cases $\overline{c}$ reflect case analysis on objects of type $(\mathbf{nd}\ \boldsymbol{A})$ but the $\boldsymbol{A}$ is not refinable, which means that it is not even type correct to give a pattern utilizing **impi** even though during runtime it is perfectly plausible that such a situation may arise. Recall that this is the reason why we extended the $\forall$ type over Ω_A (Section 4.1.2). This well-formed side condition simplifies our auxiliary judgments (as we do not need to pass around Ω) but it is not necessary. In this last example, if we did try Signature Coverage on objects of type $(\mathbf{nd}\ \boldsymbol{A})$, then when analyzing the constant **impi** neither scSkip nor scUnify would apply and hence we would catch that it is impossible to perform this split. The rule scUnify would fail because it would

not be able to construct a unifier that did *not* refine $\boldsymbol{A}$.

$$\frac{\Omega \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x'} {\in} \boldsymbol{A}^{\#}.\ \tau \quad \Omega_2 \text{ closed}}{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c} \backslash \boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$$

During evaluation, the expression (fn $\overline{c}$)$\backslash \boldsymbol{x}$ is reduced to fn ($\overline{c} \backslash \boldsymbol{x}$). We want to guarantee that evaluation preserves the typing (and hence coverage) of expressions. Therefore, rule coverPop handles this case and says that the coverage of ($\overline{c} \backslash \boldsymbol{x}$) depends on the coverage of $\overline{c}$. The side condition on Ω_2 is the same as in the typing rule for $c \backslash \boldsymbol{x}$ (cPop).

$$\frac{\cdot \vdash \textsc{closed} \text{ supports } \Omega}{\Omega \vdash \mathit{nil} \text{ covers } (\forall (\Omega_A, \boldsymbol{x} {\in} \boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverEmpty}^{\#}$$

Our examples have shown that we make heavy use of extending parameter functions. However, our example executions required us to provide an initial parameter function. The rule coverEmpty$^{\#}$ states that the empty function (fn $\cdot$) is a covered parameter function in Ω as long as (1) Ω does not contain any parameters and (2) Ω cannot be weakened by any parameters. We achieve this check via our premise "$\cdot \vdash$ CLOSED supports Ω" which checks that the context does not contain any declarations $\boldsymbol{y} \overset{\triangledown}{\in} \boldsymbol{B}^{\#}$ and if it contains weakening declarations, they must only be of world CLOSED.

$$\frac{\begin{array}{l}(\Omega_A, \boldsymbol{x} {\in} \boldsymbol{A}^{\#})\ \mathsf{ctx} \\ \text{All elements of } \Omega_A \text{ occur free in } \boldsymbol{A} \\ g = \epsilon \Omega_A.\ \epsilon \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ \nu \boldsymbol{\Gamma_g}.\ (\Omega_A; \boldsymbol{x} \mapsto f) \\ g \text{ in } \overline{c} \\ \boldsymbol{\Gamma} \gg^{\nabla} \overline{c} \text{ covers } \nabla \boldsymbol{\Gamma}.\ (\forall \Omega_A, \boldsymbol{x} {\in} \boldsymbol{A}^{\#}.\ \tau)\end{array}}{\Omega \vdash \overline{c} \text{ covers } \nabla \boldsymbol{\Gamma}.\ (\forall (\Omega_A, \boldsymbol{x} {\in} \boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverNewLF}^{\#}$$

The rule coverNewLF$^{\#}$ states that we can construct a covered parameter function in Ω extended by $\boldsymbol{\Gamma}$ if our list of cases provide (1) cases for all compatible parameters in $\boldsymbol{\Gamma}$ and (2) a case g which handles all parameters in Ω. Condition (1) is checked

via the auxiliary Parameter Function Coverage. The "with" construct described in Section 3.7.2 is syntactic sugar for a construction extending parameter functions in this manner.

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \Vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type} \\ c = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}).\ \nu\boldsymbol{\Gamma}.\ ((\boldsymbol{x}\ \boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)}\ \mathsf{coverNewLF}$$

$$\frac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \sigma\ \mathsf{wff} \\ c = \epsilon u \in (\nabla\boldsymbol{\Gamma}.\ \sigma).\ \nu\boldsymbol{\Gamma}.\ ((u \backslash \boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\sigma \supset \tau)}\ \mathsf{coverNewMeta}$$

The last of our rules are coverNewLF and coverNewMeta which handle creating functions that utilize ν over cases to refine objects and expressions of type $\boldsymbol{A}$ and σ, respectively. As an example, the coverage of part 2 of Lemma 3.4.2 uses coverNewMeta.

Finally, we comment that the only pattern one can write to match computation-level functional-types $\nabla\boldsymbol{\Gamma}.\ (\forall\Omega.\ \tau)$ correspond to matching against a single pattern variable u (from coverSimple).

5.3.8 Coverage Enhancements

Our coverage-checking judgment is powerful enough to write all of our examples. However, in order to simplify the presentation of our examples (as well as introduce a more user-friendly implementation) we enhance the above algorithm with support of multiple refinements, pattern generalizations, as well as subordination:

1. *Multiple Refinements:* Technically, in order to perform case analysis over an expression of type $\langle \mathbf{nat} \rangle$ (Example 2.1.1) we will need to first use coverPairLF to get an LF object of type **nat** and then use coverLF to split on this type. For

example a function over $\langle \mathbf{nat} \rangle$ would look as follows:

$$\text{fn } \langle \boldsymbol{N} \rangle \mapsto \text{case } \boldsymbol{N} \text{ of } \mathbf{z} \mapsto f_1 \mid (\mathbf{s}\ \boldsymbol{N_1}) \mapsto f_2$$

We allow the user to express these refinements together as:

$$\text{fn } \langle \mathbf{z} \rangle \mapsto f_1 \mid \langle \mathbf{s}\ \boldsymbol{N_1} \rangle \mapsto f_2$$

This same technique also allows us to internalize a second use of coverLF on $\boldsymbol{N_1}$ and write a covered function of the form:

$$\text{fn } \langle \mathbf{z} \rangle \mapsto f_1 \mid \langle \mathbf{s}\ \mathbf{z} \rangle \mapsto f_{2a} \mid \langle \mathbf{s}\ (\mathbf{s}\ \boldsymbol{N_1'}) \rangle \mapsto f_{2b}$$

2. *Pattern Generalizations:* In our example of bracket abstraction (Example 4.5.2), the rule coverLF determines that we need to provide a case for (1) the identity function, (2) a function using the constant **MP**, (3) a function using the constant **K**, (4) a function using the constant **S**, and (5) a function using any parameter $\boldsymbol{y}^{\#}$. Therefore, we need cases of the form (1) $\langle \boldsymbol{\lambda x.\ x} \rangle \mapsto f_1$, (2) $\langle \boldsymbol{\lambda x.}\ \mathbf{MP}\ (\boldsymbol{H_1}\ \boldsymbol{x})\ (\boldsymbol{H_2}\ \boldsymbol{x}) \rangle \mapsto f_2$, (3) $\langle \boldsymbol{\lambda x.}\ \mathbf{K} \rangle \mapsto f_3$, (4) $\langle \boldsymbol{\lambda x.}\ \mathbf{S} \rangle \mapsto f_4$, and (5) $\langle \boldsymbol{\lambda x.}\ \boldsymbol{y}^{\#} \rangle \mapsto f_5$. However, we intend to do the same thing for cases (3), (4), and (5). Therefore, we permit the programmer to write a more general case: $\langle \boldsymbol{\lambda x.\ H} \rangle \mapsto f_*$ where $\boldsymbol{H}$ is a pattern variable and hence handles the last three cases.

3. *Subordination:* When we perform case analysis over a type under parameters $\boldsymbol{\Gamma}$, the result is abstracted by all parameters. However, in our discussion of coverPairLF we already saw one example when this abstraction is not neces-

sary, i.e. in handling the unit type. Even more importantly, this comes up with respect to LF types. For example, consider the cntvar function from Example 3.3.7. In the **lam** case we performed case analysis over the type $\nabla \boldsymbol{x} \overset{\triangledown}{\in} \mathbf{exp}^{\#}.\ \langle \mathbf{nat} \rangle$ and matched against the pattern $\nu \boldsymbol{x} \overset{\triangledown}{\in} \mathbf{exp}^{\#}.\ \langle \boldsymbol{N} \rangle$. Technically, our rules stipulate that in order to guarantee the function is covered, the pattern should instead be $\nu \boldsymbol{x} \overset{\triangledown}{\in} \mathbf{exp}^{\#}.\ \langle \boldsymbol{N'}\ \boldsymbol{x} \rangle$. However, as we discussed in the example, this latter pattern is not necessary as we are guaranteed that the $\boldsymbol{x}$ cannot occur in objects of type **nat**. Formally, we say that **exp** is not *subordinate* to **nat**. Therefore, we enhance our algorithm to take into account subordination information.

It is important to emphasize that this is only an enhancement and does not increase the computational strength of our system. If we had not made this enhancement, then the matching in cntvar would yield an LF object of type $(\mathbf{exp} \rightarrow \mathbf{nat})$ and one could call a function subord, which is as follows:

$$
\begin{array}{l}
\mu \mathsf{subord} \in \langle \mathbf{exp} \rightarrow \mathbf{nat} \rangle \supset \langle \mathbf{nat} \rangle. \\
\quad \text{fn}\ \ \langle \boldsymbol{\lambda x.}\ \mathbf{z} \rangle \qquad\quad \mapsto \mathbf{z} \\
\quad\ \ |\ \ \langle \boldsymbol{\lambda x.}\ \mathbf{s}\ (\boldsymbol{N}\ \boldsymbol{x}) \rangle \mapsto \text{let}\ \langle \boldsymbol{x} \rangle = \mathsf{subord}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{N}\ \boldsymbol{x} \rangle\ \text{in}\ \langle \mathbf{s}\ \boldsymbol{x} \rangle
\end{array}
$$

However, notice that the function subord is not doing anything interesting as objects of type **exp** cannot occur in objects of type **nat**.

5.4 Programs as Proofs

We have alluded to using Delphin to write proofs, which is the subject of this section. A function is *total* if, for all possible inputs, it is guaranteed to always terminate with a value. If one can guarantee that a function is total, then the function can

be interpreted as a proof that for every well-typed input there exists a well-typed output.

This correspondence does not yield meaningful proofs when restricted to simple types. For example, the plus function (Example 3.3.3) adds two numbers and has the type **nat** $\supset$ **nat** $\supset$ **nat**. The totality of this function only entails that for every two numbers, there exists a number.

However, when we use more expressive types, i.e. dependent types, the totality of functions gives us more meaningful information. Assume we represent the addition of two numbers as objects of type **add** $\boldsymbol{N_1}$ $\boldsymbol{N_2}$ $\boldsymbol{R}$, where $\boldsymbol{R}$ is the result of adding $\boldsymbol{N_1}$ and $\boldsymbol{N_2}$. In this example, a total function of type $\langle$**add** $\boldsymbol{N_1}$ $\boldsymbol{N_2}$ $\boldsymbol{R}\rangle \supset \langle$**add** $\boldsymbol{N_2}$ $\boldsymbol{N_1}$ $\boldsymbol{R}\rangle$ is also a proof that our **add** operation is commutative.

When interpreting functions as proofs, a recursive call corresponds to an appeal of the induction hypothesis. The fact that something needs to get smaller in every recursive call corresponds to (1) guaranteeing that the function terminates or (2) guaranteeing that every call to the induction hypothesis is valid.

5.4.1 Totality: Progress and Termination

In order to determine if a function is total, we must check that the function yields an output for every input. This corresponds to guaranteeing that functions never get stuck as well as ensuring that they always terminate.

Guaranteeing that programs never get stuck (progress) is a property of our type system, which is proved in Theorem 5.5.22 and relies on the soundness of our coverage checker. However, Delphin is designed to be a general purpose programming language, and hence programs may not terminate. For example, we may write the function μfoo. fn $u \mapsto$ foo u. Therefore, determining if a program terminates requires an external check.

The main challenge in testing termination is in guaranteeing that the arguments of every recursive call get smaller with respect to some well-founded ordering. Our implementation (Chapter 6) includes a termination checker supporting lexicographic extensions of the subterm ordering over the inputs. Determining if one expression is a subterm of another requires a subterm relation on LF objects, which has been formalized by Rohwedder and Pfenning (1996). The support of lexicographic extensions allows us to look at multiple inputs together. For example, if a function has two arguments, then the termination order can express that (1) either the first argument gets smaller, or (2) the first argument stays the same and the second gets smaller. Lexicographic extensions to the subterm ordering suffice for our examples here and are also the orderings used in Twelf (Pfenning and Schürmann 1998). However, more powerful orderings may also be used. The main contributions of this chapter is with respect to coverage (for type safety) and we defer a more detailed analysis of termination to future work.

In Section 5.2.4 we demonstrated and discussed how we can annotate our evalBeta function (Example 5.1.1) with world information (Example 5.2.2). This conversion is straightforward and automatically performed in our implementation (Chapter 6). We will now illustrate the conversion and proof interpretation of the dependently-typed examples from Sections 4.5 and 4.6.

5.4.2 Natural Deduction vs. Combinators

This section extends the functions from Section 4.5 with world information and discusses termination. We will write two functions converting between derivations in our natural deduction calculus (Example 2.2.3) and derivations in our Hilbert-style calculus, also known as typed combinators (Example 2.2.4). The totality of both functions allows us to conclude that any formula derivable in one calculus can also

be derived in the other, i.e. both calculi are equivalent with respect to what they can prove.

From Combinators

We start with the easy direction of converting combinators into natural deduction derivations, as shown below:

Example 5.4.1 (Combinators to Natural Deduction Derivations with Worlds)**.**

μhil2nd $\in \nabla$CLOSED. ($\langle$**comb** $\boldsymbol{A}\rangle \supset \langle$**nd** $\boldsymbol{A}\rangle$).
$\nu u \in$CLOSED.

$$
\begin{array}{rlll}
\text{fn} & \langle \mathbf{K} \rangle & \mapsto & \langle \mathbf{impi}\ (\boldsymbol{\lambda x}\mathbf{:(nd}\ \boldsymbol{A}\mathbf{).}\ \mathbf{impi}\ (\boldsymbol{\lambda y}\mathbf{:(nd}\ \boldsymbol{B}\mathbf{).}\ \boldsymbol{x}\mathbf{))} \rangle \\
\mid & \langle \mathbf{S} \rangle & \mapsto & \langle \mathbf{impi}\ (\boldsymbol{\lambda x}\mathbf{:(nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B} \Rightarrow \boldsymbol{C})\mathbf{).} \\
 & & & \quad \mathbf{impi}\ (\boldsymbol{\lambda y}\mathbf{:(nd}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B})\mathbf{).} \\
 & & & \quad\quad \mathbf{impi}\ (\boldsymbol{\lambda z}\mathbf{:(nd}\ \boldsymbol{A}\mathbf{).} \\
 & & & \quad\quad\quad \mathbf{impe}\ (\mathbf{impe}\ \boldsymbol{x}\ \boldsymbol{z})\ (\mathbf{impe}\ \boldsymbol{y}\ \boldsymbol{z}))))\rangle \\
\mid & \langle \mathbf{MP}\ \boldsymbol{H_1}\ \boldsymbol{H_2} \rangle & \mapsto & \text{let } \langle \boldsymbol{D_1} \rangle = (\mathsf{hil2nd} \backslash u)\ \langle \boldsymbol{H_1} \rangle \text{ in} \\
 & & & \text{let } \langle \boldsymbol{D_2} \rangle = (\mathsf{hil2nd} \backslash u)\ \langle \boldsymbol{H_2} \rangle \text{ in} \\
 & & & \quad \langle \mathbf{impe}\ \boldsymbol{D_1}\ \boldsymbol{D_2} \rangle
\end{array}
$$

This function is modified from our previous version such that it is annotated as supporting world CLOSED. The older function was also typeable in our new system as we never introduce new parameters. Recall that a function of type $\langle$**comb** $\boldsymbol{A}\rangle \supset \langle$**nd** $\boldsymbol{A}\rangle$ expresses a function that is not weakenable over any parameters while a function of type ∇CLOSED. ($\langle$**comb** $\boldsymbol{A}\rangle \supset \langle$**nd** $\boldsymbol{A}\rangle$) expresses a function that is weakenable over parameters supported by CLOSED, which are none. Therefore, world annotations did not play a significant role in this function, but we will see it is necessary in the reverse direction.

This function is covered as there are no parameters supported by CLOSED and there is a case for every constant in the signature (Signature Coverage) that can

be used to construct expressions of type $\langle \mathbf{comb}\ \boldsymbol{A} \rangle$. We can easily see that the function terminates as the only recursive calls occur on subterms of the input, i.e. the termination order is the subterm ordering on the only explicit input argument.

To Combinators

We now show the two-step algorithm that translates from natural deduction derivations to combinators. We begin with bracket abstraction:

Example 5.4.2 (Bracket Abstraction)**.**

$\mu \mathsf{ba} \in \nabla \textsc{open}.\ (\langle \mathbf{comb}\ \boldsymbol{A} \rightarrow \mathbf{comb}\ \boldsymbol{B} \rangle \supset \langle \mathbf{comb}\ (\boldsymbol{A} \Rightarrow \boldsymbol{B}) \rangle)$.
$\nu u \in \textsc{open}.$

$$
\begin{array}{lll}
\text{fn}\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{x} \rangle & \mapsto & \langle \mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \mathbf{K}) \\
 & & \qquad (\mathbf{K} : \mathbf{comb}\ (\boldsymbol{A} \Rightarrow \boldsymbol{A} \Rightarrow \boldsymbol{A})) \rangle \\
\mid \langle \boldsymbol{\lambda x.}\ \mathbf{MP}\ (\boldsymbol{H_1}\ \boldsymbol{x})\ (\boldsymbol{H_2}\ \boldsymbol{x}) \rangle & \mapsto & \text{let}\ \langle \boldsymbol{H_1'} \rangle = (\mathsf{ba} \backslash u)\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{H_1}\ \boldsymbol{x} \rangle\ \text{in} \\
 & & \text{let}\ \langle \boldsymbol{H_2'} \rangle = (\mathsf{ba} \backslash u)\ \langle \boldsymbol{\lambda x.}\ \boldsymbol{H_2}\ \boldsymbol{x} \rangle\ \text{in} \\
 & & \qquad \langle \mathbf{MP}\ (\mathbf{MP}\ \mathbf{S}\ \boldsymbol{H_1'})\ \boldsymbol{H_2'} \rangle \\
\mid \langle \boldsymbol{\lambda x.}\ \boldsymbol{H} \rangle & \mapsto & \langle \mathbf{MP}\ \mathbf{K}\ \boldsymbol{H} \rangle
\end{array}
$$

We declare ba to make sense with respect to world OPEN. The function by itself would also make sense in CLOSED, but we make a stronger statement by declaring it to respect OPEN. This function is intended to be used as a lemma (or helper function) in the main conversion function, which will have world OPEN. Therefore, in order for the main function to call this function, it must also have world OPEN (or some world that subsumes it). In other words, we will access ba in scope of parameters compatible with OPEN, and hence the function ba must also support such parameters.

In this example we are doing case analysis over an LF function in world OPEN. Local Parameter Coverage determines that we need to have a case for (1) $\langle \boldsymbol{\lambda x.}\ \boldsymbol{x} \rangle$. Additionally, Signature Coverage determines that we need to have cases for (2) $\langle \boldsymbol{\lambda x.}\ \mathbf{MP}\ (\boldsymbol{H_1}\ \boldsymbol{x})\ (\boldsymbol{H_2}\ \boldsymbol{x}) \rangle$, (3) $\langle \boldsymbol{\lambda x.}\ \mathbf{K} \rangle$, and (4) $\langle \boldsymbol{\lambda x.}\ \mathbf{S} \rangle$. Finally, Global Param-

eter Coverage appeals to Schematic Parameter Coverage which determines that we need to have a case for (5) $\langle \boldsymbol{\lambda x.\ y}^{\#} \rangle$. Cases (3), (4), and (5) would all do the same thing, so as an enhancement we reduce it to one case $\langle \boldsymbol{\lambda x.\ H} \rangle$ where $\boldsymbol{H}$ is a pattern variable which can match all three cases. This last case partly overlaps with the **MP** case and it returns a different (shorter) combinator, which is also valid.

This function terminates as recursive calls are made on subterms of the (explicit) input. However, it may not be immediately obvious that $\langle \boldsymbol{\lambda x.\ H_1\ x} \rangle$ is a subterm of $\langle \boldsymbol{\lambda x.}\ \mathbf{MP}\ \boldsymbol{(H_1\ x)\ (H_2\ x)} \rangle$. Intuitively, one may consider $\langle \boldsymbol{\lambda \Gamma.\ M_1} \rangle$ as a subterm of $\langle \boldsymbol{\lambda \Gamma.\ M_2} \rangle$ if M_1 is a subterm of M_2. However, although this is a sufficient property to conclude that one expression is a subterm of another, it is not a necessary one. The actual subterm ordering (Rohwedder and Pfenning 1996) exploits properties of subordination (Section 5.3.8) to yield a more powerful ordering.

Next we write the function nd2hil which traverses natural deduction derivations and uses ba to convert them into combinators. This function dynamically introduces parameters of **nd** $\boldsymbol{A}$ and **comb** $\boldsymbol{A}$ together. We therefore declare it to respect world OPEN and give the new code below:

Example 5.4.3 (Natural Deduction Derivations to Combinators)**.**

$$\text{type } \mathsf{ndParamFun} = \langle (\mathbf{nd}\ \boldsymbol{A})^{\#} \rangle \supset \langle \mathbf{comb}\ \boldsymbol{A} \rangle$$

$$\begin{array}{l}
\mu\mathsf{nd2hil} \in \nabla\textsc{open}.\ (\mathsf{ndParamFun} \supset \langle \mathbf{nd}\ \boldsymbol{A} \rangle \supset \langle \mathbf{comb}\ \boldsymbol{A} \rangle). \\
\quad \nu u{\in}\textsc{open}. \\
\qquad \text{fn } W \mapsto \\
\qquad\quad \text{fn } \langle \mathbf{impi}\ (\boldsymbol{\lambda d.}\ \boldsymbol{D}\ \boldsymbol{d}) \rangle \ \mapsto \\
\qquad\qquad\quad (\text{case } (\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ (\mathsf{nd2hil}\backslash u)\ (W \text{ with } \langle \boldsymbol{d} \rangle \mapsto \langle \boldsymbol{h} \rangle)\ \langle \boldsymbol{D}\ \boldsymbol{d} \rangle) \\
\qquad\qquad\qquad\quad \text{of } (\nu \boldsymbol{d}.\ \nu \boldsymbol{h}.\ \langle \boldsymbol{H}\ \boldsymbol{h} \rangle) \ \mapsto \ (\mathsf{ba}\backslash u)\ \langle \boldsymbol{\lambda h.}\ \boldsymbol{H}\ \boldsymbol{h} \rangle) \\
\qquad\quad \mid \langle \mathbf{impe}\ \boldsymbol{D_1}\ \boldsymbol{D_2} \rangle \ \mapsto \\
\qquad\qquad\quad \text{let } \langle \boldsymbol{H_1} \rangle = (\mathsf{nd2hil}\backslash u)\ W\ \langle \boldsymbol{D_1} \rangle \text{ in} \\
\qquad\qquad\quad \text{let } \langle \boldsymbol{H_2} \rangle = (\mathsf{nd2hil}\backslash u)\ W\ \langle \boldsymbol{D_2} \rangle \text{ in} \\
\qquad\qquad\qquad \langle \mathbf{MP}\ \boldsymbol{H_1}\ \boldsymbol{H_2} \rangle \\
\qquad\quad \mid \langle \boldsymbol{d}^{\#} \rangle \ \mapsto \ W\ \langle \boldsymbol{d} \rangle
\end{array}$$

Our conversion of nd2hil proceeds as straightforwardly as the previous examples, but much more is happening. First, the appeal to ba is well-typed as we declared the world of ba to also be OPEN. Second, and most importantly, notice that the type of the parameter function ndParamFun does not change. The type of nd2hil is $\nabla\textsc{open}.\ (\mathsf{ndParamFun} \supset \langle \mathbf{nd}\ \boldsymbol{A} \rangle \supset \langle \mathbf{comb}\ \boldsymbol{A} \rangle)$ which states that in scope of any parameters supported by OPEN, if the user provides (1) a function to handle all the parameters, and (2) a derivation in the natural deduction calculus, then the function will return an equivalent combinator.

The first argument to the function is the parameter function W, which we have always *interpreted* as not being weakenable over parameters, necessitating the manual extension of W via the "with" syntax in the **impi** case. However, now the type truly does express that W must be explicitly extended. In fact, it is now *not* well-typed to replace $(W \text{ with } \langle \boldsymbol{d} \rangle \mapsto \langle \boldsymbol{h} \rangle)$ with W.

The new nd2hil function is covered as it provides cases for compatible constants in the signature as well as for parameters of type $\mathbf{nd}\ \boldsymbol{A}$, which are the only type of

parameters in OPEN that can occur in the input argument. Finally, the function is also terminating as the argument of type **nd** $\boldsymbol{A}$ gets smaller during every recursive call following the subterm ordering in Rohwedder and Pfenning (1996). More specifically, it terminates because (1) $\boldsymbol{D_1}$ and $\boldsymbol{D_2}$ are both subterms of **impe** $\boldsymbol{D_1}$ $\boldsymbol{D_2}$, and (2) $\boldsymbol{D}$ $\boldsymbol{d}$ is a subterm of **impi** $(\boldsymbol{\lambda d.}$ $\boldsymbol{D}$ $\boldsymbol{d})$. Perhaps the latter seems intuitive, but the reader should keep in mind that objects are equivalent modulo α, so we are also saying that $\boldsymbol{D}$ $\boldsymbol{d}$ is a subterm of **impi** $(\boldsymbol{\lambda d'.}$ $\boldsymbol{D}$ $\boldsymbol{d'})$. More specifically, given the LF term **impi** $(\boldsymbol{\lambda d'.}$ $\boldsymbol{D}$ $\boldsymbol{d'})$, then $\boldsymbol{D}$ $\boldsymbol{y}$ is a subterm for all parameters $\boldsymbol{y}$. Additionally, the subterm ordering is enhanced based on subordination (Section 5.3.8) so that under certain circumstances one may also conclude that $\boldsymbol{D}$ $\boldsymbol{M}$ is a subterm even if $\boldsymbol{M}$ is not a parameter; however, these advanced properties of the subterm ordering are not necessary for our examples here. The interested reader is referred to Rohwedder and Pfenning (1996) for a detailed explanation of this enhanced subterm ordering.

Example 5.4.4 (Sample nd2hil Execution with Worlds)**.**

$\nu w{\in}\text{CLOSED}.\ (\mathsf{nd2hil}\backslash w)\ (\text{fn}\ \cdot)\ \langle\textbf{impi}\ (\boldsymbol{\lambda d}\textbf{:(nd}\ \boldsymbol{p}\textbf{)}.\ \boldsymbol{d})\rangle$
$\ldots \rightarrow \nu w{\in}\text{CLOSED}.\ \text{case}\ (\nu\boldsymbol{d}.\ \nu\boldsymbol{h}.\ (\mathsf{nd2hil}\backslash w)\ (W\ \text{with}\ \langle\boldsymbol{d}\rangle \mapsto \langle\boldsymbol{h}\rangle)\ \langle\boldsymbol{d}\rangle)$
$\qquad\qquad \text{of}\ (\nu\boldsymbol{d}.\ \nu\boldsymbol{h}.\ \langle\boldsymbol{H}\ \boldsymbol{h}\rangle) \mapsto (\mathsf{ba}\backslash w)\ \langle\boldsymbol{\lambda d}\textbf{:(nd}\ \boldsymbol{p}\textbf{)}.\ \boldsymbol{H}\ \boldsymbol{d}\rangle$
$\ldots \rightarrow \nu w{\in}\text{CLOSED}.\ \text{case}\ (\nu\boldsymbol{d}.\ \nu\boldsymbol{h}.\ (W\ \text{with}\ \langle\boldsymbol{d}\rangle \mapsto \langle\boldsymbol{h}\rangle)\ \langle\boldsymbol{d}\rangle)$
$\qquad\qquad \text{of}\ (\nu\boldsymbol{d}.\ \nu\boldsymbol{h}.\ \langle\boldsymbol{H}\ \boldsymbol{h}\rangle) \mapsto (\mathsf{ba}\backslash w)\ \langle\boldsymbol{\lambda d}\textbf{:(nd}\ \boldsymbol{p}\textbf{)}.\ \boldsymbol{H}\ \boldsymbol{d}\rangle$
$\ldots \rightarrow \nu w{\in}\text{CLOSED}.\ \text{case}\ (\nu\boldsymbol{d}.\ \nu\boldsymbol{h}.\ \langle\boldsymbol{h}\rangle)$
$\qquad\qquad \text{of}\ (\nu\boldsymbol{d}.\ \nu\boldsymbol{h}.\ \langle\boldsymbol{H}\ \boldsymbol{h}\rangle) \mapsto (\mathsf{ba}\backslash w)\ \langle\boldsymbol{\lambda d}\textbf{:(nd}\ \boldsymbol{p}\textbf{)}.\ \boldsymbol{H}\ \boldsymbol{d}\rangle$
$\ldots \rightarrow \nu w{\in}\text{CLOSED}.\ (\mathsf{ba}\backslash w)\ \langle\boldsymbol{\lambda h}\textbf{:(comb}\ \boldsymbol{p}\textbf{)}.\ \boldsymbol{h}\rangle$
$\ldots \rightarrow \nu w{\in}\text{CLOSED}.\ \langle\textbf{MP (MP S K) K}\rangle$

Example 5.4.4 shows the same sample execution from Example 4.5.4 except we now preface our execution under a weakening declaration $w\overset{\triangledown}{\in}\text{CLOSED}$ (similar to Example 5.2.4). By executing top-level expressions in CLOSED, we are guaranteed to be able to access any function as for all $\mathcal{W}$, CLOSED $\leq \mathcal{W}$. By executing top-level expressions in CLOSED we are telling the type system that we do not intend to weaken

these expressions by any parameters, which is necessary in the above example. The following expression is not well-typed:

$$\nu w{\in}\textsc{open}.\ (\mathsf{nd2hil}\backslash w)\ (\text{fn}\ \cdot)\ \langle \mathbf{impi}\ (\boldsymbol{\lambda d}{:}(\mathbf{nd}\ \boldsymbol{p}).\ \boldsymbol{d})\rangle$$

Type checking on this expression fails on "fn $\cdot$" as the empty function is not a total function over $\langle(\mathbf{nd}\ \boldsymbol{A})^{\#}\rangle$ weakenable by parameters in OPEN. However, the same function is total when we consider it weakenable by parameters in CLOSED as there are none. The coverage of the empty function in CLOSED follows by the rule $\mathsf{coverEmpty}^{\#}$ of Section 5.3.

5.4.3 HOAS vs. de Bruijn with Proofs

In Section 4.6 we presented functions to translate between encodings of λ-expressions. We use **exp** from Example 2.1.2 as our encoding utilizing HOAS. Example 2.2.5 gives our first-order de Bruijn encoding where the context Φ of free variables is represented as a natural number **nat** (Example 2.1.1); variables in Φ are represented as objects of type **variable** $\ulcorner\ \Phi\ \urcorner$; and λ-expressions over Φ are represented as objects of type **term** $\ulcorner\ \Phi\ \urcorner$.

Additionally, in Chapter 4, we used dependent types and made our functions return a proof that the translation is correct utilizing an *equivalence* judgment represented using the **equiv** type family from Example 2.2.6. In this section we extend our functions with world information such that the type system now guarantees that functions are covered. Additionally, we discuss the ordering necessary to conclude that the functions also terminate and hence are total. The existence of total functions in both directions allow us to conclude that for any λ-expression encoded in one representation, there exists an *equivalent* λ-expression in the other representation.

Converting HOAS to de Bruijn

Example 5.4.5 (HOAS to de Bruijn)**.**

$$
\begin{array}{l}
\mu\mathsf{toDebruijn} \in \nabla\textsc{open}. \\
\qquad (\forall \boldsymbol{P}{\in}\mathbf{nat}.\ (\forall \boldsymbol{E}{\in}\mathbf{exp}^{\#}.\ \exists \boldsymbol{X}{\in}(\mathbf{variable}\ \boldsymbol{P}).\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ (\mathbf{var'}\ \boldsymbol{X})\rangle) \\
\qquad \supset \forall \boldsymbol{E}{\in}\mathbf{exp}.\ \exists \boldsymbol{T}{\in}(\mathbf{term}\ \boldsymbol{P}).\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ \boldsymbol{T}\rangle). \\
\quad \nu u{\in}\textsc{open}. \\
\quad \text{fn}\ \boldsymbol{P} \mapsto \text{fn}\ W \mapsto \\
\qquad \text{fn}\ \langle \mathbf{lam}\ (\boldsymbol{\lambda x}.\ \boldsymbol{E}\ \boldsymbol{x})\rangle \ \mapsto \\
\qquad\quad \text{let}\ W' = (\text{fn}\ \langle \boldsymbol{x}^{\#}\rangle \mapsto \text{let}\ (\langle \boldsymbol{Y}\rangle,\ \langle \boldsymbol{D}\rangle) = W\ \langle \boldsymbol{x}\rangle\ \text{in}\ (\langle \mathbf{succ}\ \boldsymbol{Y}\rangle, \langle \mathbf{eqVar}\ \boldsymbol{D}\rangle)) \\
\qquad\qquad \text{in}\ (\text{case}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \nu \boldsymbol{d}{\in}(\mathbf{equiv}\ (\mathbf{s}\ \boldsymbol{P})\ \boldsymbol{x}\ (\mathbf{var'}\ \mathbf{one}))^{\#}. \\
\qquad\qquad\qquad\qquad (\mathsf{toDebruijn}\backslash u)\ (\mathbf{s}\ \boldsymbol{P})\ (W'\ \text{with}\ \langle \boldsymbol{x}\rangle \mapsto (\langle \mathbf{one}\rangle,\ \langle \boldsymbol{d}\rangle))\ \langle \boldsymbol{E}\ \boldsymbol{x}\rangle) \\
\qquad\qquad\qquad\quad \text{of}\ (\nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\langle \boldsymbol{T}\rangle,\ \langle \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d}\rangle)) \ \mapsto \\
\qquad\qquad\qquad\qquad (\langle \mathbf{lam'}\ \boldsymbol{T}\rangle,\ \langle \mathbf{eqLam}\ (\boldsymbol{\lambda x}.\ \boldsymbol{\lambda d}.\ \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d})\rangle)) \\
\\
\qquad \mid\ \langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2}\rangle \ \mapsto \\
\qquad\quad \text{let}\ (\langle \boldsymbol{T_1}\rangle,\ \langle \boldsymbol{D_1}\rangle) = (\mathsf{toDebruijn}\backslash u)\ \boldsymbol{P}\ W\ \langle \boldsymbol{E_1}\rangle\ \text{in} \\
\qquad\quad \text{let}\ (\langle \boldsymbol{T_2}\rangle,\ \langle \boldsymbol{D_2}\rangle) = (\mathsf{toDebruijn}\backslash u)\ \boldsymbol{P}\ W\ \langle \boldsymbol{E_2}\rangle\ \text{in} \\
\qquad\qquad (\langle \mathbf{app'}\ \boldsymbol{T_1}\ \boldsymbol{T_2}\rangle,\ \langle \mathbf{eqApp}\ \boldsymbol{D_1}\ \boldsymbol{D_2}\rangle) \\
\\
\qquad \mid\ \langle \boldsymbol{x}^{\#}\rangle \ \mapsto \\
\qquad\quad \text{let}\ (\langle \boldsymbol{Y}\rangle,\ \langle \boldsymbol{D}\rangle) = W\ \langle \boldsymbol{x}\rangle\ \text{in} \\
\qquad\qquad (\langle \mathbf{var'}\ \boldsymbol{Y}\rangle,\ \langle \boldsymbol{D}\rangle)
\end{array}
$$

Example 5.4.5 is a straightforward conversion from Example 4.6.1 where the function is declared to be weakenable over OPEN; we add ∇OPEN to the type, we add $\nu u{\in}$OPEN to the body, and we replace every occurrence of toDebruijn with (toDebruijn$\backslash u$). The function is covered as it handles all compatible constants in the signature as well as compatible parameters in OPEN. Additionally, the function terminates by just looking at the argument of type **exp**, which gets smaller at each recursive call.

Converting de Bruijn to HOAS

Example 5.4.6 (de Bruijn to HOAS)**.**

$$
\begin{array}{l}
\mu\mathsf{toHOAS} \in \nabla\textsc{open}. \\
\qquad (\forall \boldsymbol{P}{\in}\mathbf{nat}.\ (\forall \boldsymbol{X}{\in}(\mathbf{variable}\ \boldsymbol{P}).\ \exists \boldsymbol{E}{\in}\mathbf{exp}.\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ (\mathbf{var'}\ \boldsymbol{X})\rangle) \\
\qquad \supset \forall \boldsymbol{T}{\in}(\mathbf{term}\ \boldsymbol{P}).\ \exists \boldsymbol{E}{\in}\mathbf{exp}.\ \langle \mathbf{equiv}\ \boldsymbol{P}\ \boldsymbol{E}\ \boldsymbol{T}\rangle). \\
\ \nu u{\in}\textsc{open}. \\
\quad \text{fn}\ \boldsymbol{P} \mapsto \text{fn}\ W \mapsto \\
\qquad \text{fn}\ \ \langle \mathbf{lam'}\ \boldsymbol{T}\rangle\ \ \mapsto \\
\qquad\quad (\text{case}\ (\nu \boldsymbol{x}{\in}\mathbf{exp}^{\#}.\ \nu \boldsymbol{d}{\in}(\mathbf{equiv}\ (\mathbf{s}\ \boldsymbol{P})\ \boldsymbol{x}\ (\mathbf{var'}\ \mathbf{one}))^{\#}. \\
\qquad\qquad (\mathsf{toHOAS}\backslash u) \\
\qquad\qquad\quad (\mathbf{s}\ \boldsymbol{P}) \\
\qquad\qquad\quad ((\text{fn}\ \ \nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\ \langle \mathbf{one}\rangle\ \mapsto\ (\langle \boldsymbol{x}\rangle,\ \langle \boldsymbol{d}\rangle)\ ) \\
\qquad\qquad\qquad\ \mid\ \nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\ \langle \mathbf{succ}\ \boldsymbol{X}\rangle \mapsto \\
\qquad\qquad\qquad\qquad (\text{let}\ (\langle \boldsymbol{E}\rangle,\ \langle \boldsymbol{D}\rangle) = W\ \langle \boldsymbol{X}\rangle \\
\qquad\qquad\qquad\qquad\ \text{in}\ \nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\langle \boldsymbol{E}\rangle,\ \langle \mathbf{eqVar}\ \boldsymbol{D}\rangle))\ \backslash \boldsymbol{x}\backslash \boldsymbol{d}\ )) \\
\qquad\qquad\qquad\ \backslash \boldsymbol{x}\backslash \boldsymbol{d}) \\
\qquad\qquad\quad \langle \boldsymbol{T}\rangle \\
\qquad\qquad) \\
\qquad\qquad\ \text{of}\ (\nu \boldsymbol{x}.\ \nu \boldsymbol{d}.\ (\langle \boldsymbol{E}\ \boldsymbol{x}\rangle,\ \langle \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d}\rangle))\ \ \mapsto \\
\qquad\qquad\qquad\qquad (\langle \mathbf{lam}\ (\boldsymbol{\lambda x}.\ \boldsymbol{E}\ \boldsymbol{x})\rangle,\ \langle \mathbf{eqLam}\ (\boldsymbol{\lambda x}.\ \boldsymbol{\lambda d}.\ \boldsymbol{D}\ \boldsymbol{x}\ \boldsymbol{d})\rangle)))
\end{array}
$$

$$
\begin{array}{l}
\qquad\ \mid\ \langle \mathbf{app'}\ \boldsymbol{T_1}\ \boldsymbol{T_2}\rangle\ \mapsto \\
\qquad\qquad \text{let}\ (\langle \boldsymbol{E_1}\rangle,\ \langle \boldsymbol{D_1}\rangle) = (\mathsf{toHOAS}\backslash u)\ \boldsymbol{P}\ W\ \langle \boldsymbol{T_1}\rangle\ \text{in} \\
\qquad\qquad \text{let}\ (\langle \boldsymbol{E_2}\rangle,\ \langle \boldsymbol{D_2}\rangle) = (\mathsf{toHOAS}\backslash u)\ \boldsymbol{P}\ W\ \langle \boldsymbol{T_2}\rangle\ \text{in} \\
\qquad\qquad\quad (\langle \mathbf{app}\ \boldsymbol{E_1}\ \boldsymbol{E_2}\rangle,\ \langle \mathbf{eqApp}\ \boldsymbol{D_1}\ \boldsymbol{D_2}\rangle) \\
\qquad\ \mid\ \langle \mathbf{var'}\ \boldsymbol{X}\rangle\ \mapsto\ W\ \langle \boldsymbol{X}\rangle
\end{array}
$$

Example 5.4.6 is a straightforward conversion from Example 4.6.2 where the function is declared to be weakenable over OPEN; we add ∇OPEN to the type, we add $\nu u{\in}$OPEN to the body, and we replace every occurrence of toHOAS with (toHOAS$\backslash u$). The function is covered as it handles all compatible constants in the signature and there are no compatible parameters in OPEN. Although case analysis does not need to consider parameters, the function must still be declared in this world as we recursively execute the function under new parameters. The function terminates by just looking at the argument of type **term** $\boldsymbol{P}$, which gets smaller at each recursive call.

5.5 Type Safety

A rigorous proof of type safety is presented in Appendix B. The proof required quite a number of auxiliary lemmas, which mostly established properties of substitutions (including unifiers). Additionally, as we have dependent types, many properties on types were also needed. We will start by presenting some basic properties, especially with respect to types. Second, we will present some useful properties of worlds. We will then give the main properties with respect to weakenings, strengthenings, and substitutions. We will finally discuss type preservation and progress which allows us to conclude that Delphin is a type safe language.

5.5.1 LF and Basic Properties on Types

Lemma 5.5.1 (LF-Level Substitution Composition is Well-Typed)**.**
If (1) $\mathbf{\Gamma}\ \mathbf{ctx_{lf}}$ *and (2)* $\mathbf{\Gamma'} \vdash^{\mathbf{lf}} \boldsymbol{\gamma} : \mathbf{\Gamma}$ *and (3)* $\mathbf{\Gamma_2} \vdash^{\mathbf{lf}} \boldsymbol{\gamma_2} : \mathbf{\Gamma'}$*, then* $\mathbf{\Gamma_2} \vdash^{\mathbf{lf}} (\boldsymbol{\gamma} \circ^{\mathbf{lf}} \boldsymbol{\gamma_2}) : \mathbf{\Gamma}$.

Proof. By induction lexicographically on $\mathcal{E} :: \mathbf{\Gamma'} \vdash^{\mathbf{lf}} \boldsymbol{\gamma} : \mathbf{\Gamma}$ and $\mathcal{F} :: \mathbf{\Gamma_2} \vdash^{\mathbf{lf}} \boldsymbol{\gamma_2} : \mathbf{\Gamma'}$. A detailed proof is provided in the appendix as Lemma B.1.17. We give this lemma just to highlight that the use of $\uparrow_{\boldsymbol{x}}$ is needed for this property to hold and this property does not hold in the alternative formulation of substitutions (Harper and Pfenning 2005), as discussed in Section 3.4.2. □

Lemma 5.5.2 (Computation Level Doesn't Influence Well-Formedness of Types)**.**
We show that computation-level constructs do not have any influence on the well-formedness of types. We present these two lemmas together, but they do not need to be mutually recursive.

A. If $\Omega_1, u' {\in} \tau', \Omega_2 \vdash \delta\ \mathsf{wff}$*, then* $\Omega_1, \Omega_2 \vdash \delta\ \mathsf{wff}$.

B. If $\Omega_1, u' \overset{\triangledown}{\in} \mathcal{W}', \Omega_2 \vdash \delta\ \mathsf{wff}$*, then* $\Omega_1, \Omega_2 \vdash \delta\ \mathsf{wff}$.

Proof. By induction on $\mathcal{E} :: \Omega \vdash \delta\ \mathsf{wff}$. A detailed proof is provided in the appendix as Lemma B.3.3. □

Lemma 5.5.3 (Casting Substitution to LF)**.**
If $\Omega' \vdash \omega : \Omega$*, then* $\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega\| : \|\Omega\|$.

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$. A detailed proof is provided in the appendix as Lemma B.4.7. □

Lemma 5.5.4 (id Substitution is Well-Typed)**.**
If $\Omega\ \mathsf{ctx}$*, then* $\Omega \vdash \mathrm{id}_\Omega : \Omega$.

Proof. By induction on Ω. A detailed proof is provided in the appendix as Lemma B.5.5. □

Lemma 5.5.5 (Substitution Preserves Well-Formedness of Types)**.**
If (1) $\Omega \vdash \delta\ \mathsf{wff}$ *and (2)* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash \delta[\omega]\ \mathsf{wff}$.

Proof. By induction on $\mathcal{E} :: \Omega \vdash \delta\ \mathsf{wff}$. A detailed proof is provided in the appendix as Lemma B.6.1. □

Lemma 5.5.6 (Context Weakening over Well-Formedness with Worlds)**.** *The well-formedness of types holds under* any *extensions to the context.*
If (1) $\Omega \vdash \delta\ \mathsf{wff}$ *and (2)* $(\Omega, \Omega')\ \mathsf{ctx}$*, then* $\Omega, \Omega' \vdash \delta\ \mathsf{wff}$.

Proof. This follows from Lemma B.7.1 of the appendix, noting that $\Omega \leq_* (\Omega, \Omega')$. This property does not hold over expressions. □

5.5.2 World Properties

We will present properties with respect to worlds. One important technicality is that the appendix utilizes an alternate world inclusion judgment $(\Omega, \boldsymbol{A}) \in_{\mathrm{alt}} \mathcal{W}$ which is

more complicated than our usual one $(\Omega, \boldsymbol{A}) \in \mathcal{W}$, but it is easier to prove properties about. We prove that these two judgments are equivalent in Lemmas B.9.1 and B.9.2.

Lemma 5.5.7 (Substitution Property of World Inclusion)**.**
If (1) $(\Omega, \boldsymbol{A}) \in \mathcal{W}$ *and (2)* $\mathcal{W}$ world *and (3)* $\Omega_2 \vdash \omega_2 : \Omega$*, then* $(\Omega_2, \boldsymbol{A}[\omega]) \in \mathcal{W}$*.*

Proof. A detailed proof is provided in the appendix as Lemma B.12.6. □

Lemma 5.5.8 (World Subsumption Preserves Inclusion)**.**
If (1) $(\Omega, \boldsymbol{A}) \in \mathcal{W}_1$ *and (2)* $\mathcal{W}_1 \leq \mathcal{W}_2$ *and (3)* $\mathcal{W}_2$ world*, then* $(\Omega, \boldsymbol{A}) \in \mathcal{W}_2$*.*

Proof. A detailed proof is provided in the appendix as Lemma B.13.2. □

Lemma 5.5.9 (Transitivity of World Subsumption)**.**
If (1) $\mathcal{W}_1 \leq \mathcal{W}_2$ *and (2)* $\mathcal{W}_2 \leq \mathcal{W}_3$ *and (3)* $\mathcal{W}_3$ world*, then* $\mathcal{W}_1 \leq \mathcal{W}_3$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \mathcal{W}_1 \leq \mathcal{W}_2$ and $\mathcal{F} :: \mathcal{W}_2 \leq \mathcal{W}_3$. A detailed proof is provided in the appendix as Lemma B.13.3. □

5.5.3 Weakenings

Lemma 5.5.10 (Context Weakening over Coverage without Worlds)**.**
If (1) $\Omega \vdash \overline{c}$ covers τ *and (2)* Ω_2 *closed and (3)* (Ω, Ω_2) ctx*, then* $\Omega, \Omega_2 \vdash \overline{c}$ covers τ*.*

Proof. Follows directly from a more generalized version of this lemma in the appendix, Lemma B.14.2. □

Lemma 5.5.11 (Context Weakening over Coverage with Worlds)**.**
If (1) $\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash \overline{c}$ covers τ *and (2)* $\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash \mathcal{W}$ *supports* Ω_2
and (3) $(\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2)$ ctx*, then* $\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash \overline{c}$ covers τ*.*

Proof. Follows directly from a more generalized version of this lemma in the appendix, Lemma B.14.3. □

Lemma 5.5.12 (Context Weakening over Typing without Worlds)**.**
If Ω_2 *closed and* (Ω, Ω_2) ctx*, then*

- *if* $\Omega \vdash e \in \delta$*, then* $\Omega, \Omega_2 \vdash e \in \delta$*.*
- *if* $\Omega \vdash c \in \tau$*, then* $\Omega, \Omega_2 \vdash c \in \tau$*.*
- *if* $\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$*, then* $(\Omega, \Omega_2) \vdash \mathrm{id}_{(\Omega,\Omega_2)}, \overline{f}/\overline{\alpha} : (\Omega, \Omega_2), \overline{\alpha \in \delta}$*.*

Proof. Follows directly from a more generalized version of this lemma in the appendix, Lemma B.14.4. □

Lemma 5.5.13 (Context Weakening over Typing with Worlds)**.**
If $(\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2)$ ctx *and* $\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash \mathcal{W}$ *supports* Ω_2*, then*

- *if* $\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash e \in \delta$*, then* $\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash e \in \delta$*.*
- *if* $\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash c \in \tau$*, then* $\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash c \in \tau$*.*
- *if* $\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash \mathrm{id}_{\Omega, u \overset{\triangledown}{\in} \mathcal{W}}, \overline{f}/\overline{\alpha} : \Omega, u \overset{\triangledown}{\in} \mathcal{W}, \overline{\alpha \in \delta}$*,*
  then $\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash \mathrm{id}_{\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2}, \overline{f}/\overline{\alpha} : \Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2, \overline{\alpha \in \delta}$*.*

Proof. Follows directly from a more generalized version of this lemma in the appendix, Lemma B.14.5. □

5.5.4 Strengthenings

Lemma 5.5.14 (Strengthening of World in Coverage)**.**
If (1) $\mathcal{W}_0 \leq \mathcal{W}$ *and (2)* $\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash \overline{c}$ covers τ *and (3)* $(\Omega, u \overset{\triangledown}{\in} \mathcal{W}_0, \Omega_2)$ ctx*,*
then $\Omega, u \overset{\triangledown}{\in} \mathcal{W}_0, \Omega_2 \vdash \overline{c}$ covers τ*.*

Proof. By induction on $\mathcal{E} : \Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash \overline{c}$ covers τ. A detailed proof is provided in the appendix as Lemma B.15.9. □

Lemma 5.5.15 (Strengthening of World in Typing)**.**

If $\mathcal{W}_0$ world *and* $\mathcal{W}_0 \leq \mathcal{W}$*, then*

- *if* $\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e \in \delta$*, then* $\Omega, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_2 \vdash e \in \delta$*.*
- *if* $\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash c \in \tau$*, then* $\Omega, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_2 \vdash c \in \tau$*.*
- *if* $\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash \mathrm{id}_{\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2}, \overline{f}/\overline{\alpha} : \Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \overline{\alpha\in\delta}$*,*
  then $\Omega, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_2 \vdash \mathrm{id}_{\Omega, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_2}, \overline{f}/\overline{\alpha} : \Omega, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_2, \overline{\alpha\in\delta}$*.*

Proof. By induction on e and c and $\overline{f}$. A detailed proof is provided in the appendix as Lemma B.15.10. Notice that there are two cases each for $e\backslash\boldsymbol{x}$, $c\backslash\boldsymbol{x}$, and u'. There are three cases for $e\backslash u'$. □

5.5.5 Substitutions

Lemma 5.5.16 (Main Substitution Property over Typing)**.**

If $\Omega' \vdash \omega : \Omega$*, then*

- *if* $\Omega \vdash e \in \delta$*, then* $\Omega' \vdash e[\omega] \in \delta[\omega]$*.*
- *if* $\Omega \vdash c \in \tau$*, then* $\Omega' \vdash c[\omega] \in \tau[\omega]$*.*
- *if* $(\Omega, \overline{\alpha\in\delta})$ ctx *and* $\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha\in\delta}$*,*
  then $\Omega' \vdash \mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha} : \Omega', \overline{\alpha\in\delta}[\omega]$*.*

Proof. By induction on e and c and $\overline{f}$. A detailed proof is provided in the appendix as Lemma B.17.7. For illustration, we summarize two interesting cases:

Case: $\boldsymbol{M}$ and by inversion: $\mathcal{E} = \dfrac{\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\,\mathsf{isLF}$

$\Omega' \vdash \omega : \Omega$ by assumption
$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega\| : \|\Omega\|$ by Lemma 5.5.3 (Casting)
$\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{M}[\|\omega\|] : \boldsymbol{A}[\|\omega\|]$ by LF Substitution Lemma (see appendix)
$\Omega' \vdash \boldsymbol{M}[\omega] \in \boldsymbol{A}[\omega]$ by isLF and Subst. Application

Case: $(e\backslash\boldsymbol{x})$ and by inversion: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash e \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \qquad \Omega_2 \text{ closed}\end{array}}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$\Omega', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2' \vdash \omega : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$
and $\Omega', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$
and $(\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x})$ is a subterm of ω
and Ω_2' closed by assumption and inversion
A proof that ω has this property is formalized in the appendix.
$\Omega' \vdash \omega_1 : \Omega$ by inversion using tpSubIndNew
$\Omega' \vdash e[\omega_1] \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}[\omega_1].\ \tau[\uparrow_{\boldsymbol{x'}} \omega_1, \boldsymbol{x'}/\boldsymbol{x'}]$
by induction hypothesis on e with $\mathcal{E}_1$ and Subst. Application
$\Omega', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2' \vdash e[\omega_1]\backslash\boldsymbol{x_0} \in \tau[\uparrow_{\boldsymbol{x'}} \omega_1, \boldsymbol{x'}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega'}, \boldsymbol{x_0}/\boldsymbol{x'}]$ by pop
$\Omega', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2' \vdash e[\omega_1]\backslash\boldsymbol{x_0} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}]$ by subst. props.
$\Omega', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2' \vdash e[\omega]\backslash\boldsymbol{x}[\omega] \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}][\omega]$ by subst. weakenings
$\Omega', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2' \vdash (e\backslash\boldsymbol{x})[\omega] \in (\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x'}])[\omega]$ by Subst. Application

□

Corollary 5.5.17 (Substitution Composition is Well-Typed)**.**

If Ω ctx *and* $\Omega' \vdash \omega : \Omega$ *and* $\Omega_2 \vdash \omega_2 : \Omega'$*, then* $\Omega_2 \vdash (\omega \circ \omega_2) : \Omega$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$ and $\mathcal{F} :: \Omega_2 : \omega_2 : \Omega'$. A detailed proof is provided in the appendix as Lemma B.18.1. □

Lemma 5.5.18 (Equality of Substitution Composition applied to Types)**.**

If $\Omega_1 \vdash \omega_1 : \Omega_0$ *and* $\Omega_2 \vdash \omega_2 : \Omega_1$ *and* $\Omega_0 \vdash \delta$ wff*, then* $\delta[\omega_1][\omega_2] = \delta(\omega_1 \circ \omega_2)$*.*

Proof. By induction on $\mathcal{E} :: \Omega_0 \vdash \delta$ wff. The existence of $(\omega_1 \circ \omega_2)$ is established by Corollary 5.5.17 and the details of this induction can be found in the appendix, Lemma B.12.1. □

5.5.6 Type Preservation

Theorem 5.5.19 (Type Preservation)**.**

- *If* $\Omega \vdash e \in \delta$ *and* $\Omega \vdash e \to f$*, then* $\Omega \vdash f \in \delta$*.*
- *If* $\Omega \vdash c \in \tau$ *and* $\Omega \vdash c \to c'$*, then* $\Omega \vdash c' \in \tau$*.*
- *If* $\Omega \vdash c \in \tau$ *and* $\Omega \vdash c \xrightarrow{*} c'$*, then* $\Omega \vdash c' \in \tau$*.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash e \to f$ and $\mathcal{E} :: \Omega \vdash c \to c'$ and $\mathcal{E} :: \Omega \vdash c \xrightarrow{*} c'$. A detailed proof is provided in the appendix as Theorem B.19.4, but we outline one case for illustration:

Case: $\mathcal{E} = \dfrac{}{\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash (\nu u' \in \mathcal{W}_2.\ e) \backslash u \to e[\uparrow_u \mathrm{id}_\Omega, u/u']}$

$\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash (\nu u' \in \mathcal{W}_2.\ e) \backslash u \in \tau[\uparrow_u \mathrm{id}_\Omega, u/u']$	by assumption and inversion
$\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash \mathcal{W}_2$ supports Ω_2	by inversion using popW
$\mathcal{W} \leq \mathcal{W}_2$	by inversion using popW
$\Omega \vdash \nu u' \in \mathcal{W}_2.\ e \in \nabla \mathcal{W}_2.\ \tau$	by inversion using popW
$\Omega, u' \overset{\triangledown}{\in} \mathcal{W}_2 \vdash e \in \tau$	by inversion using newW
Ω ctx and $\mathcal{W}$ world and $(\Omega, u \overset{\triangledown}{\in} \mathcal{W}_2, \Omega_2)$ ctx	by well-formedness properties
$\Omega, u \overset{\triangledown}{\in} \mathcal{W}_2 \vdash \mathcal{W}_2$ supports Ω_2	by properties of supports
$\Omega, u \overset{\triangledown}{\in} \mathcal{W}_2 \vdash (\uparrow_u \mathrm{id}_\Omega, u/u') : \Omega, u' \overset{\triangledown}{\in} \mathcal{W}_2$	by Lemma 5.5.4 and subst. typing
$\Omega, u \overset{\triangledown}{\in} \mathcal{W}_2 \vdash e[\uparrow_u \mathrm{id}_\Omega, u/u'] \in \tau[\uparrow_u \mathrm{id}_\Omega, u/u']$	by Lemma 5.5.16 (Substitutions)
$\Omega, u \overset{\triangledown}{\in} \mathcal{W}_2, \Omega_2 \vdash e[\uparrow_u \mathrm{id}_\Omega, u/u'] \in \tau[\uparrow_u \mathrm{id}_\Omega, u/u']$	by Lemma 5.5.13 (Weakening)
$\Omega, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash e[\uparrow_u \mathrm{id}_\Omega, u/u'] \in \tau[\uparrow_u \mathrm{id}_\Omega, u/u']$	by Lemma 5.5.15 (Strengthening)

The last step of this case illustrates the admissibility of world subsumption that we built into popW.

□

Corollary 5.5.20 (Soundness)**.** *Parameters cannot escape their scope. If* $\Omega \vdash e \in \tau$ *and* $\Omega \vdash e \to e'$*, then all parameters in* e *and* e' *are declared in* Ω*.*

Proof. Follows directly from type preservation (Theorem 5.5.19) noting that typing in a context Ω guarantees that all encountered parameters are in Ω (easily proved by induction on the typing rules). □

5.5.7 Progress

Lemma 5.5.21 (Liveness Property)**.**

If

(1) Ω *does not contain any declarations of the form* $\alpha{\in}\delta$

(2) $\Omega \vdash \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ \forall\overline{\alpha{\in}\delta}.\ \tau$

(3) and $\Omega, \mathbf{\Gamma} \vdash \mathrm{id}_{(\Omega,\mathbf{\Gamma})}, \overline{v}/\overline{\alpha} : \Omega, \mathbf{\Gamma}, \overline{\alpha{\in}\delta}$,

then there exists a c_i *in* $\overline{c}$ *such that* $\Omega \vdash c_i \xrightarrow{*} \nu\mathbf{\Gamma}.\ (\overline{v} \mapsto f)$.

Proof. By induction on $\mathcal{E} :: \Omega \vdash \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ \forall\overline{\alpha{\in}\delta}.\ \tau$. A detailed proof is provided in the appendix as Lemma B.20.4. Condition (1) establishes that the context does not contain any pattern-variable declarations, which is an invariant when we execute programs. Condition (2) establishes that the cases are covered and condition (3) gives us the list of arguments (which are values) being applied to the function. Finally, the result stipulates that some case in $\overline{c}$ will match. For example, if $\Omega \vdash \overline{c}$ covers $\forall\alpha{\in}\delta.\ \tau$, then condition (3) specifies that we have a value v such that $\Omega \vdash v \in \delta$ and this lemma tells us that there exists a c_i that matches, i.e. $\Omega \vdash c_i \xrightarrow{*} (v \mapsto f)$.

We consider a list of values as functions may range over lists. The context $\mathbf{\Gamma}$ is a generalization to handle cases of the form $\overline{c}\backslash\boldsymbol{x}$, which corresponds to a use of coverPop. We will now show a specialized version of coverPop to illustrate what is happening. The actual case (in the appendix) needs to deal with renaming substitutions, which we abstract away from for clarity:

Case:

$$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Omega \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ (\nabla \boldsymbol{\Gamma}.\ \forall \overline{\alpha \in \delta}.\ \tau) \qquad \Omega_2 \text{ closed}}}{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c} \backslash \boldsymbol{x} \text{ covers } \nabla \boldsymbol{\Gamma}.\ \forall \overline{\alpha \in \delta}.\ \tau}\ \mathsf{coverPop}$$

$\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2$ does not contain any decs. of $\alpha^* \in \delta^*$ — by assumption

$\Omega_2 = \cdot$ — by above with premise Ω_2 closed

$\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \boldsymbol{\Gamma} \vdash \mathrm{id}_{(\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \boldsymbol{\Gamma})}, \overline{v}/\overline{\alpha} : \Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \boldsymbol{\Gamma}, \overline{\alpha \in \delta}$ — by assumption (notice that this relies on $\Omega_2 = \cdot$)

There exists a c_i in $\overline{c}$ such that

$\Omega \vdash c_i \xrightarrow{*} \nu \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}.\ \nu \boldsymbol{\Gamma}.\ (\overline{v} \mapsto f)$ — by induction hypothesis on $\mathcal{E}_1$

$\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash c_i \backslash \boldsymbol{x} \xrightarrow{*} (\nu \boldsymbol{\Gamma}.\ (\overline{v} \mapsto f))$ — by Operational Semantics

$\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash c_i \backslash \boldsymbol{x} \xrightarrow{*} \nu \boldsymbol{\Gamma}.\ (\overline{v} \mapsto f)$ — Since $\Omega_2 = \cdot$

When we execute programs, the contexts will not contain any pattern-variable declarations. Therefore, the Ω_2 closed condition guarantees that Ω_2 is empty during execution, which we see was crucial as we would not be able to appeal to the induction hypothesis if the values $\overline{v}$ make sense in a context extended by a nonempty Ω_2. This highlights in the actual proofs that the construct $c\backslash\boldsymbol{x}$ (and $e\backslash\boldsymbol{x}$) must refer to the $\boldsymbol{x}$ which is the last (or newest) parameter.

□

Theorem 5.5.22 (Progress)**.**

If Ω does not contain any decs. of $\alpha \in \delta$, then

- *if $\Omega \vdash e \in \delta$ and e is not a value, then $\Omega \vdash e \to f$.*
- *if $\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$, then for all e_i in $\overline{e}$ such that e_i is not a value, $\Omega \vdash e_i \to e_i'$.*

Proof. By induction on e and $\overline{f}$. A detailed proof is provided in the appendix as Lemma B.21.1. This theorem is straightforward given liveness (Lemma 5.5.21). □

5.5.8 Putting it Together

Theorem 5.5.23 (Type Safety)**.**

Delphin is a type safe language, i.e. if Ω *does not contain any decs. of* $\alpha{\in}\delta$ *and* $\Omega \vdash e \in \delta$ *and* e *is not a value, then there exists an* f *such that* $\Omega \vdash e \rightarrow f$ *and* $\Omega \vdash f \in \delta$*.*

Proof. Follows directly from type preservation (Theorem 5.5.19) and progress (Theorem 5.5.22). □

5.6 Conclusion

In this chapter we have extended the core Delphin system from Chapter 4 to guarantee programs never get stuck and conclude that Delphin is a type safe language when all functions are covered. To determine when functions are covered we needed to distinguish functions by the type of parameters they support, which we call their worlds.

Furthermore, we extended our dependently-typed examples from Chapter 4 with world information and discussed how the same functions can also be viewed as proofs of various properties. This justifies how the Delphin type system allows one to manipulate and reason about complex data while utilizing HOAS to abstract away from mundane details of representing and manipulating first-order representations of variables.

This concludes our theoretical development. Next, in Chapter 6 we present the actual Delphin programming language which has been a large implementation effort for the past three years utilizing this underlying type theory to allow one to write programs and express proofs using LF representations. Afterwards, we will present related work in Chapter 7 and conclude in Chapter 8.

Chapter 6

Implementation

The previous chapters formed the core type system and meta-theory for the Delphin programming language. This chapter aims to serve as a user manual for the system, which is available on the Delphin website (`http://www.delphin.logosphere.org/`). Additionally, we will summarize some of the challenges with respect to the implementation. The main implementation efforts focused on the development of a unification algorithm, a type reconstructor, a type checker, an evaluator, a coverage checker, and a termination checker. The current version of Delphin is 1.5.1 and consists of approximately 18,000 lines of Standard ML code not including code adopted from the Twelf system (Pfenning and Schürmann 1998) to handle LF specific operations.

A very short introduction to Delphin is also available as *System Description: Delphin – A Functional Programming Language for Deductive Systems* (Poswolsky and Schürmann 2008). We will show the concrete (ASCII) code for all the examples throughout this dissertation, which have been thoroughly discussed when they were introduced. We will motivate the features of Delphin through examples.

6.1 Basics

Delphin is a functional programming language built to facilitate the encoding of, manipulation of, and reasoning over dependent higher-order datatypes. Delphin is a two level system that separates representation-level functions from computation-level functions. The representation level is the logical framework LF (Harper et al. 1993), which supports dependent types and higher-order abstract syntax (HOAS). The computation level provides mechanisms such as case analysis and recursion to allow for the manipulation of data. Delphin's most novel feature is in its support of the dynamic creation of parameters (i.e. scoped constants).

Delphin is first and foremost a general purpose programming language supporting complex data structures. However, it also contains strong tools for meta-reasoning allowing one to determine if functions are total, and hence Delphin can also be used to formalize proofs. Thus, it is well-suited to be used in the setting of the Logosphere project (Schürmann et al. 2008), a digital library of formal proofs that brings together different proof assistants and theorem provers with the goal to facilitate the exchange of mathematical knowledge by converting proofs from one logical formalism into another. Delphin has been successfully used in expressing translations between HOL, Nuprl, and various other logics.

The reserved characters and keywords are given in Figure 6.1. Parentheses are standard and used for grouping. The programmer may place comments between `(*` and `*)`; the nesting of comments is permitted. We will use *id* and ***id*** to stand for computation-level and representation-level identifiers (or names). Identifiers are separated by whitespace and we allow reserved symbols such as as "`->`" inside identifiers. Therefore, one must use whitespace to separate identifiers. For example, "`A->B`" is a single identifier while "`A -> B`" refers to a functional type.

```
  .   ;   _   *   ,   #   \   :   =   |   @   (  )   <  >   [  ]   {  }
(*  *)   ->   <-   =>   and   case   end   fn   fun   in   left   let
      of   params   right   sig   type   unit   use   val   with
            %abbrev   %infix   %name   %postfix   %prefix
```

Figure 6.1: Reserved Characters and Keywords

We distinguish data (LF objects) from computation-level expressions by enclosing the former in `<...>`. For example, the expression `e f` refers to computation-level application whereas the application in **`<M N>`** occurs at the LF level (representation level). Similarly, expressions of type **`<A1 -> A2>`** are of the form **`<M>`** where **`M`** is an LF function.

The top-level grammar is given in Figure 6.2. We use `;` to separate top-level declarations from each other. In this chapter we will explain the grammar via examples. Datatypes are declared via `sig`. Recursive functions are defined via `fun` and mutual recursion is expressed with the `and` keyword. The `val` keyword is used to evaluate an expression and one may also just write an expression e which corresponds to `val it =` e. Computation-level type aliases can be created by using the `type` keyword. Functions are called on arguments that are constructed from constants in the signature as well as parameters (dynamically created constants). The `params` keyword is used to specify which parameters any following `fun` is expected to handle, which we also refer to as a function's world (Chapter 5). Finally, one may load Delphin files using the `use` keyword and import datatype declarations formatted for Twelf with `sig use`.

6.2 LF Syntax and Datatypes

Our representation level is the Edinburgh Logical Framework LF (Harper et al. 1993), as discussed in Chapter 2. We will quickly summarize the LF level and show how to

		Concrete Syntax	Formal (Chapter 4)
compdec	::=	*id* `:` τ `=` *e*	Explicit type annotation
	|	*id* `=` *e*	Omitted type annotation
topdec	::=	`sig` *condec* `...` *condec* `;`	Datatype Declaration
	|	`sig use "`*filename*`" ;`	Loads a Signature
	|	`fun` *compdec* `;`	Recursive Functions ($\mu u{\in}\tau.\ e$)
	|	`fun` *compdec* `and ... and` *compdec* `;`	Mutually recursive extension
	|	`val` *compdec* `;`	Named top-level expression
	|	*e* `;`	Top-level expression named `it`
	|	`use "`*filename*`" ;`	Loads a Delphin file
	|	`type` *id* `=` τ `;`	Type Alias
	|	`params =` *worldec*	World Declarations

Figure 6.2: Delphin Top-Level Grammar

declare datatypes (the signature) in Delphin.

6.2.1 LF Syntax

We adopt the same syntax for LF objects as in Twelf (Pfenning and Schürmann 1998) and we present the concrete syntax in Figure 6.3. The concrete syntax follows the formal syntax very closely where the ASCII equivalents of $\boldsymbol{\lambda x{:}A}$ and $\boldsymbol{\Pi x{:}A}$ are `[`$\boldsymbol{x:A}$`]` and {$\boldsymbol{x:A}$}. We use `->` as syntactic sugar to express functions that are not dependent. We allow the programmer to write "_" which stands for a *hole* which is filled in automatically during type reconstruction. If the *hole* cannot be uniquely filled in, an appropriate error message is displayed.

Application (juxtaposition) is left associative and has the highest precedence. After that, `->` and `<-` have the same precedence and are right and left associate, respectively.

			Concrete Syntax	Formal (Chapter 2)
LF Types	A, B	::=	*id*	a
		|	*A M*	$A\ M$
		|	{*id* : *A*} *B*	$\Pi x{:}A.\ B$
		|	{*id*} *B*	$\Pi x{:}A.\ B$, for any A
		|	*A* -> *B*	$\Pi x{:}A.\ B$, when x not in B
		|	*B* <- *A*	$\Pi x{:}A.\ B$, when x not in B
		|	_	(anything)
LF Objects	M, N	::=	*id*	x or c
		|	*M N*	$M\ N$
		|	[*id* : *A*] *N*	$\lambda x{:}A.\ N$
		|	[*id*] *B*	$\lambda x{:}A.\ N$, for any A
		|	_	(anything)
		|	*M* : *A*	(type ascription)
Kinds	K	::=	type	*type*
		|	{*id* : *A*} *K*	$\Pi x{:}A.\ K$
		|	{*id*} *B*	$\Pi x{:}A.\ K$, for any A
		|	*A* -> *K*	$\Pi x{:}A.\ K$, when x not in K

Figure 6.3: Concrete Syntax for LF

6.2.2 Declaring Datatypes in Delphin

Datatype declarations are made in Delphin as:

```
sig condec ... condec ;
```

We write *condec* to stand for constant declaration, which is mainly used to declare type constants and object constants (constructors). Additionally, we have added support for definitions (*def*) as well as setting default preferences of variable names (*name*) and the declaration of infix operators of various precedences (*prec*). The grammar for datatype declarations is given in Figure 6.4. We will describe the syntax by declaring a few datatypes that we will use throughout this chapter.

		Concrete Syntax	Description
name	::=		(No printing preference)
	|	`%name` ***id***	Pattern variable names
	|	`%name` ***id id***	Pattern vars. and parameters
prec	::=		(No precedence preference)
	|	`%infix left` *nat*	Left-associative infix
	|	`%infix right` *nat*	Right-associative infix
	|	`%postfix` *nat*	Postfix
	|	`%prefix` *nat*	Prefix
def	::=	`<`***id*** `:` ***A*** `=` ***M***`>` *prec*	Definition
	|	`<`***id*** `=` ***M***`>` *prec*	Definition
	|	`<`***id*** `:` ***A*** `=` ***M***`>` `%abbrev` *prec*	Abbreviation
	|	`<`***id*** `=` ***M***`>` `%abbrev` *prec*	Abbreviation
condec	::=	`<`***id*** `:` ***K***`>`	***a*:*K*** (Type Declaration)
	|	`<`***id*** `:` ***A***`>` *name prec*	***c*:*A*** (Constructors)
	|	*def*	Definition (i.e. sugar)

Figure 6.4: Datatype Declarations (`sig` Declarations)

```
(* Untyped λ-expressions using HOAS (Example 2.1.2) *)
sig <exp : type> %name E x
    <lam : (exp -> exp) -> exp>
    <app : exp -> exp -> exp>;
```

We represent untyped λ-expressions as objects of type **`exp`**. The first line declares the type **`exp`** and is followed by the definition of two constructors. Applications are represented using the **`app`** constructor taking two arguments while abstractions are represented utilizing HOAS as the **`lam`** constructor only takes one argument which is an LF function. This means that variables of the untyped λ-calculus are represented as LF variables of type **`exp`**. As an example, the identity function is represented as **`lam [x:exp] x`**.

The %name declaration is completely optional and specifies a default preference to the naming of anonymous variables appearing in the output. In the example above, we declared that the names of pattern variables of type **exp** should be defaultly prefixed with **E** and parameters should be prefixed with **x**.

```
(* Formulas (Example 2.2.2) *)
sig <o : type>  %name A
    <ar : o -> o -> o> %infix right 10
    <p : o>
```

We will soon turn to an example with dependent types, but we first represent a simple type of formulas containing an arrow type and a base type. The **ar** constructor is declared as a right-associative infix operator by writing %infix right 10. The number is used to allow the user to declare operators with varying precedence. This is an enhancement to simplify the input and output of code.

We now turn to a couple of dependently-typed examples. We represent derivations of **A** in a simple natural deduction calculus as objects of type **nd A**. Additionally, we represent combinators of the same formula using the type **comb A**.

```
(* Natural Deduction (Example 2.2.3) *)
sig <nd : o -> type> %name D d
    <impi : (nd A -> nd B) -> nd (A ar B)>
    <impe : nd (A ar B) -> nd A -> nd B> ;

(* Typed Combinators (AKA Hilbert-Style Calculus) (Example 2.2.4) *)
sig <comb : o -> type> %name H h
    <K : comb (A ar B ar A)>
    <S : comb ((A ar B ar C) ar (A ar B) ar A ar C)>
    <MP : comb (A ar B) -> comb A -> comb B> ;
```

Our constructors above contain free variables, which are implicitly Π-quantified. For example, the actual type of **impi** is:

```
impi : {A : o} {B : o} (nd A -> nd B) -> nd (A ar B)
```

Therefore, **impi** formally has three arguments although the declared type only takes one argument. The use of implicit Π-quantification is simply a frontend convenience. The programmer will only provide one argument to **impi** as specified, but it will be expanded out under-the-hood. The object **impi D** is automatically converted to **impi _ _ D**. The use of underscores is always safe as the type of **D** uniquely determines what the indexed object must be.

6.2.3 Definitions

As another convenience we allow for the user to define *definitions* (see *def* category in Figure 6.4). As an example, one can define **idFun = lam [x:exp] x**, which allows one to use **idFun** in their code as shorthand for the identity function. Additionally, the use of definitions will persist when Delphin outputs data, which may significantly simplify the output.

If one writes a definition that has arguments that do not occur *strict* (Pfenning and Schürmann 1998), an error message will result. Under these circumstances, one may use the optional `%abbrev` keyword to declare a definition as a frontend *abbreviation*, which means that any use of the definition is expanded out immediately during parsing. Abbreviations have no *strictness* requirements, but they don't have the benefit of appearing in the output.

6.2.4 Importing Datatypes from Twelf

Delphin allows the user to import datatype declarations from Twelf files via the top-level construct:

`sig use "`*filename*`" ;`

Delphin will open the file named *filename* and add all defined datatypes to the signature. The file is expected to be a properly formed Twelf (Pfenning and Schürmann 1998) file. This feature is merely a tool to allow one to easily move from Twelf to Delphin.

6.3 Computation-Level Expressions and Types

Delphin's most novel feature is the *new* construct, which is written as {<x>}e and has type {<x>}τ. Evaluation of e occurs while the binding `x` remains uninstantiated. Therefore, for the scope of e, the variable `x` can be thought of as a fresh constant, which we call a parameter. One may view this as providing a way to dynamically extend the signature. Expressions {<x>}e evaluate to {<x>}v, where v is a value.

Delphin pervasively distinguishes between LF types $\boldsymbol{A}$, parameter types $\boldsymbol{A}$`#`, and computation-level types τ. In the expression {<x>}e, the variable `x` has type $\boldsymbol{A}$`#`. Additionally, `x` is also an object of type $\boldsymbol{A}$, and as such $\boldsymbol{A}$`#` can be seen as a subtype of $\boldsymbol{A}$.

We write τ for computation-level types and present the grammar (with formal mappings) in Figure 6.5. Computation-level types τ include (1) a `unit` type, (2) a function type (`->`), (3) a pair type (`*`), and (4) our newness type ({<x>}τ). We also permit *id* to refer to type aliases declared via the top-level `type` keyword.

We will start by writing a function that evaluates expressions of type `exp` under λ-binders. We will consider the evalBeta function from Example 3.3.5 (and extended with worlds in Example 5.2.2).

		Concrete Syntax	Formal (Chapter 4)
dec	::=	τ	$(u{\in}\tau)$
	|	**<*id* : *A*>**	$(\boldsymbol{x}{\in}\boldsymbol{A})$
	|	**<*id* : *A*#>**	$(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$
newdec	::=	**<*id* : *A*#>**	$(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$
	|	**<*id* : *A*>**	$(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$
	|	**<*id*>**	$(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$, for some $\boldsymbol{A}$
τ, σ	::=	`unit`	unit
	|	*dec* `->` σ	$\forall\alpha{\in}\delta.\ \sigma$
	|	(*dec* `and ... and` *dec*) `->` σ	$\forall\overline{\alpha{\in}\delta}.\ \sigma$
	|	*dec* `*` σ	$\exists\alpha{\in}\delta.\ \sigma$
	|	{*newdec*} τ	$\nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$
	|	**<*A*>**	$\langle\boldsymbol{A}\rangle$
	|	**<*A*#>**	$\langle\boldsymbol{A}^{\#}\rangle$
	|	*id*	(type alias)
	|	_	(anything)

Figure 6.5: Concrete Syntax for Computation-Level Types τ

```
params = <exp>;
fun evalBeta : <exp> -> <exp>
 = fn <app E1 E2> => (case (evalBeta <E1>, evalBeta <E2>)
                      of (<lam [x] F x>, <V>) => evalBeta <F V>
                       | (<F>, <V>) => <app F V>)
    | <lam [x] E x> => (case ({<x : exp#>} evalBeta <E x>)
                          of  {<x : exp#>} <E' x> => <lam [x] E' x>)
    | <x#> => <x> ;
```

Function `evalBeta` traverses through λ-expressions and reduces β-redexes. The application case evaluates both arguments and if the first argument happens to evaluate to **<lam [x] F x>**, we do `evalBeta` **<F V>**, which continues on the body of the function (**F x**) by substituting **V** for **x**. The **<lam [x] E x>** case illustrates how we can recurse under λ-binders by creating a new parameter to which **E** may

be applied. The expression {<x : exp#>} evalBeta <E x> triggers the recursion of evalBeta in scope of a fresh parameter x. The last case <x#> defines what the function should do when it encounters one of these parameters; in this case we define parameters to just evaluate to themselves.

Since one may dynamically create arbitrary parameters, a Delphin function expects arguments that are constructed from constants in the signature as well as parameters. For example, given a function foo, the evaluation of {<x>}foo <x> would get stuck if foo does not provide a case for parameters. The params keyword is used to specify worlds, i.e. to indicate which parameters our functions are intended to handle. Note that a function may call a function with a different params specification as long as the callee can handle all the parameters of the caller. The ability to call functions that make sense with respect to different parameters is known as *world subsumption* (Section 5.1.2). With respect to evalBeta, we use the params keyword to declare it to support parameters x of type exp#. If we omitted this declaration we would get a *Match Non-Exhaustive Warning* as Delphin would not be able to determine if the function is defined for all parameters it may encounter. For example, assuming idFun = lam [x:exp] x, then the expression {<y : (exp -> exp)#>} evalBeta <y idFun> would get stuck as evalBeta does not handle a case for parameters of type (exp -> exp)#. However, since we specified that the world of evalBeta only supports parameters of type exp#, the above expression is disallowed and would generate an appropriate warning message (*World Subsumption Error*) indicating that the function evalBeta is being executed in scope of unsupported parameters.

Functions are defined by cases. We write τ_1 -> τ_2 for the type of non-dependent functions, and we write either (1) <x : *A*> -> τ or (2) <x : *A*#> -> τ when x can occur in τ. Our evalBeta example showed a function without dependent types, and

we now turn to an example of writing a translator from natural deduction derivations to combinators.

The first step in our translator is called bracket abstraction, or `ba`, which internalizes abstraction in the combinator calculus (Example 4.5.2 or Example 5.4.2). If **`M`** has type **`comb A -> comb B`**, then we can use `ba` to get a combinator **`M'`** of type **`comb (A ar B)`**. Subsequently, if we have a **`N`** of type **`comb A`**, then the object **`MP M' N`** is equivalent to **`M N`** in the combinator calculus.

```
params = <nd A>, <comb A>;
fun ba : <comb A -> comb B> -> <comb (A ar B)>
  = fn <[x] x> => <MP (MP S K) (K : comb (A ar A ar A))>
     | <[x] MP (H1 x) (H2 x)> =>
                  (case ((ba <[x] H1 x>), (ba <[x] H2 x>))
                     of (<H1'>, <H2'>) => <MP (MP S H1') H2'>)
     | <[x] H> => <MP K H> ;
```

The first line uses the `params` keyword to declare that the functions that follow are intended to handle an arbitrary collection of dynamically created (1) natural deduction derivations and (2) combinators. Functions are defined by cases, and in this example we are performing case analysis over LF functions. The first case handles the creation of *identity* combinators, i.e. of type **`A ar A`**. The second case handles **`MP`** by performing bracket abstraction on both parts and using the **`S`** combinator on the results. Notice that we perform case analysis on the results of the recursive calls since we need to extract the LF-level (representation-level) **`H1'`** from the computation-level **`<H1'>`**. The third case handles combinators **`H`** of type **`comb B`** that do not use the hypothetical combinator **`x`**; in this situation, we use the **`K`** combinator. This function indeed covers all cases. Note that the last two cases are partly overlapping; during execution, Delphin will always pick the first case that matches.

Observe that **`A`** and **`B`** occur free in the type of `ba`. The full type of `ba` is:

```
(<A : tp> and <B : tp> and <comb A -> comb B>) -> <comb (A ar B)>
```

The first two elements of the first argument are treated *implicitly*, analogous to the implicit Π-quantification we adopted in the declaration of constructors. This is a front-tend convenience and it is expanded out under-the-hood. For example, the expression `ba <[x] H x>` is automatically converted into `ba (<_> and <_> and <[x] H x>)`. The underscores are filled in automatically by the type reconstruction algorithm, which is always safe as the type of the third element uniquely determines the first two arguments. This greatly simplifies our code as it is redundant to explicitly supply an input argument which is indexed in another input argument.

The use of `and` in type declarations and applications will typically be implicit as in the above example. All free variables in types are attached using `and` to the first argument in which they appear. Although `and` will typically be implicit, it is important to note the difference between functions of type (1) `(<x : A> and` τ`) ->` σ and (2) `(<x : A> *` τ`) ->` σ. A function of type (1) is always applied to two elements while a function of type (2) takes one argument which evaluates into a pair of two elements. This distinction is only necessary because of dependent types and is discussed in Section 4.1. The `x` can occur in σ in type (1) but not in (2) as the latter would make σ dependent on the evaluation of a pair, and we only permit dependencies on data, not computation.

We have motivated our grammar with some examples and we present the actual grammar for computation-level expressions e in Figure 6.6. Application is expressed by juxtaposition, is left associative, and has the highest precedence. After application, we use `,` and `*` and `and` for pair-like constructs which are all right-associative infix operators. The symbols `@` and `\` are left-associative infix operations with the same precedence as applications. We use `@` as syntactic sugar representing LF-level (representation-level) application lifted to the computation level. This syntactic

		Concrete Syntax	Formal (Chapter 4)
letdec	::=	`val` *pat* `=` *e*	(let val)
	|	`fun` *compdec*	(let fun)
	|	`fun` *compdec* `and ... and` *compdec*	(let fun)
e, *f*	::=	*id*	$\boldsymbol{x}$ or u
	|	`<`$\boldsymbol{M}$`>`	$\boldsymbol{M}$ or $\langle \boldsymbol{M} \rangle$
	|	`()`	()
	|	*e f*	$e\ f$
	|	*e* (*f* `and ... and` *f*)	$e\ \overline{f}$
	|	*e*`,` *f*	$(e,\ f)$
	|	{*newdec*} *e*	$\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e$
	|	*e* `\x`	$e \backslash \boldsymbol{x}$
	|	`fn .`	fn $\cdot$
	|	`fn` *c* `|` ... `|` *c*	fn $\overline{c}$
	|	*e* `@` *f*	(lifted application)
	|	`let` *letdec* ... *letdec* `in` *f* `end`	(let sugar)
	|	`case` *e* `of` *c* `|` ... `|` *c*	(case sugar)
	|	`case` (*e* `and ... and` *e*) `of` *c* `|` ... `|` *c*	(case sugar)
	|	*e* `with .`	(with sugar)
	|	*e* `with` *c* `|` ... `|` *c*	(with sugar)
	|	*e* `:` τ	(type ascription)

Figure 6.6: Concrete Syntax for Computation-Level Expressions e, f

sugar is expanded out as follows:

$$e \texttt{ @ } f = \texttt{(fn <x> => fn <y> => <x y>)}\ e\ f$$

For example, `<x> @ <y>` evaluates to `<x y>`.

There is one significant difference between our theoretic development and our grammar in Figure 6.6. In the formal system, data and computation are distinguished as separate syntactic categories ($\boldsymbol{M}$ and e), but this is ambiguous in the concrete syntax; for example, *id* is used for both $\boldsymbol{x}$ and u. Therefore, our concrete grammar cleanly distinguishes data from computation by requiring *everything* that is data to be enclosed in `<...>`, which introduces an ambiguity. Formally, we used the types

$\boldsymbol{A}$ and $\langle \boldsymbol{A} \rangle$ to distinguish between (1) data and (2) computation resulting in data (see Chapters 3 and 4). However, our concrete grammar hides this distinction using **<*A*>** for both. Formally, we write $\boldsymbol{M}$ for objects of type $\boldsymbol{A}$ and $\langle \boldsymbol{M} \rangle$ for values of $\langle \boldsymbol{A} \rangle$. However, **<*M*>** is used for both in our implementation.

Consider the following function:

```
fun id : <x:o> -> <nd x> -> <nd x>
    = fn <x> => fn <D> => <D>;
```

In this example, **<x>** is formally an $\boldsymbol{M}$ while **<D>** is an $\langle \boldsymbol{M} \rangle$. Since the type is dependent on the first argument, the first argument to `id` must be data, not computation. For example, consider that **<p>** and `e` both have type **<o>**.

- The expression `id` **<p>** has type **<nd p> -> <nd p>**.
- The expression `id e` is not well-typed as the type depends on the evaluation of `e`. In this situation, Delphin will report *Incompatible Types (LF vs Meta)* indicating that the function must be applied only to a value **<*M*>** but it was given an arbitrary computation. In other words, the function formally expects an $\boldsymbol{A}$ but it was given an $\langle \boldsymbol{A} \rangle$.

Therefore, we emphasize that there is a significant difference between functions of type **<x : *A*> ->** τ and functions of type **<*A*> ->** τ. The first can only be applied to expressions of the form **<*M*>** while the second can be applied to any well-typed e. First, note that in the above example, we could have treated the first argument implicitly. In fact, for any function which needs to only be applied to an **<*M*>**, it could have been handled implicitly. Therefore, this issue is usually completely hidden. Second, the programmer can be explicit as our type reconstruction algorithm takes all expressions of the form **<*M*>** and reasons that they may be interpreted as either

an $\boldsymbol{M}$ or $\langle \boldsymbol{M} \rangle$. Our type reconstruction algorithm is very robust in inferring omitted information and if ambiguous, it will prefer the creation of functions that range over computation $\langle \boldsymbol{A} \rangle$ instead of values $\boldsymbol{A}$.

We now return to our translator and write the function `toComb`, which traverses natural deduction derivations and uses `ba` to convert them into combinators. This example is given in Example 4.5.3 and extended with world information in Example 5.4.3. We show a slight variation to illustrate some syntactic sugar on cases.

```
type paramFun = <(nd B)#> -> <comb B>;
params = <nd A>, <comb A>;
fun toComb : paramFun -> <nd A> -> <comb A>
  = fn W <impi [d] D d> =>
         (case ({<d>}{<h>} toComb (W with <d> => <h>) <D d>)
            of ({<d>}{<h>} <H h>) => ba <[h] H h>)
     | W <impe D1 D2> => (case ((toComb W <D1>), (toComb W <D2>))
                           of (<H1>, <H2>) => <MP H1 H2>)
     | W <d#> => W <d>;
```

We will see that it is necessary to introduce new parameters of **`nd A`** and **`comb A`** together. In order to maintain the relationship between these parameters, we pass around a *parameter function*, i.e. a computational function whose domains ranges over parameters. In this example, we pass around a function `W` of type **`<(exp B)#> -> <comb B>`**. The first line declares a type alias `paramFun` using the `type` keyword. Type aliases can greatly increase the readability of programs. Note that **`B`** occurs free and hence is implicitly attached to the first argument.

The second line uses the `params` keyword to declare that the function that follows is intended to handle the same parameters as we declared for `ba`. Technically, as we did not change the world from our last `params` declaration, this has no effect but we place it here for emphasis.

We now turn to discussing the actual `toComb` function. The first argument is the

parameter function `W`, and we will need to update this function when we create new parameters. `W` will be called when we encounter a dynamically created parameter. In the **`impi`** case, we recurse on **`D`** by first creating a parameter **`d`** to which **`D`** can be applied. However, `W` is not defined with respect to the new **`d`**. Therefore, before recursing, we create a fresh combinator **`h`** and extend `W`, via the `with` keyword, to map **`d`** to **`h`**. The `with` construct is syntactic sugar used to extend parameter functions (Section 3.7.2) The expression "`{<d>}{<h>} toComb (W with <d> => <h>) <D d>`" contains a recursive call in the presence of the new **`d`** and **`h`**. During the recursion, the **`d`** will be converted to **`h`** by the last case of `toComb`. Because the **`h`** may occur in the result, we utilize higher-order matching by matching the result against "`{<d>}{<h>}<H h>`". The variable **`H`** is an LF function resulting from abstracting away all occurrences of **`h`** in the result. The call to bracket abstraction `ba` internalizes this abstraction in the combinator calculus. The **`app`** case recurses on both components and glues the results together with **`MP`**. Finally, the last case handles parameters by applying the parameter function `W`. Note that the pattern `<x#>` matches only parameters and the pattern variable **`x`** has type `(exp A)#`.

The formal grammar for cases c is given in Figure 6.7. Patterns are a subset of expressions e with an underscore (_) as a catch-all. Additionally, LF objects and types occurring in patterns are extended with *id*`#` to allow the programmer to indicate that a pattern variable is a parameter variable, which was used in our last case of `toComb`. One may formally declare pattern variables using `[...]`, but it can always be omitted (which we recommend) as Delphin will automatically declare all free variables in patterns as pattern variables.

		Concrete Syntax	Formal (Chapter 4)
$e_{\text{pat}}, f_{\text{pat}}$	::=	*id*	$\boldsymbol{x}$ or u
	|	**<$\boldsymbol{M}_{\mathbf{pat}}$>**	$\boldsymbol{M}$ or $\langle \boldsymbol{M} \rangle$
	|	`()`	$()$
	|	(*e*pat`,` *f*pat`)`	$(e,\ f)$
	|	{*newdec*} *e*pat	$\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e$
	|	*e*pat **\x**	$e \backslash \boldsymbol{x}$
	|	_	(catch-all pattern)
	|	*e*pat `:` τ	(type ascription)
$\boldsymbol{M}_{\mathbf{pat}}$	::=	***id*#**	$\boldsymbol{x}^{\#}$ (parameter variable)
	|	... (Same as $\boldsymbol{M}$ but using $\boldsymbol{A}_{\mathbf{pat}}$)	
$\boldsymbol{A}_{\mathbf{pat}}$	::=	***id*#**	$\boldsymbol{A}^{\#}$ (parameter type)
	|	... (Same as $\boldsymbol{A}$ but using $\boldsymbol{M}_{\mathbf{pat}}$)	
patvar	::=	*id* `:` τ	$(u \in \tau)$
	|	**<*id* `:` *A*>**	$(\boldsymbol{x} \in \boldsymbol{A})$
	|	**<*id* `:` *A*#>**	$(\boldsymbol{x} \in \boldsymbol{A}^{\#})$
c	::=	`[`*patvar*`]` *c*	$\epsilon \alpha \in \delta.\ c$
	|	{*newdec*} *c*	$\nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ c$
	|	*c* **\x**	$c \backslash \boldsymbol{x}$
	|	*e*pat `=>` *f*	$e \mapsto f$
	|	`(`*e*pat `and ... and` *e*pat`)` `=>` *f*	$\overline{e} \mapsto f$
	|	`(`*e*pat `...` *e*pat`)` `=>` *f*	(sugar building `and`)

Figure 6.7: Concrete Syntax for Cases *c*

Our *new* operator can range over cases, which comes up in `toComb` when we look at the desugared `with` statement (Section 3.7.2). The actual desugared form of `(W with <d> => <h>)` is:

```
((fn {<d>}{<h>}(<d> => <h>)
   | {<d>}{<h>}(<y#> => (let val <R> = W <y>
                             in {<d>}{<h>}<R> end) \d \h ))
 \d \h)
```

There is little reason for a programmer to use the \ operator over cases, but it will

come up during evaluation. Patterns with `and` are used to introduce functions whose types use `and`. However, as is similar to ML and Haskell, we allow the programmer to juxtapose multiple patterns together in order to introduce a curried function that looks at the arguments together. In ML and Haskell a curried function that looks at its arguments together is handled by doing case analysis over a pair; in Delphin, we instead use `and` for dependencies. For example, our `toComb` function expressed both arguments via juxtaposition and under-the-hood this function is converted to:

```
fun toComb : paramFun -> <nd A> -> <comb A>
  = fn X1 => fn X2 => case (X1 and X2)
                        of (W and <impi [d] D d>) => ...
                         | (W and <impe D1 D2>) => ...
                         | (W and <d#>) => ...;
```

6.4 Programs as Proofs

If a function passes the coverage checker, then its application is guaranteed never to get stuck. If a function passes the coverage checker and is terminating, then it is total and may be interpreted as a proof. We have implemented a coverage-checking algorithm (Chapter 5) as well as a termination checker, allowing for Delphin to serve as a system for writing mathematical proofs.

6.4.1 Coverage

Since computation occurs with respect to a dynamic collection of parameters, determining if a list of cases is exhaustive is a non-trivial problem. If a list of cases is incomplete, Delphin will return a *Match Non-Exhaustive Warning* providing a list of cases which are missing. Additionally, if one tries to call a function which makes

		Concrete Syntax	Formal (Chapter 5)
paramtype	::=	`<`$\boldsymbol{A}_{\mathbf{pat}}$`>`	The type $\boldsymbol{A}$ in $(\Omega, \boldsymbol{A})$
	|	`[`*patvar*`]` *paramtype*	Building Ω in $(\Omega, \boldsymbol{A})$
worlddec	::=	`* ;`	$*$ world
	|	`. ;`	$\cdot$ world (i.e. CLOSED)
	|	*paramtype* `;`	Single parameter Type
	|	*paramtype* `, ... ,` *paramtype* `;`	Multiple parameter types

Figure 6.8: Concrete Syntax for World Declarations (`params` Declarations)

sense with respect to an incompatible collection of parameters, Delphin will return a *World Subsumption Error*. If no warning message is generated, then programs are guaranteed not to get stuck (i.e. type safety holds).

As illustrated, we distinguish functions based on the collection of parameters they support, which we refer to as their respective worlds. The `params` keyword declares the world of all `fun` declarations that follow. We have already shown examples using `params` and we now present its formal grammar in Figure 6.8. The default world is `*` declaring functions to support all possible parameters, while one can use `params = .` to declare functions that are not intended to support any parameters, also known as the CLOSED world. Open worlds are simply a set of LF types, which we express as a comma-delimited list of $\boldsymbol{A}_{\mathbf{pat}}$. The free variables in $\boldsymbol{A}_{\mathbf{pat}}$ can be declared explicitly via `[...]` similar to pattern variables, but it may also always be inferred. We use $\boldsymbol{A}_{\mathbf{pat}}$ instead of $\boldsymbol{A}$ so the programmer can easily annotate variables as parameter variables, i.e. the **`E`** in **`<equiv (s P) (E#) (var' one)>`** has type **`exp#`**. The addition of world information requires an additional annotation to functions, whose transformation is described in Chapter 5.

It is important to note that the `params` keyword only annotates world information to top-level functions declared with `fun`. The `toComb` function is declared to support

parameters, while the first argument to `toComb` is a parameter function `W` that is not accessible in scope of any new parameters. For example, if we forgot to extend `W` in the **`impi`** case, then we would get a warning message, but we can recursively access `toComb` without any extension as it is declared to support the parameters that were created.

The evaluation of top-level expressions (using `val`) is *always* declared in the CLOSED world. As an example, consider:

```
val test1 = toComb (fn .) <impi [d:nd p] d>
```

Notice that `test1` must be evaluated in the CLOSED world, as the first argument `(fn .)` is a parameter function which is total in the empty world but not necessarily in a world with parameters (see Example 5.4.4)

Therefore, all top-level expressions and functions are automatically annotated with world information based on `params`, following the discussion in Chapter 5. We find that this is the most intuitive way to program with Delphin, but this also means that one does not have as much flexibility as if we allowed the programmer to perform the world annotation manually via the $\nabla\mathcal{W}.\ \tau$ construct.

6.4.2 Termination

The prototype termination checker for Delphin supports lexicographic extensions of the subterm ordering (Rohwedder and Pfenning 1996) over the inputs. The termination order is lexicographic on the first input, followed by the second, and so on. However, we omit from the termination order both (1) implicit arguments and (2) arguments containing computation-level functions. For example, the termination checker for our `toComb` function just checks that the second explicit argument (of type **`<nd A>`**) gets smaller.

The subterm ordering for LF objects has been formalized by Rohwedder and Pfenning (1996) and we extend it in a very straightforward manner to handle computation-level pairs. Recall that $\langle \boldsymbol{M} \rangle = (\boldsymbol{M},\ ())$. Therefore, we define $\langle \boldsymbol{M} \rangle < \langle \boldsymbol{N} \rangle$ if and only if $\boldsymbol{M} < \boldsymbol{N}$ following the LF subterm ordering.

Our `ba` and `toComb` functions pass the termination checker, but the function `evalBeta` fails due to the **`app`** case. This is to be expected as the evaluation of untyped λ-expressions can indeed be non-terminating.

Since `ba` and `toComb` pass the coverage and termination checker, Delphin has verified that we can view `toComb` as a proof that every natural deduction derivation can be converted into a combinator of the appropriate type.

Finally, Delphin checks for the termination of mutually recursive functions by checking that each function respects its termination order when making recursive calls to itself or any mutually defined function. When we are checking the termination order for function i, we verify that the order gets smaller when calling function $j \leq i$ and we verify that it gets smaller *or stays equal* when calling function $j > i$. As a very contrived example, the following mutually recursive function passes the termination checker, but it would fail if we reversed the order of the two functions:

```
fun foo
  = fn <E> => bar <E>

and bar
  = fn <app E1 E2> => foo <E1>
     | <M> => <M>;
```

```
Delphin, Release Version 1.5.1, April 23, 2008
D-- use "chapter6.d";
...
D-- val test1 = evalBeta <lam [x:exp] x> ;
val test1 : <exp>
      = <lam ([x:exp] x)>
D-- val test2 = toComb (fn .) <impi [d:nd p] d> ;
val test2 : <comb (p ar p)>
      = <MP (MP S K) K>
D-- val test3 = toComb (fn .) <impi [x:nd p] impi [y : nd p] x> ;
val test3 : <comb (p ar p ar p)>
      = <MP (MP S (MP K K)) (MP (MP S K) K)>
```

Figure 6.9: Delphin Sample Execution

6.5 Interactive Top Level

Delphin provides a top-level interactive loop where one may write, execute, and experiment with programs. Error and warning messages are reported in the same format as in SML of New Jersey, allowing one to use the SML Emacs mode to jump to error locations easily. One may load LF Signatures (Twelf files) by typing `sig use "filename"` and may run Delphin files by typing `use "filename"`. For illustrative and debugging purposes, Delphin provides the ability to pretty-print arbitrary expressions with options to make pattern variables and implicit arguments explicit. Additionally, Delphin allows the disabling/enabling of the coverage checker and the termination checker.

Figure 6.9 illustrates a use of the interactive top level in running our `evalBeta` and `toComb` functions. The file `chapter6.d` is the code presented in this chapter. Recall that the first argument to `toComb` is a parameter function. We supply it with an empty function (`fn .`), which is appropriate as no parameters exist at the top level.

6.6 ML Interface

The Delphin implementation contains a `Delphin` structure with functions to start the top level and load files, as well as the ability to set many environment variables.

The major functions are shown below:

- `Delphin.top ();`

  Starts the interactive top level (Section 6.5) designed to be similar to that of SML of New Jersey.

- `Delphin.loadFile ("`*filename*`");`

  Executes the Delphin file in *filename*. This is equivalent to doing `use "`*filename*`";` in the top level.

We next show the major environment variables via assignments to their default values:

- `Delphin.debug := false;`

  When set to `false` Delphin will not output the comparisons made by the termination checker and will not output the body of functions.

  When set to `true` Delphin will

  1. output the body of functions conforming to preferences set in `pageWidth`, `printPatternVars`, and `printImplicit`.
  2. Output every comparison made by the termination checker when it is enabled via `enableTermination`.

- `Delphin.enableCoverage := true;`

  When set to `true` Delphin utilizes the `params` declaration to annotate top-level functions with world information as discussed in Chapter 5. Top-level expressions are also annotated, but always with the CLOSED world.

When set to `false`, functions will not be annotated with world information and the `params` declaration will be ignored. Delphin functions and expressions will conform to Chapter 4 and there will be no meta-level check to tell if functions are covered.

- `Delphin.enableTermination := true;`

  When set to `true` it will check the termination of functions using the ordering discussed in Section 6.4.2.

  When set to `false` no termination analysis will be performed.

- `Delphin.stopOnWarning := false;`

  When set to `false` Delphin will output `WARNING` messages when functions are not covered or are not terminating.

  When set to `true` Delphin will stop on the first `WARNING` message. This may be useful if the programmer wants to make sure that functions correspond to proofs without checking for warnings.

- `Delphin.pageWidth := 80;`

  This is used to indicate the maximum number of characters per a line that the output will produce.

- `Delphin.printPatternVars := false;`

  When set to `false`, pattern variables appear as free variables in patterns and parameter variables are annotated with `#`.

  When set to `true`, Delphin will additionally output the explicit declaration of all pattern variables using the `[...]` syntax. This may be useful for debugging coverage errors as the coverage checker returns a list of cases that are missing

and due to dependencies it may be complicated to deduce the type of the pattern variables manually.

- `Delphin.printImplicit := false;`

  When set to `false`, Delphin's output will omit implicit information with respect to both data (LF) and computation.

  When set to `true`, all implicit information will be outputted with respect to both data (LF) and computation.

- `Delphin.doubleCheck := false;`

  When set to `false`, Delphin will trust the type reconstruction algorithm.

  When set to `true`, Delphin will additionally run a type checker to double-check the result. This is an expensive operation and only useful for debugging.

6.7 Installation Instructions

The only prerequisite to installing Delphin is to first install SML of New Jersey for your appropriate platform. Delphin has been developed and tested using v110.65.

After installing SML and downloading the Delphin source, there are two ways of executing Delphin programs:

- Assuming you are using a Unix-like system you can do `make` in the root Delphin directory, which will create an executable file `bin/delphin` which will take one directly to the Delphin interactive top level.

- For the most flexibility (and if you are not using a Unix-like system) you must run `sml` and compile Delphin using the `sources.cm` in the `src` directory. As an example, if you are in the root Delphin directory, you can start the interactive top level as follows:

```
yourcomputer:~/delphin> cd src
yourcomputer:~/delphin/src> sml
Standard ML of New Jersey v110.65
- CM.make "sources.cm";
...
- Delphin.top();

Delphin, Release Version 1.5.1, April 23, 2008

D-- Have Fun!
```

6.8 Case Studies

Delphin has been used for numerous other examples, which are all available on our website. Here we just outline a few.

- *Hindley-Milner Type Inference.* We have an extensive case study of Mini-ML. We have implemented the operational semantics, proved value soundness, and proved type preservation. The most interesting feature is a Hindley-Milner style type-inference algorithm. Parameters are used in place of references, where a new parameter can be thought of as a fresh "memory location". We pass around environments mapping memory locations to data, which are implemented as parameter functions. As parameters cannot escape their scope (Corollary 5.5.20), our type system gives us the beneficial side-effect of guaranteeing that all memory locations are *freed*.

- *Logic.* We have implemented and proved cut elimination of the intuitionistic sequent calculus. In the framework of expressing morphisms between logics, we have expressed translations from HOL to Nuprl as well as translations between sequent and natural deduction calculi.

- *Church-Rosser.* We have proved the Church-Rosser theorem for the untyped

λ-calculus using the method of parallel reduction.

- *Equivalence between HOAS and de Bruijn notation.* A more advanced use of parameter functions occurs when we convert expressions of the untyped λ-calculus encoded in HOAS to an encoding utilizing de Bruijn indices. In this dissertation we have shown how to convert between both representations and ensure that our conversion is correct by returning an equivalence proof using the **`equiv`** relation from Example 2.2.6. However, we can additionally prove that this relation is deterministic. In other words, if **`equiv E T`** and **`equiv E T'`**, then **`T`** and **`T'`** must be identical. Additionally, if **`equiv E T`** and **`equiv E' T`**, then **`E`** and **`E'`** must be identical. This allows us to conclude that if we translate an object into the other representation and back again, we will get the same object that we started with. We show the code for this advanced example in Section 6.10.3.

  This advanced example illustrates the expressivity of parameter functions, as we can use them to express complex statements about parameters (or contexts). For example, we use parameter functions to express the invariant that all parameters are mapped to different de Bruijn indices. Note that this required us to reason about *impossibility*. This is achieved by encoding an empty type without any constructors:

  ```
  sig <empty : type>;
  ```

  From **`empty`** one can prove anything. Any function which takes objects of type **`empty`** is total as all possible cases are covered – since there are none. However, this function can never be applied since there is no way to generate any such input.

The reasoning power of this comes from writing functions that *return* objects of type **empty**. For example, the invariant that every **exp#** is mapped to only one de Bruijn index, is maintained by passing around a function of type:

```
type detB-Contradiction = <lt (s P') (s P)>
                        -> <equiv (s P) E (var' one) #>
                        -> <equiv (s P') E (var' one) #>
                        -> <empty>;
```

The **<lt (s P') (s P)>** condition is a less-than relation asserting that **P'** is smaller than **P**, which guarantees that the second and third arguments are different mappings for **E**.

It is possible that the coverage checker will determine that a type is empty. For example, it may perform a splitting discussed in Chapter 5 which results in zero cases. However, it is in general undecidable to determine if a type is empty. Therefore, the **empty** datatype is also useful for proving a type is empty when it requires more complicated reasoning than splitting. For example, the following `contraApp` function is total:

```
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun contraApp : <P:nat>
                -> <equiv P (app E1 E2) (var' X)>
                -> <empty>
   = fn <s P> <eqVar D> => contraApp <P> <D>;
```

Here we proved by induction that the type **<equiv P (app E1 E2) (var' X)>** is empty, which follows from the definition of `equiv` and the `params` declaration. Note that the **E** in **<equiv (s P) (E#) (var' one)>** is declared to be a parameter type and if it was not, then this function would not be covered.

Therefore, when we encounter such a case, we can now handle it by writing:

```
(fn .) (contraApp <P> <D>)
```

The coverage checker can easily conclude that for any τ, the function `fn .` is a total function of type `<empty> -> ` τ as there is no way to construct an object of type **empty**.

6.9 Implementation Details

In this section we will discuss the key implementation concerns involving the representation level, type reconstruction, implicit arguments, operational semantics, coverage checking, and termination checking.

- *Representation Level (LF).* Recall that the programmer can take advantage of HOAS by representing object-language bindings as LF functions. Here we will briefly discuss the implementation of LF which happens under-the-hood.

  Delphin borrows some Twelf code to deal with LF-level unification and LF-level type reconstruction, but with extensions. The implementation of LF in Twelf exploits invariants that do not hold in Delphin.[1]

  The main property of LF is that all terms possess a unique canonical form modulo $\alpha\beta\eta$. The implementation utilizes de Bruijn notation to avoid dealing with α-equivalence. For efficiency, we use a spine notation similar to the one by Cervesato and Pfenning (2003). Instead of constantly computing canonical forms, we use *weak-head normal forms* (Dowek et al. 1998; Cervesato and Pfenning 2003), which is a very useful optimization as it is wasteful to compute the entire canonical form only to discover that two terms differ in their top-level constructors.

[1] For example, all pattern (existential) variables make sense in the empty context in Twelf, but this does not hold in Delphin.

Coverage analysis and type reconstruction make use of eigenvariables (also called logic variables) and unification. In the presences of eigenvariables, we must add some form of closures. Therefore, we utilize explicit substitutions and follow the unification algorithm by Dowek et al. (1998) which describes higher-order unification in this setting. The fragment of higher-order patterns are of the form $E\ x_1\ \ldots\ x_n$ where x_i is a fresh parameter (with respect to E) and all x_i's are distinct. Equations that fall within the fragment of higher-order patterns are solved immediately, while all other equations are postponed as constraints. Such constraints are reawakened when a variable in the postponed equation is instantiated. It is also important to note that the Delphin unification algorithm extends the LF version with support for variables of parameter type (i.e. variables that can only be instantiated with parameters) and computation-level types.

- *Type Reconstruction.* The type reconstruction algorithm converts terms into the Delphin internal syntax. Our algorithm will always report either a principal type, a type error, or in the case of leftover constraints, an indication that the type is ambiguous and needs further annotations. The result of this algorithm has no free variables and no logic variables, and thus is a valid expression. Reconstruction occurs in three stages:

  1. The first stage is an *approximate* type reconstruction, which effectively throws out all dependent types and determines the simply-typed form of the type.
  2. With this approximate information we create appropriate eigenvariables and employ unification to infer the exact type (with dependencies).
  3. The formal system (internal syntax) requires an explicit declaration of

pattern variables, but we allow the user to express patterns by just using free variables. Therefore, the final step finds all free variables and explicitly quantifies them to the respective pattern where the variable first occurs.

As we mentioned earlier, Delphin uses the concrete syntax `<`$\boldsymbol{M}$`>` ambiguously to refer to either $\boldsymbol{M}$ or $\langle \boldsymbol{M} \rangle$ (Figure 6.6). Therefore, the type reconstruction algorithm specially handles `<`$\boldsymbol{M}$`>` allowing it to refer to either version. If it can be typed with either interpretation, Delphin will prefer $\langle \boldsymbol{M} \rangle$ over $\boldsymbol{M}$ when converting the term to the internal (formal) grammar.

Finally, we include a type checker which can be manually employed (via the environment variable `Delphin.doubleCheck` discussed in Section 6.6), but it is only necessary for debugging purposes as type reconstruction is guaranteed to return well-typed expressions.

- *Implicitness*. It is redundant to explicitly supply an input argument that is indexed in another input argument. Therefore, we extend the reconstruction algorithm above to allow free variables in types. Free variables are implicitly quantified via `and` with the first argument in which they occur. Similarly, we also implicitly quantify free variables in type types of LF constants, just as Twelf does. When applying a function, the reconstruction algorithm will automatically fill in the implicit arguments. This support is just a frontend convenience and does not affect the underlying theory. Evaluation, coverage, and termination are unaware of what is implicit/explicit in the frontend.

  Additionally, Delphin supports the syntax "_" for terms that are explicit but can be inferred. This syntax instructs the reconstructor to fill in the term automatically, which is possible when the type forces it to be a particular

term.

- *Operational Semantics*. The evaluator for Delphin is straightforward. If multiple branches match, it will pick the first branch that matches and continue execution. Pattern variables are filled in with eigenvariables and we employ unification during runtime to instantiate them. The problem of higher-order matching is much simpler than full-blown unification, so there is room here to speed up the execution.

- *Coverage Checking*. We have implemented the algorithm discussed in Chapter 5. The programmer uses the `params` keyword to declare a function's world, and the world annotation proceeds as discussed. The enhancements in Section 5.3.8 have been made. The core algorithm generates a list of base patterns while one of our enhancements allows the programmer to write a more general pattern covering multiple base patterns. If the programmer writes a pattern which falls outside the fragment of higher-order patterns (Dowek et al. 1998), an appropriate error message will be displayed as Delphin will not be able to determine when that pattern will match.

- *Termination*. The termination algorithm is straightforward. As discussed in Section 6.4.2 we have a fixed strategy for choosing a termination ordering, and the programmer needs to order the arguments in the order in which things get smaller. Alternatively, we could allow the arguments in any order and force the user to specify the termination order, or automatically infer the order. However, for simplicity we have chosen to use a fixed order.

  The termination algorithm works by applying functions to eigenvariables and performing a virtual execution on every branch stopping at calls to other already termination-checked functions and checking the termination order at

every recursive call.

6.10 Code Source

We conclude this chapter by presenting the actual code for all the examples presented throughout this dissertation as well as the advanced case study proving that **`equiv`** is deterministic. We have annotated our code with comments `(* ... *)` indicating any warning messages that are reported as well as the output of our sample executions.

6.10.1 Simply-Typed Examples

We present all the simply-typed examples from Chapter 3 with appropriate world annotations (Chapter 5), which are made automatically based on the `params` keyword. The code in this section is available on our website as `simpleDelphin.d`.

```
(* Natural Numbers (Example 2.1.1) *)
sig <nat : type> %name N n
    <z : nat>
    <s : nat -> nat>;

(* Untyped λ-expressions using HOAS (Example 2.1.2) *)
sig <exp : type> %name E x
    <lam : (exp -> exp) -> exp>
    <app : exp -> exp -> exp>;

(* Untyped λ-expressions a la de Bruijn (Example 2.1.4) *)
   (* Variables *)
   sig <variable : type> %name X
       <one : variable>
       <succ : variable -> variable>;
   (* Expressions *)
   sig <term : type> %name T t
       <lam' : term -> term>
       <app' : term -> term -> term>
       <var' : variable -> term>;

(* Untyped Combinators (Example 2.1.5) *)
sig <comb : type> %name C u
    <K : comb>
    <S : comb>
    <MP : comb -> comb -> comb> ;

(* Addition (Example 3.3.3) *)
params = <exp>, <comb>;
fun plus : <nat> -> <nat> -> <nat>
 = fn <z> <M> => <M>
    | <s N> <M> => let val <x> = plus <N> <M>
                   in <s x>
                   end;
```

```
(* Closed Eager Evaluator (Example 3.3.4 and 5.2.3 *)
params = .;
fun eval : <exp> -> <exp>
 = fn <app E1 E2> =>
            (case (eval <E1>, eval <E2>)
               of (<lam [x] F x>, <V>) => eval <F V>
                     (* Match Non-Exhaustive Warning:
                      *  We do not handle <app>
                      *
                      * Termination Warning:
                      *  <F V> not smaller than <app E1 E2>
                      *)
            )
    | <lam [x] E x> => <lam [x] E x> ;

(* Open Evaluator (Example 3.3.5)
 * This is also for Example 5.2.2, but we
 * use a different world than in the code.
 * The important part is that the world contains exp.
 *)
params = <exp>, <comb>;
fun evalBeta : <exp> -> <exp>
 = fn <app E1 E2> => (case (evalBeta <E1>, evalBeta <E2>)
                      of (<lam [x] F x>, <V>) => evalBeta <F V>
                            (* Termination Warning:
                             *  <F V> not smaller than <app E1 E2>
                             *)
                       | (<x>, <V>) => <app x V>)
    | <lam [x] E x> => (case ({<x>} evalBeta <E x>)
                         of {<x>}<E' x> => <lam [x] E' x>)
    | <x#> => <x> ;
(*
 * Sample execution of evalBeta
 * (Example 3.3.6 and 5.2.4)
 *)
val sample1 = evalBeta <lam [y:exp] y> ;
(* OUTPUT:
 * val sample1 : <exp>
 *     = <lam ([x:exp] x)>
 *)
```

```
(* Variable Counting (Example 3.3.7 *)
params = <exp>, <comb>;
fun cntvar  : <exp> -> <nat>
 = fn <app E1 E2>     =>  plus (cntvar <E1>) (cntvar <E2>)
    | <lam [x] E x>   =>  (case ({<x>} cntvar <E x>)
                           of  ({<x>} <N>)  => <N>)
    | [<x:exp#>] <x>  => <s z>;

(* Sample execution of cntvar (Example 3.3.8) *)
val sample2 = cntvar <lam [y:exp] app y y> ;
(* OUTPUT:
 * val sample2 : <nat>
 *     = <s (s z)>
 *)

(* Lemma 3.4.2 is over any type A, τ and σ,
 * but we just show it for one example.
 *)
params = *;
fun lemma-part1 : ({<x:exp#>} (<exp> -> <nat>))
                  -> (({<x:exp#>}<exp>) -> {<x:exp#>}<nat>)
 = fn u1 => fn u2 => {<x:exp#>} (u1 \x) (u2 \x) ;

params = *;
fun lemma-part2 : (({<x:exp#>}<exp>) -> {<x:exp#>}<nat>)
                  -> ({<x:exp#>} (<exp> -> <nat>))
 = fn u1 => fn {<x:exp#>} ((E \x) => (u1 E) \x) ;

params = *;
fun lemma-part3 : ({<x:exp#>} (<exp> * <nat>))
                  -> (({<x:exp#>}<exp>) * {<x:exp#>}<nat>)
 = fn {<x>}((u1 \x), (u2 \x)) => (u1, u2) ;

params = *;
fun lemma-part4 : (({<x:exp#>}<exp>) * {<x:exp#>}<nat>)
                  -> ({<x:exp#>} (<exp> * <nat>))
 = fn (u1, u2) => {<x:exp#>} ((u1 \x), (u2 \x));

val exampleF : (({<x:exp#>}<exp>) -> {<x:exp#>}<nat>)
 = fn {<x>} <x> => {<x>} <s z>
    | {<x>} <_> => {<x>} <z> ;
```

```
val lemmaTest = {<x>} ((lemma-part2 exampleF) \x) <x> ;
(* OUTPUT:
 * val exampleF : ({<x : exp#>} <exp>) -> ({<x : exp#>} <nat>)
 *     = fn ...
 * val lemmaTest : {<x : exp#>} <nat>
 *     = {<x : exp#>} <s z>
*)

(* Combinators to λ-expressions (Example 3.7.1) *)
params = . ;
fun comb2exp : <comb> -> <exp>
  = fn <K> => <lam [x:exp] lam [y:exp] x>
     | <S> => <lam [x:exp]
                    lam [y:exp]
                    lam [z:exp] app (app x z) (app y z)>
     | <MP C1 C2> => let
                       val <E1> = comb2exp <C1>
                       val <E2> = comb2exp <C2>
                     in
                       <app E1 E2>
                     end;

val id1 :  <exp> = comb2exp <MP (MP S K) K> ;
(* OUTPUT:
 * val id1 : <exp>
 *  = <app
 *      (app
 *        (lam
 *          ([x:exp]
 *            lam ([y:exp] lam ([z1:exp] app (app x z1) (app y z1)))))
 *        (lam ([x:exp] lam ([y:exp] x))))
 *      (lam ([x:exp] lam ([y:exp] x)))>
 *)

val reducedId1 = evalBeta id1;
(* OUTPUT:
 * val reducedId1 : <exp>
 *        = <lam ([x:exp] x)>
 *)
```

```
(* Bracket Abstraction (Example 3.7.2) *)
params = <exp>, <comb>;
fun ba : <comb -> comb> -> <comb>
  = fn <[x] x> => <MP (MP S K) K>
     | <[x] MP (C1 x) (C2 x)> =>
                      let
                        val <C1'> = ba <[x] C1 x>
                        val <C2'> = ba <[x] C2 x>
                      in
                        <MP (MP S C1') C2'>
                      end
     | <[x] C> => <MP K C> ;

(* Untyped λ-expressions to Combinators (Example 3.7.3) *)
type convParamFun = <exp#> -> <comb>
params = <exp>, <comb>;
fun exp2comb : convParamFun  -> <exp> -> <comb>
 = fn W =>
      fn <lam [x] E x> =>
            (case ({<x>}{<u>} exp2comb (W with <x> => <u>) <E x>)
              of ({<x>}{<u>} <C u>) => ba <[u] C u>)
       | <app E1 E2> =>
            let
               val <C1> = exp2comb W <E1>
               val <C2> = exp2comb W <E2>
            in
               <MP C1 C2>
            end
       | <x#> => W <x>;
```

```
(* Same as above with desugared "with" (Section 3.7.2) *)
params = <exp>, <comb>;
fun exp2comb : convParamFun  -> <exp> -> <comb>
 = fn W =>
      fn <lam [x] E x> =>
            (case ({<x:exp#>}{<u:comb#>}
                      exp2comb
                         ((fn {<x>}{<u>}(<x> => <u>)
                            | {<x>}{<u>}(<y> =>
                                          (let
                                             val <R> = W <y>
                                           in
                                             {<x>}{<u>}<R>
                                           end) \x \u )
                          ) \x \u)
                          <E x>)
              of ({<x:exp#>}{<u:comb#>} <C u>) => ba <[u] C u>)
       | <app E1 E2> =>
            let
              val <C1> = exp2comb W <E1>
              val <C2> = exp2comb W <E2>
            in
              <MP C1 C2>
            end
       | <x#> => W <x>;

(* Example Execution of exp2comb (Example 3.7.4) *)
val idC = exp2comb (fn .) <lam [x] x> ; (* = <MP (MP S K) K> *)
(* OUTPUT:
 * val idC : <comb>
 *     = <MP (MP S K) K>
 *)
```

```
(* HOAS to de Bruijn (Example 3.8.1) *)
params = <exp>, <comb>;
fun toDebruijn : (<exp#> -> <variable>) -> <exp> -> <term>
 = fn W =>
    fn <lam [x] E x> =>
          let
            val W' = fn <y#> => <succ> @ (W <y>)
          in
            case ({<x:exp#>} toDebruijn (W' with <x> => <one>) <E x>)
              of {<x:exp#>}(<T>) => <lam' T>
          end
      | <app E1 E2> =>
          let
            val <T1> = toDebruijn W <E1>
            val <T2> = toDebruijn W <E2>
          in
            <app' T1 T2>
          end
      | <x#> => <var'> @ (W <x>);

(* de Bruijn to HOAS (Example 3.8.2) *)
params = <exp>, <comb>;
fun toHOAS : (<variable> -> <exp>) -> <term> -> <exp>
 = fn W =>
    fn <lam' T> =>
         (case ({<x:exp#>}
                  toHOAS
                     ((fn {<x>}(<one> => <x>)
                        | {<x>}(<succ X> => (let
                                               val <E> = W <X>
                                             in {<x>}<E> end) \x)
                      ) \x)
                     <T>)
                of {<x>} <E x> => <lam [x] E x>)
     | <app' T1 T2> =>
         let
           val <E1> = toHOAS W <T1>
           val <E2> = toHOAS W <T2>
         in
           <app E1 E2>
         end
     | <var' X>  => W <X>;
```

```
(* Examples converting between representations (Example 3.8.3) *)
val test1 = toDebruijn (fn .) <lam [x] lam [y] app x y>;
(* OUTPUT:
 * val test1 : <term>
 *     = <lam' (lam' (app' (var' (succ one)) (var' one)))>
 *)

val test2 = toHOAS (fn .) test1;
(* Match Non-Exhaustive Warning generated for "fn ."
 * OUTPUT:
 * val test2 : <exp>
 *     = <lam ([x:exp] lam ([x1:exp] app x x1))>
 *)
```

6.10.2 Dependently-Typed Examples

We now present the dependently-typed examples from Chapter 4 with world annotations (Chapter 5). This code is available on our website as `depDelphin.d`.

```
(* Natural Numbers (Example 2.1.1) *)
sig <nat : type> %name N n
    <z : nat>
    <s : nat -> nat>;

(* Untyped λ-expressions using HOAS (Example 2.1.2) *)
sig <exp : type> %name E x
    <lam : (exp -> exp) -> exp>
    <app : exp -> exp -> exp>;

(* Formulas (Example 2.2.2) *)
sig <o : type>  %name A
    <ar : o -> o -> o> %infix right 10
    <p : o> (* base type *) ;

(* Natural Deduction (Example 2.2.3) *)
sig <nd : o -> type> %name D d
    <impi : (nd A -> nd B) -> nd (A ar B)>
    <impe : nd (A ar B) -> nd A -> nd B> ;

(* Typed Combinators (AKA Hilbert-Style Calculus) (Example 2.2.4) *)
sig <comb : o -> type> %name H h
    <K : comb (A ar B ar A)>
    <S : comb ((A ar B ar C) ar (A ar B) ar A ar C)>
    <MP : comb (A ar B) -> comb A -> comb B> ;

(* Well-formed untyped λ-expressions a la de Bruijn (Example 2.2.5) *)
   (* an object of type "variable N" is a number representing 1 to N *)
   sig <variable : nat -> type> %name X
       <one : variable (s N)>
       <succ : variable N -> variable (s N)> ;
   (* Expressions (indexed by size of context) *)
   sig <term : nat -> type> %name T t
       <var' : variable P -> term P>
       <lam' : term (s P) -> term P>
       <app' : term P -> term P -> term P> ;
```

```
(* Equality relation (Example 2.2.6) *)
sig <equiv : {P} exp -> term P -> type>
    <eqVar : equiv P E (var' X) -> equiv (s P) E (var' (succ X))>
        (* eqVar is saying:
         *   if E is the Xth variable when the context has size P,
         *   then E is also the (X+1)st variable in the same context
         *        extended with one element.
         *)
    <eqLam :({x:exp}(equiv (s P) x (var' one)) -> equiv (s P) (E x) T)
           -> equiv P (lam [x] E x) (lam' T)>
    <eqApp : equiv P E1 T1 -> equiv P E2 T2
            -> equiv P (app E1 E2) (app' T1 T2)>;

(* Simple identity function *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun id : <A:o> -> <nd A> -> <nd A>
    = fn <A> => fn <D> => <D>;

(* BAD attempt at recursive identity function (Example 4.1.1) *)
(* Causes Match Non-Exhaustive Warnings on inner functions
 * as well as Termination Warnings since the order is set
 * defaultly to the order of the explicit arguments, but
 * we want the order to only be on the second argument.
 * (Solution is to make the first argument implicit.)
 *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun id1 : <A:o> -> <nd A> -> <nd A>
  = fn <A ar B> =>
          (fn <impi [d] D d> =>
                (case {<d:nd A>} id1 <B> <D d>
                   of {<d:nd A>} <D' d> => <impi [d:nd A] D' d>)
          )
     | <B> =>
          (fn <impe D1 D2> =>
                let
                  val <D1'> = id1 <A ar B> <D1>
                  val <D2'> = id1 <A> <D2>
                in
                  <impe D1' D2'>
                end)
        | <A> => fn <d#> => <d>;
```

```
(* Good (fully explicit) recursive identity function
 * (Example 4.1.2)
 * Causes Termination Warnings since the order is set
 * defaultly to the order of the explicit arguments, but
 * we want the order to only be on the second argument.
 * (Solution is to make the first argument implicit.)
 *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun id2 : (<A:o> and <nd A>) -> <nd A>
     = fn (<A ar B> and <impi [d] D d>) =>
                (case {<d:nd A>} id2 (<B> and <D d>)
                   of {<d:nd A>} <D' d> => <impi [d:nd A] D' d>)
        | (<B> and <impe D1 D2>) =>
                let
                  val <D1'> = id2 (<A ar B> and <D1>)
                  val <D2'> = id2 (<A> and <D2>)
                in
                  <impe D1' D2'>
                end
        | (<A> and <d#>) => <d>;

(* Final recursive identity function utilizing implicit types
 * (Example 4.1.3) *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun id3 : <nd A> -> <nd A>
     = fn <impi [d] D d> => (case {<d>} id3 <D d>
                               of {<d>}<D' d> => <impi [d] D' d>)
        | <impe D1 D2> => let
                            val <D1'> = id3 <D1>
                            val <D2'> = id3 <D2>
                          in
                            <impe D1' D2'>
                          end
        | <d#> => <d>;
```

```
(* Combinators to natural deduction derivations
 * (Example 4.5.1 and 5.4.1) *)
params = . ;
fun hil2nd : <comb A> -> <nd A> =
    fn <K> => <impi [x:nd A] impi [y:nd B] x>
     | <S> => <impi [x:nd (A ar B ar C)]
                    impi [y:nd (A ar B)]
                    impi [z:nd A] impe (impe x z) (impe y z)>
     | <MP H1 H2> => let
                      val <D1> = hil2nd <H1>
                      val <D2> = hil2nd <H2>
                    in
                      <impe D1 D2>
                    end;

(* Bracket Abstraction (Example 4.5.2 and 5.4.2) *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun ba : <comb A -> comb B> -> <comb (A ar B)>
  = fn <[x] x> => <MP (MP S K) (K : comb (A ar A ar A))>
     | <[x] MP (H1 x) (H2 x)> =>
                     let
                       val <H1'> = ba <[x] H1 x>
                       val <H2'> = ba <[x] H2 x>
                     in
                       <MP (MP S H1') H2'>
                     end
     | <[x] H> => <MP K H> ;
```

```
(* Natural deduction derivations to Combinators
 * (Example 4.5.3 and 5.4.3) *)
type ndParamFun = (<nd B#> -> <comb B>)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun nd2hil : ndParamFun  -> <nd A> -> <comb A>
 = fn W =>
      fn <impi [d] D d> =>
            (case ({<d>}{<h>} nd2hil (W with <d> => <h>) <D d>)
              of ({<d>}{<h>} <H h>) => ba <[h] H h>)
       | <impe D1 D2> =>
            let
              val <H1> = nd2hil W <D1>
              val <H2> = nd2hil W <D2>
            in
              <MP H1 H2>
            end
       | <d#> => W <d>;

(* Sample execution (Example 4.5.4 and 5.4.4) *)
val test-nd2hil = nd2hil (fn .)  <impi [d:nd p] d> ;
(* OUTPUT:
 * val test-nd2hil : <comb (p ar p)>
 *     = <MP (MP S K) K>
 *)
```

```
(* HOAS to de Bruijn (Example 4.6.1 and 5.4.5) *)
(* We comment out the first argument because it is inferable
 * and we do not want it in the termination order *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun toDebruijn : (* <P:nat> -> *)
                 (<E:exp#> -> <X: variable P> * <equiv P E (var' X)>)
                 -> <E:exp> -> <T:term P> * <equiv P E T>
  = (* fn <P> => *)
     fn W =>
      (fn <lam [x] E x> =>
          let
            val W' : <x:exp#> ->
                      <Y: variable (s P)> * <equiv (s P) x (var' Y)>
               = fn <x#> => let val (<Y>,<D>) = W <x>
                            in (<succ Y>, <eqVar D>) end
          in
            case ({<x:exp#>}{<d:(equiv (s P) x (var' one))#>}
                        toDebruijn (* <s P> *)
                                   (W' with <x> => (<one>,<d>))
                                   <E x>)
             of {<x>}{<d>}(<T>, <D x d>) =>
                                  (<lam' T>, <eqLam [x] [d] D x d>)
          end
       | <app E1 E2> =>
          let
            val (<T1>,<D1>) = toDebruijn (* <P> *) W <E1>
            val (<T2>,<D2>) = toDebruijn (* <P> *) W <E2>
          in
            (<app' T1 T2>, <eqApp D1 D2>)
          end
      | <x#> =>
         let
           val (<Y>, <D>) = W <x>
         in
           (<var' Y>, <D>)
         end);
```

```
(* de Bruijn to HOAS (Example 4.6.2 and 5.4.6) *)
(* We comment out the first argument because it is inferable
 * and we do not want it in the termination order *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun toHOAS : (* <P:nat> -> *)
             (<X:variable P> -> <E:exp> * <equiv P E (var' X)>)
             -> <T:term P> -> <E:exp> * <equiv P E T>
  = (* fn <P> => *)
     fn W =>
       (fn <lam' T> =>
            (case ({<x:exp>}{<d:equiv (s P) x (var' one)>}
                    toHOAS (* <s P> *)
                           ((fn  {<x>}{<d>} (<one> => (<x>, <d>))
                              |  {<x>}{<d>} (<succ X> =>
                                     (let
                                        val (<E>, <D>) = W <X>
                                      in
                                        {<x>}{<d>} (<E>, <eqVar D>)
                                      end) \x \d)
                            ) \x \d)
                           <T>)
                of {<x>}{<d>} (<E x>, <D x d>) =>
                                  (<lam [x] E x>, <eqLam [x] [d] D x d>))
        | <app' T1 T2> =>
            let
                val (<E1>,<D1>) = toHOAS (* <P> *) W <T1>
                val (<E2>,<D2>) = toHOAS (* <P> *) W <T2>
             in
                 (<app E1 E2>, <eqApp D1 D2>)
             end
        | <var' X>  => W <X>);
```

```
(* Sample conversions (Example 4.6.3) *)
val test1 : (<T:term z> * _)
          = toDebruijn (* <z> *) (fn .)  <lam [x] lam [y] app x y>;
(* OUTPUT:
 * val test1 : <T : term z> *
 *                 <equiv z (lam ([x:exp] lam ([y:exp] app x y))) T>
 *     = <lam' (lam' (app' (var' (succ one)) (var' one)))> ,
 *          <eqLam
 *              ([x:exp] [d:equiv (s z) x (var' one)]
 *                  eqLam
 *                     ([x1:exp] [d1:equiv (s (s z)) x1 (var' one)]
 *                         eqApp (eqVar d) d1))>
 *)
val test2 = let
             val (<T>,<_>) = test1
             val (<E>,<_>) = toHOAS (* <z> *) (fn .) <T>
            in
             <E>
            end;
(* OUTPUT:
 * val test2 : <exp>
 *     = <lam ([x:exp] lam ([x1:exp] app x x1))>
 *)
```

6.10.3 Advanced Example Proving `equiv` is Deterministic

In this section we show our advanced case study that proves that `equiv` is deterministic. We will load `depDelphin.d` (Section 6.10.2) so that we have access to the datatypes it declared. This code is available on our website as `advancedEquiv.d`.

```
use "depDelphin.d";

(* These next two relations allow us to state that
 * terms/exps are equal (identical). *)
sig <eqexp : exp -> exp -> type>
    <eqexpid : eqexp E E>;

sig <eqterm : term N -> term N -> type>
    <eqtermid : eqterm T T>;

(* It is useful to have an empty type to argue about
 * something being impossible. *)
sig <empty : type>;

(* Standard less-than relation
 * (we will use it to talk about context extensions) *)
sig <lt : nat -> nat -> type>
    <lt_b : lt M (s M)>
    <lt_ind : lt M N -> lt M (s N)> ;

(* ********************************************************
 * We now show that (equiv P E T) is deterministic.
 * Which allows us to conclude:
 *
 * PART A:  toHOAS (toDebruijn E) = E
 *   This is shown by proving
 *     <equiv P E1 T> -> <equiv P E2 T> -> <eqexp E1 E2>
 *        (as fun detA-closed)
 *
 * PART B:  toDebruijn(toHOAS T) = T
 *   This is shown by proving
 *     <equiv P E T1> -> <equiv P E T2> -> <eqterm T1 T2>
 *        (as fun detB-closed)
 * ******************************************************** *)
```

```
(*
 * ********************************************
 * (PART A)
 ********************************************
 *)
(* The scope of params applies to all fun declarations until
 * the next params.
 *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun equalApp : <eqexp E1 E1'> -> <eqexp E2 E2'>
               -> <eqexp (app E1 E2) (app E1' E2')>
         = fn <eqexpid> <eqexpid> => <eqexpid> ;

fun equalLam : <{x:exp} eqexp (E1 x) (E1' x)>
               -> <eqexp (lam [x] E1 x) (lam [x] E1' x)>
         = fn <[x] eqexpid> => <eqexpid> ;

fun lessProp2 : <lt (s M) N> -> <lt M N>
    =  fn <lt_b> => <lt_ind lt_b>
        | <lt_ind L> => let
                          val <P> = lessProp2 <L>
                        in
                          <lt_ind P>
                        end;

fun lessProp1 : <lt (s M) (s N)> -> <lt M N>
     = fn <lt_b> => <lt_b>
        | <lt_ind L> => lessProp2 <L> ;

fun lessContra2 : <M:nat> -> <lt (s M) M> -> <empty>
     = fn <s N> <L> => lessContra2 <N> (lessProp1 <L>);

fun lessContra : <lt M M> -> <empty>
     = fn <lt_ind L> => lessContra2 <_> <L>;

fun lessProp : <N:nat> -> <lt (s M) N> -> <lt M N>
    = fn <s N> <L> =>  <lt_ind> @ (lessProp1 <L>);
```

```
type detA-World = <equiv (s P) E (var' one) #>
                  -> <equiv (s P) E' (var' one) #>
                  -> <eqexp E E'>;

fun extend : (<P':nat> -> <lt P (s P')> -> <E:exp>
                       -> <(equiv (s P') E (var' one))#> -> <empty>)
              -> detA-World
              -> {<x>}{<d: equiv (s P) x (var' one)>} detA-World
  = fn C => fn W
     => fn {<x>}{<d:equiv (s P) x (var' one)>} (<d> <d> => <eqexpid>)
         | {<x>}{<d:equiv (s P) x (var' one)>} (<d> <D#> =>
                                  let
                                    val <bot> = C <P> <lt_b> <_> <D>
                                  in
                                    {<x>}{<d>} (fn .) <bot>
                                    (* (fn .) is a total function of
                                     * type <empty> -> ...
                                     *)
                                  end \x \d)
         | {<x>}{<d:equiv (s P) x (var' one)>} (<D#> <d> =>
                                  let
                                    val <bot> = C <P> <lt_b> <_> <D>
                                  in
                                    {<x>}{<d>} (fn .) <bot>
                                    (* (fn .) is a total function of
                                     * type <empty> -> ...
                                     *)
                                  end \x \d)
         | {<x>}{<d:equiv (s P) x (var' one)>} (<D1#> <D2#> =>
                                  let
                                    val <R> = W <D1> <D2>
                                  in
                                    {<x>}{<d>}<R>
                                  end \x \d);

fun detA-Var : detA-World -> <equiv P E (var' X)>
               -> <equiv P E' (var' X)> -> <eqexp E E'>
 = fn W <D1 : (equiv _ E (var' one))#> <D2#> => W <D1> <D2>
    | W <eqVar D1> <eqVar D2> => detA-Var W <D1> <D2> ;
```

```
fun detA : (<P':nat> -> <lt P (s P')> -> <E:exp>
                    -> <(equiv (s P') E (var' one))#> -> <empty>)
         -> detA-World
         -> <equiv P E T> -> <equiv P E' T> -> <eqexp E E'>
 = fn C W <eqApp D1 D2> <eqApp D1' D2'> =>
           let
             val <P1> = detA C W <D1> <D1'>
             val <P2> = detA C W <D2> <D2'>
           in
             equalApp <P1> <P2>
           end
    | C W <eqLam [x] [d] D1 x d> <eqLam [x] [d] D2 x d> =>
           let
             val Cnew : (<P':nat> -> <lt (s P) (s P')> -> <E:exp>
                        -> <(equiv (s P') E (var' one))#> -> <empty>)
                      = fn <P'> <L:lt (s P) (s P')> <E3> <D3#> =>
                             C <P'> (lessProp <s P'> <L>) <E3> <D3>
           in
             (case {<x>}{<d: equiv (s P) x (var' one)>}
                    detA (Cnew with <P> <L:lt (s P) (s P)> <x> <d> =>
                                    lessContra <L>)
                         ((extend C W) \x \d)
                         <D1 x d>
                         <D2 x d>
                of {<x>}{<d>}<EQ x> => equalLam <[x] EQ x>)
           end
    | C W <D1 : equiv P E (var' X)> <D2 : equiv P E' (var' X)> =>
             detA-Var W  <D1> <D2>;

(* Putting it all together *)
params = .;
fun detA-closed  : <equiv P E1 T> -> <equiv P E2 T> -> <eqexp E1 E2>
                 = fn <D1> <D2> => detA (fn .) (fn .) <D1> <D2>;
```

```
(*
 * ********************************************
 * (PART B)
 ********************************************
 *)
params = <nd A>, <comb A>, <exp>, <equiv (s P) (E#) (var' one)>;
fun equalApp : <eqterm T1 T1'> -> <eqterm T2 T2'>
               -> <eqterm (app' T1 T2) (app' T1' T2')>
         = fn <eqtermid> <eqtermid> => <eqtermid> ;

fun equalLam : <eqterm T1 T1'> -> <eqterm (lam' T1) (lam' T1')>
         = fn <eqtermid> => <eqtermid> ;

fun equalBvar : <eqterm (var' X) (var' Y)>
                -> <eqterm (var' (succ X)) (var' (succ Y))>
         = fn <eqtermid> => <eqtermid> ;

type detB-World = <equiv P E (var' X) #> -> <equiv P E (var' Y) #>
                  -> <eqterm (var' X) (var' Y)>;

type detB-Contradiction = <lt (s P') (s P)>
                          -> <equiv (s P) E (var' one) #>
                          -> <equiv (s P') E (var' one) #>
                          -> <empty>;

fun detB-Var : <X: variable P> -> <equiv (s P) E (var' (succ X))>
               -> <P':nat> * <lt (s P') (s P)>
                   * <equiv (s P') E (var' one) #>
     = fn <one> <eqVar D#> => (<_>, <lt_b>, <D>)
        | <succ X> <eqVar D> =>
              let
                val (<P'>, <L'>, <D'>) = detB-Var <X> <D>
              in
                (<P'>, <lt_ind L'>, <D'>)
              end;

fun contraApp : <P:nat> -> <equiv P (app E1 E2) (var' X)> -> <empty>
                    = fn <s P> <eqVar D> => contraApp <P> <D>;

fun contraLam : <P:nat> -> <equiv P (lam E) (var' X)> -> <empty>
                  = fn <s P> <eqVar D> => contraLam <P> <D>;
```

```
fun detB : detB-Contradiction -> detB-World
            -> <equiv P E T1> -> <equiv P E T2> -> <eqterm T1 T2>
 = fn C W <eqApp D1 D2> <eqApp D1' D2'> =>
          let
            val <P1> = detB C W <D1> <D1'>
            val <P2> = detB C W <D2> <D2'>
          in
            equalApp <P1> <P2>
          end
    | C W <eqLam [x] [d] D1 x d> <eqLam [x] [d] D2 x d> =>
          (case {<x>}{<d: equiv (s P) x (var' one)>}
                    detB (C with <L> <d> <d> => lessContra <L>)
                         (W with <d> <d> => <eqtermid>)
                         <D1 x d>
                         <D2 x d>
                     of {<x>}{<d>}<EQ> => equalLam <EQ>)
    | C W <D1#> <D2#> => W <D1> <D2>
    | C W <eqVar (D1 : equiv P E (var' X))>
          <eqVar (D2 : equiv P E (var' Y))> =>
            equalBvar (detB C W <D1> <D2>)
    | C W <D1 : (equiv (s P) E (var' one))#>
          <eqVar (D2 : equiv P E (var' X))> =>
          let
            val (<P'>, <L'>, <D2'>) = detB-Var <X> <eqVar D2>
            val <bot> = C <L'> <D1> <D2'>
          in
            (fn .) <bot>
            (* (fn .) is a total function of
             * type <empty> -> ...
             *)
          end
    | C W <eqVar (D1 : equiv P E (var' X))>
          <D2 : (equiv (s P) E (var' one))#> =>
          let
            val (<P'>, <L'>, <D1'>) = detB-Var <X> <eqVar D1>
            val <bot> = C <L'> <D2> <D1'>
          in
            (fn .) <bot>
            (* (fn .) is a total function of
             * type <empty> -> ...
             *)
          end
```

```
    (* The next four cases handle impossible cases.  We prove these
     * cases are impossible by using contraApp and contraLam
     * which return an object of type empty, of which there are none.
     *
     * Given an E : <empty>, we can conclude anything, which
     * is why we just do:  (fn .) E
     * The type of (fn .) will be <empty> -> T.
     *)
    | C W  <eqVar (D : equiv P (app E1 E2) (var' X))> <_> =>
           (fn .) (contraApp <P> <D>)
    | C W  <_> <eqVar (D : equiv P (app E1 E2) (var' X))> =>
           fn .) (contraApp <P> <D>)
    | C W <eqVar (D : equiv P (lam E) (var' X))> <_> =>
           (fn .) (contraLam <P> <D>)
    | C W <_> <eqVar (D : equiv P (lam E) (var' X))> =>
           (fn .) (contraLam <P> <D>);

(* Putting it all together *)
params = .;
fun detB-closed : <equiv P E T1> -> <equiv P E T2> -> <eqterm T1 T2>
               = fn <D1> <D2> => detB (fn .) (fn .) <D1> <D2>;
```

Chapter 7

Related Work

The Delphin system is novel in its support of programming and reasoning over data utilizing both HOAS and dependent types. The driving force of our work is in our ∇ type which allows one to elegantly work with higher-order data. In this chapter we outline related work with respect to the ∇ type (Section 7.1), representing bindings (Section 7.2), dependent types (Section 7.3), multi-stage programming (Section 7.4), and Twelf (Section 7.5). We will conclude with avenues of future work in Section 7.6.

7.1 The ∇ Type

Our choice of using the symbol ∇ expresses our relation to Miller and Tiu's (2005) ∇ quantifier where they define a proof theory distinguishing between eigenvariables intended for instantiation from those representing scoped constants. In their logic, the formula $(\forall \boldsymbol{x}.\ \forall \boldsymbol{y}.\ \tau(\boldsymbol{x},\ \boldsymbol{y})) \supset \forall \boldsymbol{z}.\ \tau(\boldsymbol{z},\ \boldsymbol{z})$ is provable, whereas $(\nabla \boldsymbol{x}.\ \nabla \boldsymbol{y}.\ \tau(\boldsymbol{x},\ \boldsymbol{y})) \supset \nabla \boldsymbol{z}.\ \tau(\boldsymbol{z},\ \boldsymbol{z})$ is not. Similarly in Delphin, the former type is inhabited by fn $f \mapsto$ fn $\boldsymbol{z} \mapsto f\ \boldsymbol{z}\ \boldsymbol{z}$, while the latter type is generally not inhabited because nothing is known about $\tau(\boldsymbol{z},\ \boldsymbol{z})$ from $\tau(\boldsymbol{x},\ \boldsymbol{y})$. One important distinction between our ∇ quantifiers is that their system used the ∇ quantifier not to reason over formulas,

but to reason over pairs of formulas and contexts, where this local context contained the parameters that can occur free. In Delphin, there are no local contexts and parameters are placed in the main context, which we find renders it more useful for programming.

Gacek et al. (2008b) developed a variant of this earlier system that gets rid of local contexts in favor of a global collection of parameters. Most importantly, they also allow *definitions* to use ∇. This is the core logic of the Abella (Gacek et al. 2008a) interactive proof assistant. They do not have dependent types, but one may encode a judgment using Hereditary Harrop Formulas (a generalization of Horn clauses). Encodings may utilize HOAS and often look very similar to judgments encoded in LF. Given a judgment expressed as Hereditary Harrop Formulas, they use definitions to give the user access to a predicate representing the derivability of the judgment. Therefore, reasoning over judgments corresponds to reasoning over this predicate. The resulting predicate makes the context explicit requiring the user to often prove properties about the context, for which the user often uses definitions involving ∇. In Delphin, reasoning over judgments keeps the context implicit and we have seen that parameter functions can be used to express complex properties of the implicit context. Their ability to extend the meta-logic with user-specified definitions is a very powerful tool as evidenced by an example normalization proof using a logical relation encoded as a definition. One method of writing a proof by logical relations in Delphin is to define the logical relation in an intermediate logic which is encoded in LF, following the proof technique of *structural logical relations* (Schürmann and Sarnat 2008).

7.2 Representing Bindings

The underlying idea behind our work addresses the need to provide a principled way to represent bindings. As a motivating example we will discuss various ways to represent untyped λ-expressions $e ::= x \mid \lambda x.\ e \mid e_1\ e_2$ as terms of type **exp**.

7.2.1 Standard First-Order Approaches

The standard first-order technique would represent the untyped λ-calculus with three constructors: **var**, **lam**, and **app** to respectively encode variables, abstractions, and applications. The simplest way to represent variables would be to use strings. The main difficulties working with such encodings are (1) the need to write routine and tedious functions to perform capture-avoiding substitutions and (2) the need to manually reason and implement tools to check if terms are α-equivalent (i.e equivalent modulo variable renamings).

One can write a first-order encoding with built-in α-equivalence by using de Bruijn indices (de Bruijn 1972), where a variable is represented as a number pointing to its binding. However, one must still implement substitutions, and the operations on these indices are also tedious and error-prone. Additionally, weakening properties are made more complicated. For example, adding a superfluous binder to the middle of a term requires manipulation of all the indices. An important detriment to de Bruijn indices is that the resulting terms are harder to read than the string alternative.

Much work has been done on utilizing first-order encodings, including combining the above two approaches (Aydemir et al. 2008). Another approach is to use de Bruijn indices but supply a nice interface (or library of functions) to create and manipulate them, as in the Hybrid approach (Momigliano et al. 2008); this is sometimes called *lightweight* HOAS.

7.2.2 Nominal First-Order Approaches

Besides using names and numbers, there has been a great deal of work on nominal systems based on a notion of freshness (Gabbay and Pitts 2001), which utilizes Fraenkel-Mostowski (FM) set theory to provide a built-in α-equivalence relation for first-order encodings. This is achieved by representing variables using a special *atom* type. One must still write substitution functions, but they are much easier to write than in the previous first-order approaches.

This approach has made it into proof systems as well as programming languages. The Isabelle proof system (Wenzel and Berghofer 2008) now includes a nominal datatype package (Urban 2008). FreshML (Pitts and Gabbay 2000) is a programming language based on providing this nominal support, but it does not offer the reasoning capabilities afforded in Delphin. Besides not being dependently typed, the creation of *atoms* (which corresponds to our parameters) is a global effect in the sense that once created, its scope is forever. Pottier (2007) has developed a logic for reasoning about values and the names they contain in FreshML allowing one to conclude that a name does not escape its scope, which is an inherent property of Delphin's type system (Corollary 5.5.20). Using a notion of freshness based on FM set theory, Schöpp and Stark (2004) have designed a dependent type theory (rooted in category theory) where dependencies over constants are captured by *bunched* dependent types, but there does not exist any prototype implementation.

Recently, Cheney (2008) has developed a nominal type theory with a special notion of *name abstractions*. They introduce a new type which is quite similar to our ∇ type, with a similar introduction and elimination rule. Their system does not support dependent types, but they defer that to future work.

7.2.3 Weak HOAS

The term *weak* HOAS refers to a restricted form of HOAS where abstractions are not over object-level expressions but instead over a separate (usually variable) type. It is best to illustrate with an example. The typical HOAS encoding of the untyped λ-calculus (as discussed in Chapter 1) has two constructors and appears as:

```
data exp where
    lam :: (exp -> exp) -> exp
    app :: exp -> exp -> exp
```

In weak HOAS, we will alternatively have three constructors:

```
data exp where
    lam :: (variable -> exp) -> exp
    app :: exp -> exp -> exp
    var :: variable -> exp
```

In HOAS, we get substitution for *free* as object-level substitutions can be implemented via meta-level function application. In weak HOAS, one must still write their own substitution function, but it is much easier to write than in the first-order approaches (similar to the nominal approach).

Early work (Despeyroux et al. 1995) proposed using a notion of weak HOAS in Coq (Dowek et al. 1993) where the type **variable** corresponds to natural numbers. However, as they use the Coq function space, this leads to an inadequate encoding due to the existence of exotic terms (Chapter 1). They throw out exotic terms by defining an appropriate predicate.

Similar work (Honsell et al. 2001) used weak HOAS to encode the π-calculus directly in the Coq system, which turned out quite natural as bindings in the π-calculus were over a separate class of names.

Work has also proceeded in the same fashion with respect to the Isabelle proof system. In this setting, Momigliano et al. (2003) provide a combinator for Primitive recursion over weak HOAS using a predicate to throw away exotic terms.

Instead of using a predicate to remove exotic terms, a new version of weak HOAS called *parametric* HOAS (Chlipala 2008) achieved a similar result utilizing polymorphism.

7.2.4 (Full) HOAS

With respect to full HOAS, Washburn and Weirich (2008) show how one can use HOAS in a conventional programming language with polymorphism, in particular Haskell. As we motivated in Chapter 1 the use of the Haskell function space for HOAS leads to exotic terms, but they explicitly address different ways one may encode their data to remove exotic terms. However, instead of dealing with exotic terms, we will now turn to other systems providing an inherent way to adequately represent data utilizing HOAS.

The inherent support of HOAS has been the subject of a great deal of research. Our previous work also employed the design of a two level system separating representation-level functions from computation-level functions. The ∇-calculus (Schürmann et al. 2005, 2004) provided a stack based system supporting a simply-typed logical framework. In this system we did not distinguish parameters as having different types. The ∇ was not a type as it is in Delphin, but rather a method of declaring a pattern variable to only match parameters, i.e. the ∇ is equivalent to our ϵ over $\boldsymbol{A}^{\#}$. This system was implemented as Elphin and is available on the Delphin website (`http://www.delphin.logosphere.org/`), but it is now superseded by Delphin. The Delphin system is influenced by, yet completely different from Elphin. Our earlier attempts to extend Elphin with dependent types led to a terribly

complex system which was completely impractical and we therefore abandoned this approach.

The direct predecessor to Delphin is our work utilizing temporal logic (Poswolsky 2006) to reason over LF (with dependent types). This work used a past-time temporal modality to represent derivations occurring in the past, i.e. outside the scope of certain parameters. However, we hit quite a number of roadblocks trying to implement this system. Instead, our work in Delphin uses a well-founded ∇ type to represent expressions that make sense with additional parameters. In other words, we look into the future instead of the past.

Our earlier work has also inspired Westbrook's λ^{FV} (Westbrook 2006). This calculus is designed to include coverage checking in the type system. Work on λ^{FV} is still ongoing, including a proof of type safety.

The core (Chapter 4) of the Delphin system was published (Poswolsky and Schürmann 2008) with a corresponding technical report (Poswolsky and Schürmann 2007).

The problem of programming over HOAS data is actively being pursued on another frontier. Pientka (2008) presents a system to write programs over a simply-typed logical framework supporting HOAS. They use an enriched representation level rooted in *contextual modal type theory* (Nanevski et al. 2008) so that one may encode data using HOAS but then also talk about it with a uniform notion of contexts and substitutions. This means that one can use HOAS to elegantly represent data, and then work with such data in a first-order manner, but with the benefits of having the notion of substitution and contexts provided automatically instead of having to define them separately for each object language. They introduce *context variables* and have a dependent function space that ranges over contexts. Function's range over types of the form $\boldsymbol{A}[\boldsymbol{\Gamma}]$ where $\boldsymbol{\Gamma}$ contains the parameters. Their technique requires

the explicit handling of first-class substitutions and contexts. They have named their system Beluga (Pientka et al. 2008) and a prototype implementation is available, but limited only to simple types.

Pientka's work has been extended to support a logical framework with dependent types (Pientka and Dunfield 2008). No implementation of a dependently-typed Beluga is currently available, but it is under development. They motivate their paper with a similar example we have used throughout this dissertation: converting natural deduction derivations into combinators. The first step of this translation requires a function to remove abstractions with respect to combinators, called bracket abstraction (Example 4.5.2). They present their equivalent of this function but stop short of supplying the main function that converts natural deduction derivations into combinators (Example 4.5.3). This function is very difficult to write in their current system and they defer some issues to future work. One important difference between their system and Delphin is that that they lack the ability to write parameter functions, which we used in that example. The authors are currently investigating the ability and need of adding parameter functions to their system.

Although similar in goal, the work on Beluga takes a fundamentally different approach than Delphin. Programming in Beluga requires treating higher-order data in a first-order manner, while the goal of Delphin is to allow one to work with HOAS while staying in a higher-order mindset; we do not make first-order notions of context and substitutions explicit. However, recall that LF is a very powerful system for representation. If one requires more fine-grained control over the context, then one may also represent their data with a context (where the context is represented in LF). Crary (2008) has shown how one may take data encoded using HOAS and convert it to data encoded with explicit contexts and back again. This technique allows one to work with explicit contexts in Delphin, if so desired.

Delphin is designed with two levels cleanly separating representation-level functions from computation-level ones. Earlier work (Schürmann et al. 2001) presented a one level system utilizing a modal operator where $\Box \boldsymbol{A}$ stood for *closed* objects of type $\boldsymbol{A}$ and computation was performed over this type using iterators as well as a separate notion of case analysis. This system was limited to simple types and a variant was extended to support dependent types (Despeyroux and Leleu 1999). However, the authors of the latter were dissatisfied with the restrictions they needed to impose on their iterator (and case) construct and additionally deferred a proof of strong normalizing to future work. Neither of these systems have made it to the implementation stage.

The one-level systems above utilized a modality to enforce that representation-level functions did not depend on computation. Recent work on focusing (Licata et al. 2008) provides a principled way to combine the two. They provide a *positive* function space and a *negative* function space where the former corresponds to representation-level functions and the latter corresponds to computation-level functions. Their rules carefully distinguish between positive and negative constructs such that one can adequately represent bindings using the *positive* function space. Their work addresses how to combine the two function spaces and abstracts away from the details of actually providing and working with a positive function space. We believe that the techniques of Delphin are well-suited for a concrete implementation of their system.

7.3 Dependent Types

The Delphin calculus is not the first to attempt to combine dependent types with functional programming. Dependent ML (Xi and Pfenning 1999) allows types to

be indexed by integers, which has been generalized to form the ATS/LF system (Xi 2008). Haskell has also been recently extended with Generalized Algebraic Datatypes, or GADTs (Kennedy and Russo 2005; Jones et al. 2006) (as discussed in Chapter 1). The Ωmega (Sheard 2005) system also uses GADTs. The Cayenne (Augustsson 1998) language supports full dependent types and even computation with types, rendering it more expressive but at the expense of an undecidable type checker. Agda (Norell 2007) and Epigram (McBride and McKinna 2004) are two more languages inspired by dependent type theories. Even with respect to imperative programming, dependent types can prove useful (Westbrook et al. 2005).

All but the ATS/LF system lack support for HOAS. Unlike Delphin, the types in ATS/LF can only be dependent on objects of a non-dependent type, i.e. types can be indexed by natural numbers but not by derivations. Additionally, although ATS/LF supports HOAS, one must resort to encoding the context explicitly, or as they say representing terms as *terms-in-contexts* (Donnelly and Xi 2005). By making this information explicit one can perform limited reasoning about parameters in the context, but substitutions must be explicitly defined. We suspect that they can also add a ∇ type similar to ours.

7.4 Multi-Stage Programming

Our design of Delphin features a distinction between two levels – the representation level (LF) and the computation level. The concept of "levels" exhibits some similarities to systems of multi-stage programming (Taha 2004) such as MetaML and MetaOCaml. The purpose of these systems is to facilitate the writing of programs that alter, run, and specialize code; they are used to implement staged computation and partial evaluation. Therefore, these systems provide a means to go under func-

tions and perform computation on a new level. However, unlike Delphin, every level has the same function space. MetaML programs can go under functions and return new MetaML functions. In contrast, the "levels" in Delphin are different function spaces and this allows us to go under LF-level λ-binders and perform case analysis over the result; a technique that is crucial for our work and would not work in their systems.

Additional work, inspired by FreshML, has added a notion of names to a modal logic yielding a meta-programming language comparable to MetaML but with finer control as free variables are represented as atoms (Nanevski 2002). However, this system is not designed for working with higher-order representation techniques, but just staged computation.

7.5 Twelf

LF is well suited for *representation* but does not directly afford the ability to reason over representations. Twelf (Pfenning and Schürmann 1998) utilizes a logic programming methodology to conduct such reasoning by providing meta-level constructs to interpret a relation (type family) as a function. Twelf provides a notion of *mode* to distinguish the intended input/output behavior of relations.

Therefore, closely influencing our work is the underlying meta-logic of Twelf, $\mathcal{M}_2^+$ (Schürmann 2000), which is *not* higher order. Reasoning about parameters in Delphin is done via parameter functions, which requires higher-order functions. Alternatively, $\mathcal{M}_2^+$ has a built-in notion of *blocks*, which are much weaker than our parameter functions (a comparison can be found in Section 5.1). For example, our functions converting between a de Bruijn and HOAS encoding of untyped λ-expressions use higher-order functions whose behavior cannot be captured using Twelf blocks.

We envision that Delphin will eventually replace the underlying meta-logic of Twelf.

Finally, one detriment of writing logic programs is that there is no correspondence to inner functions. In Delphin, it is straightforward to perform case analysis on the output of a function call. A corresponding logic program in Twelf would provide identical clauses differing only in the output position of the relation. Logic programming utilizes backtracking, so it will execute successfully, but the Twelf system will not be able to reason whether it is covered. The issue of refining the output is known as *output coverage* which proves indispensable in practice. Twelf is implemented with limited support of output coverage where it allows one to refine the output only if there is exactly one possibility, but this has never been formalized. In Twelf, reasoning about coverage resorts to reasoning over programs converted into $\mathcal{M}_2^+$. Work on factoring (Poswolsky and Schürmann 2003; Poswolsky 2003) enhanced the power of Twelf by merging multiple clauses together, utilizing nested functions, allowing the coverage checker to detect that programs were covered even when different cases were initially spread across different clauses.

However, there is no corresponding issue in Delphin. The totality of a function in Twelf corresponds to converting a logic program into a corresponding function in $\mathcal{M}_2^+$ and checking coverage on the latter. In Delphin, we are checking the actual function one writes. The issue of factoring came from the inability to write inner functions utilizing a logic programming paradigm while in Delphin one can easily perform case analysis on the results of function calls, as we have done in our many examples.

7.6 Future Work

The Delphin programming language has been implemented and is at a stage where it can serve as a replacement for writing proofs in Twelf. However, there are still many avenues for future work. In this section, we outline the major avenues of future work.

7.6.1 Termination

Similar to Twelf, we have implemented a termination checker that checks that recursive calls get smaller with respect to lexical extensions of the subterm ordering defined by Rohwedder and Pfenning (1996). However, more powerful orderings may also be used which would increase the proof-theoretic strength of Delphin. Additionally, a detailed formalization of termination is deferred to future work.

7.6.2 Interactive Proof System

When using Delphin to write proofs, it would be useful to allow the programmer to construct the proof (or function) in an interactive manner. Delphin code is written in a standard text editor, which means that it is the programmer's job to perform case analysis and the typechecker's job to check that the provided cases are exhaustive. In an interactive system, the programmer can be prompted for each case simplifying the programmer's task. Additionally, tactics can be built-in as to allow the automatic completion of simple steps.

The result of an interactive proof system will create a well-typed Delphin function which one could print and execute.

7.6.3 Twelf to Delphin

When using Twelf, one *represents* a function as a relation in LF, which is converted into a function in the underlying meta-logic $\mathcal{M}_2^+$. The actual coverage analysis is on the result of the conversion, which may appear mysterious to some.

We would advocate that the function is initially written in Delphin, but it would also be useful to convert the LF relation into Delphin instead of $\mathcal{M}_2^+$. In fact, we expect that $\mathcal{M}_2^+$ will disappear altogether as we replace the underlying logic of Twelf with Delphin.

7.6.4 More Programming Features

With respect to programming, there are many more constructs that can and should be added to Delphin. We believe it is straightforward to add mechanisms for providing Input/Output (I/O) capabilities as well as references (or monads). As Delphin is implemented in Standard ML, we can provide mechanisms to allow the programmer access to the I/O library of ML. We expect that references can easily be added as a computational-type τ; as types cannot be dependent on expressions of type τ, there will be no issue with dependencies.

7.6.5 Examples / Features of Parameters

The Delphin website (`http://www.delphin.logosphere.org/`) provides many examples, and we hope it will continue to grow. We have seen how the dynamic creation of parameters facilitates working with data encoded using HOAS, but there may be many other avenues where our ν construct would prove useful.

In Chapter 6 we discussed an example of using parameters to implement a Hindley-Milner style type-inference algorithm. In this example, a fresh parameter

was interpreted as a new "memory location" and we passed around a function that mapped "memory locations" to data. Therefore, we used parameters to create our own form of references. A detailed investigation of this use of ν as well as others would be very interesting. One interesting property of using ν in this manner was that our type system also guaranteed that we *freed* all memory locations.

Chapter 8

Conclusion

In this dissertation we presented the Delphin programming language. We started by motivating the usefulness of representing data using dependent types and HOAS. We have shown that our language provides foundational mechanisms to manipulate and reason over such complex data, which was our thesis.

Chapter 2 presented the datatypes used in our examples as well as discussed the properties and features of Delphin's representation level, which is the Edinburgh Logical Framework LF (Harper et al. 1993).

Chapters 3 and 4 presented the core semantics of Delphin and illustrated how one may use Delphin to perform computation over higher-order encodings. In order to support adequate encodings utilizing HOAS, we designed a two level system cleanly separating representation-level functions from computation-level functions. The challenge of writing recursive functions over higher-order data comes from the need to recurse under representation-level functions. We achieved this ability by creating a *new* abstraction, $\nu \boldsymbol{x}.\ e$, which introduced the type $\nabla \boldsymbol{x}.\ \tau$. This abstraction differs from a typical functional abstraction in that e is executed while $\boldsymbol{x}$ remains uninstantiated. We refer to $\boldsymbol{x}$ as a parameter. Delphin pervasively distinguishes

parameters and we motivated the power and features of Delphin by (1) translating between derivations in different logics and (2) converting between higher-order and first-order encodings of untyped λ-expressions.

Chapter 5 extended the core system with the ability to check if a list of cases is exhaustive (coverage checking), which allowed us to prove type safety. Meta-level guarantees that functions are total allow one to interpret functions as proofs. We concluded by discussing how the totality of our motivating examples yield interesting proofs about logics and calculi.

Finally, Chapter 6 explained and illustrated the implementation of our system. The Delphin system is a complete programming language which can already be used as a replacement for writing proofs in Twelf. Delphin is the first, and currently the only, implemented functional system tackling programming over a logical framework with both higher-order encodings and dependent types.

The reader is invited to visit the Delphin website at `http://www.delphin.logosphere.org/`. On our website, one may download Delphin and also find all the examples throughout this dissertation and many more.

Bibliography

Lennart Augustsson. Cayenne - a Language with Dependent Types. In *International Conference on Functional Programming (ICFP 1998)*, pages 239–250, 1998.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15. ACM, 2008. ISBN 978-1-59593-689-9. doi: http://doi.acm.org/10.1145/1328438.1328443.

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

James Cheney. A simple nominal type theory. In *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP '08)*, Electronic Notes in Theoretical Computer Science (ENTCS), pages 90–104, Pittsburgh, PA, USA, June 2008. Elsevier.

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, Victoria, British Columbia, Canada, September 2008.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

Karl Crary. Explicit contexts in LF. In *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP '08)*, Electronic Notes in Theoretical Computer Science (ENTCS), pages 105–119, Pittsburgh, PA, USA, June 2008. Elsevier.

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. URL `http://dx.doi.org/10.1016/1385-7258(72)90034-0`.

Joëlle Despeyroux and Pierre Leleu. Primitive recursion for higher-order abstract syntax with dependant types. In *Informal proceedings of the FLoC '99 IMLA Workshop on Intuitionistic Modal Logics and Applications*, June 1999.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA 1995)*, volume 902 of *LNCS*, pages 124–138, Edinburgh, Scotland, April 1995. Springer-Verlag. Also appears as INRIA Research Report RR-2556.

Kevin Donnelly and Hongwei Xi. Combining higher-order abstract syntax with first-order abstract syntax in ATS. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 58–63, New York, NY, USA, 2005. ACM Press.

Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assis-

tant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. Rapport de Recherche 3591, INRIA, December 1998. Preliminary version appeared at JICSLP '96.

Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP '08)*, Electronic Notes in Theoretical Computer Science (ENTCS), pages 120–134, Pittsburgh, PA, USA, June 2008. Elsevier.

Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3-5):341–363, 2001.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. In *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP '08)*, Electronic Notes in Theoretical Computer Science (ENTCS), pages 75–89, Pittsburgh, PA, USA, June 2008a. Elsevier.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 33–44, Pittsburgh, PA, USA, June 2008b. IEEE Computer Society. URL `http://arxiv.org/pdf/0802.0865.pdf`.

Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, 2005. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1042038.1042041.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (Co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge Univ Press, 2000.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: http://doi.acm.org/10.1145/1159803.1159811.

Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 21–40, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094814.

B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood, New Jersey, second edition, 1988.

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-575-4. doi: http://doi.acm.org/10.1145/1190216.1190245.

Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 241–252, Pittsburgh, PA, USA, June 2008. IEEE Computer Society.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

David Madore. The Unlambda Programming Language, 2008. Unlambda homepage: `http://www.madore.org/~david/programs/unlambda/`.

Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796803004829.

Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, October 2005.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

Alberto Momigliano, Simon Ambler, and Roy Crole. A definitional approach to primitive recursion over higher order abstract syntax. In Alberto Momigliano and Marino Miculan, editors, *Proceedings of the Merlin Workshop*, Uppsala, Sweden, June 2003. ACM Press.

Alberto Momigliano, Alan J. Martin, and Amy P. Felty. Two-level hybrid: A system for reasoning using higher-order abstract syntax. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 196:85–93, 2008. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2007.09.019.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1352582.1352591.

Aleksander Nanevski. Meta-programming with names and necessity. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, Pittsburgh, PA, October 2002. ACM Press.

George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, France, January 1997. ACM Press.

Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL '08: Proceedings of the*

35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 371–382. ACM, 2008. ISBN 978-1-59593-689-9.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, July 2008.

Brigitte Pientka, Joshua Dunfield, and Renaud Germain. Beluga: Functional programming with higher-order abstract syntax, 2008. Beluga homepage: `http://www.cs.mcgill.ca/~complogic/beluga/`.

Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In *MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction*, pages 230–255, London, UK, 2000. Springer-Verlag. ISBN 3-540-67727-5.

Adam Poswolsky. Factoring report. Technical Report YALEU/DCS/TR-1256, Yale University, 2003.

Adam Poswolsky. A temporal-logic approach to functional calculi for dependent types and higher-order encodings. Technical Report YALEU/DCS/TR-1364, Yale University, 2006.

Adam Poswolsky and Carsten Schürmann. Factoring pure logic programs. Available at http://www.cs.yale.edu/homes/poswolsk/papers/factoring.pdf, 2003.

Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming (ESOP 2008)*, pages 93–107, Budapest, Hungary, 2008. ISBN 978-3-540-78738-9.

Adam Poswolsky and Carsten Schürmann. System description: Delphin – A functional programming language for deductive systems. In *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP '08)*, Electronic Notes in Theoretical Computer Science (ENTCS), pages 135–141, Pittsburgh, PA, USA, June 2008. Elsevier.

Adam Poswolsky and Carsten Schürmann. Extended report on Delphin: A functional programming language with higher-order encodings and dependent types. Technical Report YALEU/DCS/TR-1375, Yale University, 2007.

François Pottier. Static name control for FreshML. In *Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS 2007)*, Wroclaw, Poland, July 2007. URL `http://cristal.inria.fr/~fpottier/publis/fpottier-pure-freshml.ps.gz`.

Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *European Symposium on Programming (ESOP 1996)*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In *Computer Science Logic: Proceedings of the 18th International Workshop (CSL 2004)*, number 3210 in Lecture Notes in Computer Science, pages 235–249. Springer-Verlag, 2004. URL `http://www.inf.ed.ac.uk/~stark/names+binding.html`.

Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF.

In David Basin and Burkhart Wolff, editors, *Proccedings of Theorem Proving in Higher Order Logics (TPHOLs '03)*, volume LNCS-2758, Rome, Italy, 2003. Springer Verlag.

Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 69–80, Pittsburgh, PA, USA, June 2008. IEEE Computer Society.

Carsten Schürmann and Mark-Oliver Stehr. An executable formalization of the HOL/Nuprl connection in the metalogical framework twelf. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006)*, pages 150–166, Phnom Penh, Cambodia, 2006.

Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus. Functional programming with higher-order encodings. Technical Report YALEU/DCS/TR-1272, Yale University, October 2004.

Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus. Functional programming with higher-order encodings. In *Typed Lambda Calculus and Applications (TLCA 2005)*, pages 339–353, Nara, Japan, 2005.

Carsten Schürmann, Frank Pfenning, and Natarajan Shankar. Logosphere. a formal digital library, 2008. Logosphere homepage: `http://www.logosphere.org/`.

Tim Sheard. Putting curry-howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM. ISBN 1-59593-071-X. doi: http://doi.acm.org/10.1145/1088348.1088356.

Walid Taha. A gentle introduction to multi-stage programming. In Don Batory, Charles Consel, Christian Lengauer, and Martin Odersky, editors, *Domain-specific Program Generation (DSPG 2004)*, LNCS. Springer-Verlag, 2004. to appear.

Simon Thompson. *Haskell: The Craft of Functional Programming, Second Edition*. Addison-Wesley, 1999.

Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008. ISSN 0168-7433. doi: http://dx.doi.org/10.1007/s10817-008-9097-2.

Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism (extended version). *Journal of Functional Programming*, 18(1):87–140, January 2008.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Markus Wenzel and Stefan Berghofer. The Isabelle system manual, 2008. Available at `http://isabelle.in.tum.de/`.

Edwin Westbrook. Free Variable Types. In *Trends in Functional Programming (TFP 2006)*, Nottingham, UK, 2006.

Edwin M. Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *International Conference on Functional Programming (ICFP 2005)*, pages 268–279, Tallinn, Estonia, 2005.

Hongwei Xi. ATS/LF: a type system for constructing proofs as total functional programs. In Christoph Benzmüller, Chad Brown, Jörg Siekmann, and Rick Statman, editors, *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*, Studies in Logic and the Foundations of Mathematics. IFCoLog, 2008. To appear.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL '99)*, pages 214–227. ACM Press, January 1999. URL `http://www.cs.cmu.edu/~fp/papers/popl99.ps.gz`.

Appendix A

System Summary

A.1 LF Summary

A.1.1 Grammar

LF Types	$\boldsymbol{A}, \boldsymbol{B}$	::=	$\boldsymbol{a} \mid \boldsymbol{\Pi x{:}A.\ B} \mid \boldsymbol{A\ M}$
LF Objects	$\boldsymbol{M}, \boldsymbol{N}$	::=	$\boldsymbol{c} \mid \boldsymbol{x} \mid \boldsymbol{\lambda x{:}A.\ M} \mid \boldsymbol{M\ N}$
Signature	$\boldsymbol{\Sigma}$	::=	$\cdot \mid \boldsymbol{\Sigma, a{:}K} \mid \boldsymbol{\Sigma, c{:}A}$
Kinds	$\boldsymbol{K}$	::=	$\boldsymbol{type} \mid \boldsymbol{\Pi x{:}A.\ K}$
LF Context	$\boldsymbol{\Gamma}$	::=	$\cdot \mid \boldsymbol{\Gamma, x{:}A}$

A.1.2 Valid Signatures

$$\frac{}{\cdot\ \mathsf{sig}} \qquad \frac{\boldsymbol{\Sigma}\ \mathsf{sig} \quad \cdot \underline{\vdash}^{\mathsf{lf}}_{\boldsymbol{\Sigma}}\ \boldsymbol{A} : \boldsymbol{type}}{\boldsymbol{\Sigma, c{:}A}\ \mathsf{sig}} \qquad \frac{\boldsymbol{\Sigma}\ \mathsf{sig} \quad \cdot \underline{\vdash}^{\mathsf{lf}}_{\boldsymbol{\Sigma}}\ \boldsymbol{K} : \boldsymbol{kind}}{\boldsymbol{\Sigma, a{:}K}\ \mathsf{sig}}$$

A.1.3 Valid Contexts

$$\frac{}{\cdot\ \mathsf{ctx}_{\mathsf{lf}}}\ \mathsf{ctxLFEmpty} \qquad \frac{\boldsymbol{\Gamma}\ \mathsf{ctx}_{\mathsf{lf}} \quad \boldsymbol{\Gamma} \underline{\vdash}^{\mathsf{lf}}\ \boldsymbol{A} : \boldsymbol{type}}{\boldsymbol{\Gamma, x{:}A}\ \mathsf{ctx}_{\mathsf{lf}}}\ \mathsf{ctxLFAdd}$$

A.1.4 Typing

The signature $\boldsymbol{\Sigma}$ is fixed and implicit. We use $\equiv_{\alpha\beta\eta}$ to express equality between types and a formally defined judgment to determine this equality can be found in Harper and Pfenning (2005).

$$\frac{\Gamma \ \mathsf{ctx}_{\mathsf{lf}} \quad (c{:}A) \ \mathbf{in} \ \Sigma}{\Gamma \vdash^{\mathsf{lf}} c : A}\ \mathsf{LF\_ObjConst} \qquad \frac{\Gamma \ \mathsf{ctx}_{\mathsf{lf}} \quad (x{:}A) \ \mathbf{in} \ \Gamma}{\Gamma \vdash^{\mathsf{lf}} x : A}\ \mathsf{LF\_Var}$$

$$\frac{\Gamma, x{:}A \vdash^{\mathsf{lf}} M : B \quad \Gamma \vdash^{\mathsf{lf}} A : \mathit{type}}{\Gamma \vdash^{\mathsf{lf}} \lambda x{:}A.\ M : \Pi x{:}A.\ B}\ \mathsf{LF\_Lam} \qquad \frac{\Gamma \vdash^{\mathsf{lf}} M : \Pi x{:}A.\ B \quad \Gamma \vdash^{\mathsf{lf}} N : A}{\Gamma \vdash^{\mathsf{lf}} M\ N : B[\mathrm{id}_\Gamma, N/x]}\ \mathsf{LF\_App}$$

$$\frac{\Gamma \vdash^{\mathsf{lf}} M : A \quad A \equiv_{\alpha\beta\eta} B}{\Gamma \vdash^{\mathsf{lf}} M : B}\ \mathsf{LF\_EqualType}$$

. . . . . .

A.1.5 Kinding

The signature $\boldsymbol{\Sigma}$ is fixed and implicit. We use $\equiv_{\alpha\beta\eta}$ to express equality between kinds and a formally defined judgment to determine this equality can be found in Harper and Pfenning (2005).

$$\frac{\Gamma \ \mathsf{ctx}_{\mathsf{lf}} \quad (a{:}K) \ \mathbf{in} \ \Sigma}{\Gamma \vdash^{\mathsf{lf}} a : K}\ \mathsf{LF\_TypeConst} \qquad \frac{\Gamma, x{:}A \vdash^{\mathsf{lf}} B : \mathit{type} \quad \Gamma \vdash^{\mathsf{lf}} A : \mathit{type}}{\Gamma \vdash^{\mathsf{lf}} \Pi x{:}A.\ B : \mathit{type}}\ \mathsf{LF\_Pi}$$

$$\frac{\Gamma \vdash^{\mathsf{lf}} A : \Pi x{:}B.\ K \quad \Gamma \vdash^{\mathsf{lf}} M : B}{\Gamma \vdash^{\mathsf{lf}} A\ M : K[\mathrm{id}_\Gamma, M/x]}\ \mathsf{LF\_TypeApp} \qquad \frac{\Gamma \vdash^{\mathsf{lf}} A : K \quad K \equiv_{\alpha\beta\eta} K'}{\Gamma \vdash^{\mathsf{lf}} A : K'}\ \mathsf{LF\_EqualKind}$$

A.1.6 Well-Formed Kinds

$$\frac{\Gamma \ \mathsf{ctx}_{\mathsf{lf}}}{\Gamma \vdash^{\mathsf{lf}} \mathit{type} : \mathit{kind}} \qquad \frac{\Gamma \vdash^{\mathsf{lf}} A : \mathit{type} \quad \Gamma, x{:}A \vdash^{\mathsf{lf}} K : \mathit{kind}}{\Gamma \vdash^{\mathsf{lf}} \Pi x{:}A.\ K : \mathit{kind}}$$

A.2 Delphin Summary

A.2.1 Grammar and Preliminaries

Worlds	$\mathcal{W}$	::=	$\cdot \mid \mathcal{W}, (\Omega, \boldsymbol{A}) \mid *$
Types	δ	::=	$\tau \mid \boldsymbol{A} \mid \boldsymbol{A}^{\#}$
Comp. Types	τ, σ	::=	$\text{unit} \mid \forall\overline{\alpha{\in}\delta}.\ \tau \mid \exists\alpha{\in}\delta.\ \tau \mid \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau \mid \nabla\mathcal{W}.\ \tau$
Variables	α	::=	$\boldsymbol{x} \mid u$
Expressions	e, f	::=	$\alpha \mid \boldsymbol{M} \mid () \mid e\ \overline{f} \mid (e,\ f) \mid \mu u{\in}\tau.\ e \mid \text{fn}\ \overline{c}$
			$\mid \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \mid e\backslash\boldsymbol{x} \mid \nu u{\in}\mathcal{W}.\ e \mid e\backslash u$
Case	c	::=	$\epsilon\alpha{\in}\delta.\ c \mid \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \mid c\backslash\boldsymbol{x} \mid \overline{e} \mapsto f$
Context	Ω	::=	$\cdot \mid \Omega, D$
Declarations	D	::=	$\boldsymbol{x}{\in}\boldsymbol{A} \mid \boldsymbol{x}{\in}\boldsymbol{A}^{\#} \mid u{\in}\tau \mid \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \mid u\overset{\triangledown}{\in}\mathcal{W}$
Values:	v	::=	$() \mid \text{fn}\ \overline{c} \mid \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ v \mid \nu u{\in}\mathcal{W}.\ v \mid (v_1,\ v_2) \mid \boldsymbol{M}$

We use the notation $\overline{\mathcal{X}}$ to refer to a list of $\mathcal{X}$ where the empty list is universally referred to as *nil*. We use ; to deconstruct lists, i.e. $Y; \overline{\mathcal{X}}$ and $\overline{\mathcal{X}}; Y$ represent lists starting and ending with Y. We will often just write Y to refer to the singleton list. A list attached to a context, $\Omega, \overline{\alpha{\in}\delta}$, refers to the context extended with all the declarations in the list.

We use $\langle \boldsymbol{A} \rangle$ as syntactic sugar for $\exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \text{unit}$ and we similarly use $\langle \boldsymbol{A}^{\#} \rangle$ to stand for $\exists\alpha{\in}\boldsymbol{A}^{\#}.\ \text{unit}$. Similarly, we write $\langle \boldsymbol{M} \rangle$ as syntactic sugar for $(\boldsymbol{M},\ ())$. We often write $\delta \supset \sigma$ and $\delta \star \sigma$ for non-dependent uses of $\forall$ and $\exists$. Furthermore, we write Ω_1, Ω_2 to stand for the standard concatenation of contexts. And we will similarly use such syntax for the concatenation of LF contexts $\boldsymbol{\Gamma}$, and substitutions (ω and $\boldsymbol{\gamma}$).

We define $\|\Omega\|$ as an operation which takes our meta-context and converts it into a context $\boldsymbol{\Gamma}$ suitable for the logical framework (representation level).

Definition A.2.1 (Casting Context)**.**

$$\|\Omega\| = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ \|\Omega'\|, \boldsymbol{x}{:}\boldsymbol{A} & \text{if } \Omega = \Omega', \boldsymbol{x}{\in}\delta \text{ and } \delta = \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#} \\ \|\Omega'\| & \text{if } \Omega = \Omega', u{\in}\tau \\ \|\Omega'\|, \boldsymbol{x}{:}\boldsymbol{A} & \text{if } \Omega = \Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \\ \|\Omega'\| & \text{if } \Omega = \Omega', u\overset{\triangledown}{\in}\mathcal{W} \end{cases}$$

A.2.2 Well-Formedness of Types and Contexts

Well-Formed (wff) Types

$$\frac{\|\Omega\| \vdash^{\mathrm{lf}} \boldsymbol{A} : \boldsymbol{type}}{\Omega \vdash \boldsymbol{A}\ \mathsf{wff}}\ \mathsf{LF\_wff} \qquad \frac{\Omega \vdash \boldsymbol{A}\ \mathsf{wff}}{\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}}\ \mathsf{param\_wff}$$

$$\frac{}{\Omega \vdash \mathrm{unit}\ \mathsf{wff}}\ \mathrm{unit\_wff} \qquad \frac{\Omega \vdash \delta\ \mathsf{wff} \quad \Omega, \alpha{\in}\delta \vdash \tau\ \mathsf{wff}}{\Omega \vdash \exists \alpha{\in}\delta.\ \tau\ \mathsf{wff}}\ \exists\mathsf{wff}$$

$$\frac{\Omega \vdash \tau\ \mathsf{wff}}{\Omega \vdash \forall nil.\ \tau\ \mathsf{wff}}\ \forall_b\mathsf{wff} \qquad \frac{\Omega \vdash \delta_1\ \mathsf{wff} \quad \Omega, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}{\Omega \vdash \forall \alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}\ \forall_i\mathsf{wff}$$

$$\frac{\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff} \quad \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}}{\Omega \vdash \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}}\ \nabla\mathsf{wff} \qquad \frac{\mathcal{W}\ \mathsf{world} \quad \Omega \vdash \tau\ \mathsf{wff}}{\Omega \vdash \nabla \mathcal{W}.\ \tau\ \mathsf{wff}}\ \nabla_{\mathrm{world}}\mathsf{wff}$$

Valid Contexts

$$\frac{}{\cdot\ \mathsf{ctx}}\ \mathsf{ctxEmpty} \qquad \frac{\Omega\ \mathsf{ctx} \quad \Omega \vdash \delta\ \mathsf{wff}}{\Omega, \alpha{\in}\delta\ \mathsf{ctx}}\ \mathsf{ctxAdd}$$

$$\frac{\Omega\ \mathsf{ctx} \quad \Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}\ \mathsf{ctx}}\ \mathsf{ctxAddNew} \qquad \frac{\Omega\ \mathsf{ctx} \quad \mathcal{W}\ \mathsf{world}}{\Omega, u\overset{\triangledown}{\in}\mathcal{W}\ \mathsf{ctx}}\ \mathsf{ctxAddWorld}$$

A.2.3 Substitutions

$$\begin{array}{lll} \text{Substitutions:} & \omega & ::= \cdot \mid \omega, e/\alpha \mid \uparrow_\alpha \omega \\ \text{LF Substitutions:} & \boldsymbol{\gamma} & ::= \boldsymbol{\cdot} \mid \boldsymbol{\gamma, M/x} \mid \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma} \end{array}$$

Note that "$\uparrow_\alpha$" binds tighter than "," and we tacitly rename variables so that all elements in substitutions are uniquely named.

Definition A.2.2 (Casting Substitution)**.** *We next define $\|\omega\|$ which turns a computation-level substitution into an LF one by throwing out non-LF assumptions.*

$$\begin{array}{lcl} \|\cdot\| & = & \boldsymbol{\cdot} \\ \|\omega, \boldsymbol{M/x}\| & = & \|\omega\|, \boldsymbol{M/x} \\ \|\omega, e/u\| & = & \|\omega\| \end{array} \qquad \begin{array}{lcl} \|\uparrow_{\boldsymbol{x}} \omega\| & = & \uparrow_{\boldsymbol{x}} \|\omega\| \\ \|\uparrow_u \omega\| & = & \|\omega\| \end{array}$$

Typing

Static Semantics of Computation-level Substitutions:

$$\frac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase} \qquad \frac{\Omega' \vdash \omega : \Omega \quad \Omega' \vdash e \in \delta[\omega]}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$$

$$\frac{\Omega' \vdash \omega : \Omega \quad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}}{(\Omega', \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'/x}) : (\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$$

$$\frac{\Omega' \vdash \omega : \Omega \quad \mathcal{W}\ \mathsf{world}}{(\Omega', u' \overset{\triangledown}{\in} \mathcal{W}) \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u \overset{\triangledown}{\in} \mathcal{W})}\ \mathsf{tpSubWorld}$$

$$\frac{\Omega' \vdash \omega : \Omega \quad \Omega' \vdash \delta\ \mathsf{wff}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$$

Static Semantics of LF Substitutions:

$$\frac{}{\cdot \vdash^{\mathsf{lf}} \cdot : \cdot}\ \textbf{tpLF-SubBase} \qquad \frac{\boldsymbol{\Gamma' \vdash^{\mathsf{lf}} \gamma : \Gamma} \quad \boldsymbol{\Gamma' \vdash^{\mathsf{lf}} M \in A[\gamma]}}{\boldsymbol{\Gamma' \vdash^{\mathsf{lf}} (\gamma, M/x) : (\Gamma, x{:}A)}}\ \textbf{tpLF-SubInd}$$

$$\frac{\boldsymbol{\Gamma' \vdash^{\mathsf{lf}} \gamma : \Gamma} \quad \boldsymbol{\Gamma' \vdash^{\mathsf{lf}} A : \mathit{type}}}{\boldsymbol{\Gamma', x{:}A \vdash^{\mathsf{lf}} \uparrow_x \gamma : \Gamma}}\ \textbf{tpLF-SubShift}$$

Identity Substitutions

We define the identity substitutions as follows:

$$
\begin{array}{lcl}
\mathrm{id}_{\cdot} & = & \cdot \\
\mathrm{id}_{\Omega,\alpha\in\delta} & = & \uparrow_{\alpha}\mathrm{id}_{\Omega},\alpha/\alpha \\
\mathrm{id}_{\Omega,\boldsymbol{x}\in\overset{\triangledown}{\boldsymbol{A}^{\#}}} & = & \uparrow_{x}\mathrm{id}_{\Omega},\boldsymbol{x}/\boldsymbol{x} \\
\mathrm{id}_{\Omega,u\in\overset{\triangledown}{\mathcal{W}}} & = & \uparrow_{u}\mathrm{id}_{\Omega},u/u
\end{array}
\qquad
\begin{array}{lcl}
\mathbf{id}_{\cdot} & = & \cdot \\
\mathbf{id}_{\boldsymbol{\Gamma},\boldsymbol{x}:\boldsymbol{A}} & = & \uparrow_{\boldsymbol{x}}\mathbf{id}_{\boldsymbol{\Gamma}},\boldsymbol{x}/\boldsymbol{x}
\end{array}
$$

Substitution Composition

Definition A.2.3 (Substitution Composition)**.** *We define substitution composition as:*

$$
\begin{array}{lcl}
\omega\circ(\uparrow_{\alpha}\omega') & = & \uparrow_{\alpha}(\omega\circ\omega') \\
\cdot\circ\omega',\ \textit{where}\ \omega'\neq\uparrow_{\alpha''}\omega'' & = & \omega' \\
(\omega,e/\alpha)\circ\omega',\ \textit{where}\ \omega'\neq\uparrow_{\alpha''}\omega'' & = & (\omega\circ\omega',e[\omega']/\alpha) \\
(\uparrow_{\alpha}\omega)\circ(\omega',e/\alpha) & = & \omega\circ\omega'
\end{array}
$$

$$
\begin{array}{lcl}
\boldsymbol{\gamma}\circ^{\mathbf{lf}}\ (\uparrow_{\boldsymbol{x}}\boldsymbol{\gamma'}) & = & \uparrow_{\boldsymbol{x}}(\boldsymbol{\gamma}\circ^{\mathbf{lf}}\ \boldsymbol{\gamma'}) \\
\cdot\circ^{\mathbf{lf}}\ \boldsymbol{\gamma'},\ \textit{where}\ \boldsymbol{\gamma'}\neq\uparrow_{\boldsymbol{x}''}\boldsymbol{\gamma''} & = & \boldsymbol{\gamma'} \\
(\boldsymbol{\gamma},\boldsymbol{e}/\boldsymbol{x})\circ^{\mathbf{lf}}\ \boldsymbol{\gamma'},\ \textit{where}\ \boldsymbol{\gamma'}\neq\uparrow_{\boldsymbol{x}''}\boldsymbol{\gamma''} & = & (\boldsymbol{\gamma}\circ^{\mathbf{lf}}\ \boldsymbol{\gamma'},\boldsymbol{e}[\boldsymbol{\gamma'}]/\boldsymbol{x}) \\
(\uparrow_{\boldsymbol{x}}\boldsymbol{\gamma})\circ^{\mathbf{lf}}\ (\boldsymbol{\gamma'},\boldsymbol{M}/\boldsymbol{x}) & = & \boldsymbol{\gamma}\circ^{\mathbf{lf}}\ \boldsymbol{\gamma'}
\end{array}
$$

Substitution Application

Definition A.2.4 (Substitution Application)**.**

Types

$$
\begin{array}{lcl}
\boldsymbol{A}^{\#}[\omega] & = & \boldsymbol{A}[\omega]^{\#} \\
\boldsymbol{A}[\omega] & = & \boldsymbol{A}[\|\omega\|] \\
\mathrm{unit}[\omega] & = & \mathrm{unit} \\
(\forall\overline{\alpha{\in}\delta}.\ \tau)[\omega] & = & \forall\overline{\alpha{\in}\delta}[\omega].\ \tau[\omega + \overline{\alpha{\in}\delta}] \\
(\exists\alpha{\in}\delta.\ \tau)[\omega] & = & \exists\alpha{\in}\delta[\omega].\ \tau[\uparrow_{\alpha}\omega, \alpha/\alpha] \\
(\nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)[\omega] & = & \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega].\ \tau[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}] \\
(\nabla\mathcal{W}.\ \tau)[\omega] & = & \nabla\mathcal{W}.\ (\tau[\omega])
\end{array}
$$

Declaration List

$$
\begin{array}{lcl}
\mathrm{nil}[\omega] & = & \mathrm{nil} \\
(\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta})[\omega] & = & \alpha_1{\in}\delta_1[\omega]; \\
 & & \quad (\overline{\alpha{\in}\delta}[\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1])
\end{array}
$$

LF Types

$$
\begin{array}{lcl}
\boldsymbol{a}[\gamma] & = & \boldsymbol{a} \\
(\boldsymbol{\Pi x{:}A.\ B})[\gamma] & = & \boldsymbol{\Pi x{:}A}[\gamma].\ \boldsymbol{B}[\uparrow_{\boldsymbol{x}}\boldsymbol{\gamma}, \boldsymbol{x}/\boldsymbol{x}] \\
(\boldsymbol{A\ M})[\gamma] & = & (\boldsymbol{A}[\gamma])\ (\boldsymbol{M}[\gamma])
\end{array}
$$

LF Kinds

$$
\begin{array}{lcl}
\boldsymbol{type}[\gamma] & = & \boldsymbol{type} \\
(\boldsymbol{\Pi x{:}A.\ K})[\gamma] & = & \boldsymbol{\Pi x{:}A}[\gamma].\ \boldsymbol{K}[\uparrow_{\boldsymbol{x}}\boldsymbol{\gamma}, \boldsymbol{x}/\boldsymbol{x}]
\end{array}
$$

LF Objects

$$
\begin{array}{lcl}
\boldsymbol{c}[\gamma] & = & \boldsymbol{c} \\
\boldsymbol{x}[\gamma, \boldsymbol{M}/\boldsymbol{x}] & = & \boldsymbol{M} \\
\boldsymbol{x}[\gamma, \boldsymbol{M}/\boldsymbol{x'}] & & \\
\quad \textit{where } \boldsymbol{x} \neq \boldsymbol{x'} & = & \boldsymbol{x}[\gamma] \\
\boldsymbol{x}[\uparrow_{\boldsymbol{x'}}\gamma] & = & \boldsymbol{x}[\gamma] \\
(\boldsymbol{\lambda x{:}A.\ M})[\gamma] & = & \boldsymbol{\lambda x{:}A}[\gamma].\ \boldsymbol{M}[\uparrow_{\boldsymbol{x}}\boldsymbol{\gamma}, \boldsymbol{x}/\boldsymbol{x}] \\
(\boldsymbol{M\ N})[\gamma] & = & (\boldsymbol{M}[\gamma])\ (\boldsymbol{N}[\gamma])
\end{array}
$$

Expressions

$$
\begin{array}{lcl}
u[\omega, e/u] & = & e \\
u[\omega, e/\alpha] & & \\
\quad \textit{where } u \neq \alpha & = & u[\omega] \\
u[\uparrow_{\alpha}\omega] & = & u[\omega] \\
\boldsymbol{M}[\omega] & = & \boldsymbol{M}[\|\omega\|] \\
()[\omega] & = & () \\
(e\ \overline{f})[\omega] & = & (e[\omega])\ (\overline{f}[\omega]) \\
(e,\ f)[\omega] & = & (e[\omega],\ f[\omega]) \\
(\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)[\omega] & = & \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega].\ e[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}] \\
(\nu u{\in}\mathcal{W}.\ e)[\omega] & = & \nu u{\in}\mathcal{W}.\ e[\uparrow_{u}\omega, u/u] \\
(e\backslash\boldsymbol{x})[\omega] & = & e[\omega]\backslash\boldsymbol{x}[\omega] \\
(e\backslash u)[\omega] & = & e[\omega]\backslash u[\omega] \\
(\mu u{\in}\tau.\ e)[\omega] & = & \mu u{\in}\tau[\omega].\ e[\uparrow_{u}\omega, u/u] \\
(\mathrm{fn}\ \overline{c})[\omega] & = & \mathrm{fn}\ (\overline{c}[\omega])
\end{array}
$$

Case and Exp List

$$
\begin{array}{lcl}
\mathrm{nil}[\omega] & = & \mathrm{nil} \\
(c_1; \overline{c})[\omega] & = & c_1[\omega]; \overline{c}[\omega] \\
(e_1; \overline{e})[\omega] & = & e_1[\omega]; \overline{e}[\omega]
\end{array}
$$

Cases

$$
\begin{array}{lcl}
(\epsilon\alpha{\in}\delta.\ c)[\omega] & = & \epsilon\alpha{\in}\delta[\omega].\ c[\uparrow_{\alpha}\omega, \alpha/\alpha] \\
(\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c)[\omega] & = & \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega].\ c[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}] \\
(c\backslash\boldsymbol{x})[\omega] & = & c[\omega]\backslash\boldsymbol{x}[\omega] \\
(\overline{e} \mapsto f)[\omega] & = & (\overline{e}[\omega] \mapsto f[\omega])
\end{array}
$$

Operations

$$
\begin{array}{lcl}
(\mathrm{nil}/\mathrm{nil}) & = & \cdot \\
((e_1; \overline{e})/(\alpha_1; \overline{\alpha})) & = & (e_1/\alpha_1, \overline{e}/\overline{\alpha}) \\
 & & \\
\omega + \mathrm{nil} & = & \omega \\
\omega + (\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}) & = & (\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1) + \overline{\alpha{\in}\delta}
\end{array}
$$

A.2.4 World Judgments

Well-Formed Worlds

A world specification contains a list of types for which we permit the creation of new parameters. The well-formedness condition for worlds is:

$$\frac{}{\cdot\ \mathsf{world}}\ \mathsf{worldEmpty} \qquad \frac{}{*\ \mathsf{world}}\ \mathsf{worldAny}$$

$$\frac{\mathcal{W}\ \mathsf{world} \quad \Omega \vdash \boldsymbol{A}\ \mathsf{wff} \quad \Omega\ \mathsf{ctx} \quad \Omega \text{ only has decs. of } \boldsymbol{x}{\in}\delta}{\mathcal{W}, (\Omega, \boldsymbol{A})\ \mathsf{world}}\ \mathsf{worldAdd}$$

World Inclusion (Parameter Support)

The added flexibility of allowing function to range over different worlds requires us to first define which parameters are supported by a world $\mathcal{W}$. We write $(\Omega, \boldsymbol{A}) \in \mathcal{W}$ to express that world $\mathcal{W}$ supports parameter $\boldsymbol{A}$, which makes sense in Ω.

$$\frac{\Omega \vdash (\mathrm{id}_\Omega, \omega) : \Omega, \Omega'}{(\Omega, \boldsymbol{A}[\mathrm{id}_\Omega, \omega]) \in (\mathcal{W}, (\Omega', \boldsymbol{A}))}\ \mathsf{includesYes} \qquad \frac{(\Omega, \boldsymbol{A}) \in \mathcal{W}}{(\Omega, \boldsymbol{A}) \in (\mathcal{W}, (\Omega', \boldsymbol{A'}))}\ \mathsf{includesOther}$$

$$\frac{}{(\Omega, \boldsymbol{A}) \in *}\ \mathsf{includesAny}$$

World Subsumption

World subsumption utilizes World Inclusion and is defined as follows:

$$\frac{}{\cdot \leq \mathcal{W}}\ \mathsf{baseSubsume} \qquad \frac{}{\mathcal{W} \leq *}\ \mathsf{anySubsume} \qquad \frac{\mathcal{W}_1 \leq \mathcal{W}_2 \quad (\Omega, \boldsymbol{A}) \in \mathcal{W}_2}{\mathcal{W}_1, (\Omega, \boldsymbol{A}) \leq \mathcal{W}_2}\ \mathsf{containsSubsume}$$

A.2.5 Weakening

With Respect to a World

We write $\Omega \leq_{\mathcal{W}} \Omega'$ if Ω' is an extension of Ω where all parameters and weakening declarations are compatible with $\mathcal{W}$.

$$\frac{}{\Omega \leq_{\mathcal{W}} \Omega}\ \mathsf{leWorldEq} \qquad \frac{\Omega \leq_{\mathcal{W}} \Omega'}{\Omega \leq_{\mathcal{W}} \Omega', \alpha_1 \in \delta_1}\ \mathsf{leWorldAdd}$$

$$\frac{\Omega \leq_{\mathcal{W}} \Omega' \quad (\Omega', \boldsymbol{A}) \in \mathcal{W}}{\Omega \leq_{\mathcal{W}} \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\ \mathsf{leWorldAddNew} \qquad \frac{\Omega \leq_{\mathcal{W}} \Omega' \quad \mathcal{W}' \leq \mathcal{W}}{\Omega \leq_{\mathcal{W}} \Omega', u \overset{\triangledown}{\in} \mathcal{W}'}\ \mathsf{leWorldAddMarker}$$

$$\frac{\Omega \leq_{\mathcal{W}} \Omega'}{\Omega, D \leq_{\mathcal{W}} \Omega', D}\ \mathsf{leWorldMiddle}$$

Without Worlds

We write $\Omega \leq \Omega'$ to express that Ω' is an extension of Ω only by pattern-variable declarations $\alpha \in \delta$. In other words, the signature portion of the context must stay the same.

Computation-Level Contexts:

$$\frac{}{\Omega \leq \Omega}\ \mathsf{leEq} \qquad \frac{\Omega \leq \Omega'}{\Omega \leq \Omega', \alpha_1 \in \delta_1}\ \mathsf{leAdd} \qquad \frac{\Omega \leq \Omega'}{\Omega, D \leq \Omega', D}\ \mathsf{leMiddle}$$

LF-Level (Representation-Level) Contexts:

$$\frac{}{\boldsymbol{\Gamma} \leq_{\mathrm{lf}} \boldsymbol{\Gamma}}\ \textbf{leLF-Eq} \qquad \frac{\boldsymbol{\Gamma} \leq_{\mathrm{lf}} \boldsymbol{\Gamma'}}{\boldsymbol{\Gamma} \leq_{\mathrm{lf}} \boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A}}\ \textbf{leLF-Add} \qquad \frac{\boldsymbol{\Gamma} \leq_{\mathrm{lf}} \boldsymbol{\Gamma'}}{\boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A} \leq_{\mathrm{lf}} \boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A}}\ \textbf{leLF-Middle}$$

Weakening on Substitutions

We also define weakening on substitutions, which will be used in our meta-theory.

Computation-Level Substitutions:

$$\frac{}{\omega \leq \omega}\ \mathsf{leSubEq} \qquad \frac{\omega \leq \omega'}{\omega \leq (\uparrow_\alpha \omega')}\ \mathsf{leSubShift} \qquad \frac{\omega \leq \omega'}{\uparrow_\alpha \omega \leq \uparrow_\alpha \omega'}\ \mathsf{leSubMiddleShift}$$

$$\frac{\omega \leq \omega'}{\omega \leq \omega', e/\alpha}\ \mathsf{leSubAdd} \qquad \frac{\omega \leq \omega'}{(\omega, e/\alpha) \leq (\omega', e/\alpha)}\ \mathsf{leSubMiddle}$$

LF-Level (Representation-Level) Substitutions:

$$\frac{}{\boldsymbol{\gamma} \leq_{\mathrm{lf}} \boldsymbol{\gamma}}\ \mathbf{leLFsubEq} \qquad \frac{\boldsymbol{\gamma} \leq_{\mathrm{lf}} \boldsymbol{\gamma}'}{\boldsymbol{\gamma} \leq_{\mathrm{lf}} (\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}')}\ \mathbf{leLFsubShift}$$

$$\frac{\boldsymbol{\gamma} \leq_{\mathrm{lf}} \boldsymbol{\gamma}'}{\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma} \leq \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}'}\ \mathbf{leLFsubMiddleShift}$$

$$\frac{\boldsymbol{\gamma} \leq_{\mathrm{lf}} \boldsymbol{\gamma}'}{\boldsymbol{\gamma} \leq_{\mathrm{lf}} \boldsymbol{\gamma}', \boldsymbol{M}/\boldsymbol{x}}\ \mathbf{leLFsubAdd} \qquad \frac{\boldsymbol{\gamma} \leq_{\mathrm{lf}} \boldsymbol{\gamma}'}{(\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) \leq_{\mathrm{lf}} (\boldsymbol{\gamma}', \boldsymbol{M}/\boldsymbol{x})}\ \mathbf{leLFsubMiddle}$$

A.2.6 Type System

We tacitly rename variables to ensure that all elements in Ω are uniquely named. Additionally, we tacitly rename variables so that all elements in substitutions, ω, are uniquely named.

$$\frac{\Omega\ \mathsf{ctx}}{\Omega \vdash () \in \mathrm{unit}}\ \mathsf{top} \qquad \frac{(\Omega_1, u{\in}\tau, \Omega_2)\ \mathsf{ctx} \quad (\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)}{\Omega_1, u{\in}\tau, \Omega_2 \vdash u \in \tau}\ \tau\mathsf{var}$$

$$\frac{\Omega\ \mathsf{ctx} \quad ((\boldsymbol{x}{\in}\boldsymbol{A}^{\#})\ \mathrm{or}\ (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}))\ \mathrm{in}\ \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\ \mathsf{var}^{\#} \qquad \frac{\Omega\ \mathsf{ctx} \quad \|\Omega\| \overset{\mathrm{lf}}{\vdash} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$$

$$\frac{\Omega\ \mathsf{ctx} \quad \Omega \vdash \tau\ \mathsf{wff} \quad \text{for all } {}_{c_i \in \overline{c}}(\Omega \vdash c_i \in \tau) \quad \Omega \vdash \overline{c}\ \mathrm{covers}\ \tau}{\Omega \vdash \mathrm{fn}\ \overline{c} \in \tau}\ \mathsf{impI}$$

$$\frac{\Omega \vdash e \in \forall\overline{\alpha{\in}\delta}.\ \tau \quad \Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta}}{\Omega \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$$

$$\frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{new} \qquad \frac{\Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash e \in \tau}{\Omega \vdash \nu u{\in}\mathcal{W}.\ e \in \nabla\mathcal{W}.\ \tau}\ \mathsf{newW}$$

$$\frac{(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \quad \Omega_1 \vdash e \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \quad (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$$

$$\frac{(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)\ \mathsf{ctx} \quad \Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau \quad \mathcal{W} \leq \mathcal{W}_2 \quad (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)}{\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \in \tau}\ \mathsf{popW}$$

$$\frac{\Omega \vdash (\exists\alpha{\in}\delta.\ \tau)\ \mathsf{wff} \quad \Omega \vdash e \in \delta \quad \Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]}{\Omega \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\ \mathsf{pairI} \qquad \frac{\Omega, u{\in}\tau \vdash e \in \tau}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\ \mathsf{fix}$$

. . . . . .

$$\frac{\Omega \vdash \tau\ \mathsf{wff} \quad \Omega, \alpha{\in}\delta \vdash c \in \tau}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\ \mathsf{cEps} \qquad \frac{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{cNew}$$

$$\frac{\Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}] \quad \Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta} \quad \Omega \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}{\Omega \vdash \overline{e} \mapsto f \in \forall\overline{\alpha{\in}\delta}.\ \tau}\ \mathsf{cMatch}$$

$$\frac{(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \quad \Omega_1 \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \quad (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$$

A.2.7 Operational Semantics

We define *equality* over values $v_1 = v_2$, to be syntactic equality with implicit α, β, and η-conversion on LF terms $\boldsymbol{M}$, as justified by Harper and Pfenning (2005). Additionally, we use the syntax $\xrightarrow{*}$ to refer to the transitive closure (0 or more steps).

$$\frac{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash e \to f}{\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e \to \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ f}\ \mathsf{redNew} \qquad \frac{\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash e \to f}{\Omega \vdash \nu u \in \mathcal{W}.\ e \to \nu u \in \mathcal{W}.\ f}\ \mathsf{redNewW}$$

$$\frac{\Omega \vdash e \to e'}{\Omega \vdash (e,\ f) \to (e',\ f)}\ \mathsf{redPairL} \qquad \frac{\Omega \vdash f \to f'}{\Omega \vdash (e,\ f) \to (e,\ f')}\ \mathsf{redPairR}$$

$$\frac{\Omega \vdash e \to e'}{\Omega \vdash e\ \overline{f} \to e'\ \overline{f}}\ \mathsf{redAppL} \qquad \frac{\Omega \vdash f_i \to f_i'}{\Omega \vdash e\ (\overline{f_1}; f_i; \overline{f_n}) \to e\ (\overline{f_1}; f_i'; \overline{f_n})}\ \mathsf{redAppR}$$

$$\frac{\Omega_1 \vdash e \to f}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash e \backslash \boldsymbol{x} \to f \backslash \boldsymbol{x}}\ \mathsf{redPop} \qquad \frac{\Omega_1 \vdash e \to f}{\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash e \backslash u \to f \backslash u}\ \mathsf{redPopW}$$

$$\frac{}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ e) \backslash \boldsymbol{x} \to e[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{redPopElim}$$

$$\frac{}{\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash (\nu u' \in \mathcal{W}_2.\ e) \backslash u \to e[\uparrow_u \mathrm{id}_{\Omega_1}, u/u']}\ \mathsf{redPopElimW}$$

$$\frac{}{\Omega \vdash (\mathrm{fn}\ \overline{c}) \backslash \boldsymbol{x} \to \mathrm{fn}\ (\overline{c} \backslash \boldsymbol{x})}\ \mathsf{redPopFn} \qquad \frac{\Omega \vdash c_i \xrightarrow{*} (\overline{v} \mapsto e)}{\Omega \vdash (\mathrm{fn}\ (\overline{c_1}; c_i; \overline{c_n}))\ \overline{v} \to e}\ \mathsf{redexLam}$$

$$\frac{}{\Omega \vdash \mu u \in \tau.\ e \to e[\mathrm{id}_\Omega, \mu u \in \tau.\ e/u]}\ \mathsf{redFix}$$

. . . . . .

$$\frac{\Omega \vdash v \in \delta}{\Omega \vdash \epsilon \alpha \in \delta.\ c \to c[\mathrm{id}_\Omega, v/\alpha]}\ \mathsf{redEps} \qquad \frac{\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash c \to c'}{\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ c \to \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ c'}\ \mathsf{redCaseNew}$$

$$\frac{\Omega \vdash e_i \to e_i'}{\Omega \vdash ((\overline{e_1}; e_i; \overline{e_n}) \mapsto f) \to ((\overline{e_1}; e_i'; \overline{e_n}) \mapsto f)}\ \mathsf{redCasePattern}$$

$$\frac{\Omega_1 \vdash c \to c'}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash c \backslash \boldsymbol{x} \to c' \backslash \boldsymbol{x}}\ \mathsf{redCasePop}$$

$$\frac{}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ c) \backslash \boldsymbol{x} \to c[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{redCasePopElim}$$

Transitive Closure for Cases

$$\frac{}{\Omega \vdash c \xrightarrow{*} c}\ \mathsf{closeBase} \qquad \frac{\Omega \vdash c_1 \rightarrow c_2 \quad \Omega \vdash c_2 \xrightarrow{*} c_3}{\Omega \vdash c_1 \xrightarrow{*} c_3}\ \mathsf{closeInd}$$

A.2.8 Coverage

Judgment Summary

The main coverage judgment is $\Omega \vdash \overline{c}$ covers τ which checks if a list of cases is covered in the context Ω. The other judgments traverse either a signature, world, or context.

Signature Coverage	$\boldsymbol{\Sigma} \gg \overline{c}$ covers τ
Local Parameter Coverage	$\boldsymbol{\Gamma} \gg \overline{c}$ covers τ
Schematic Parameter Coverage	$\mathcal{W} \gg \overline{c}$ covers τ
Global Parameter Coverage	$\Omega \gg^{\text{world}} \overline{c}$ covers τ
Parameter Function Coverage	$\boldsymbol{\Gamma} \gg^{\nabla} \overline{c}$ covers τ
Complete Coverage	$\Omega \vdash \overline{c}$ covers τ

Preliminaries

Definition A.2.5 (Abbreviations / Operations)**.**
In Substitution Application, we defined $\omega + \overline{\alpha \in \delta}$ and we similarly define $\omega + \Gamma$ to extend a substitution ω with identities on everything in Γ.

$$\begin{array}{rcl} \omega + \cdot & = & \omega \\ \omega + (\boldsymbol{x_1}{:}\boldsymbol{A_1}, \boldsymbol{\Gamma}) & = & (\uparrow_{\boldsymbol{x_1}} \omega, \boldsymbol{x_1}/\boldsymbol{x_1}) + \boldsymbol{\Gamma} \end{array}$$

For any (possibly empty) context $\boldsymbol{\Gamma} = \cdot, \boldsymbol{x_1}{:}\boldsymbol{A_1}, \ldots, \boldsymbol{x_m}{:}\boldsymbol{A_m}$, we define the following:

$$\begin{array}{lcl} \boldsymbol{\Pi\Gamma}.\ \boldsymbol{B} & \equiv & \boldsymbol{\Pi x_1{:}A_1}.\ \ldots \boldsymbol{\Pi x_m{:}A_m}.\ \boldsymbol{B}, \text{ where } \boldsymbol{B} \neq \boldsymbol{\Pi x{:}B_2}.\ \boldsymbol{B_3} \\ \boldsymbol{M}\ \boldsymbol{\Gamma} & \equiv & \boldsymbol{M}\ \boldsymbol{x_1}\ \ldots\ \boldsymbol{x_m} \\ \nabla\boldsymbol{\Gamma}.\ \tau & \equiv & \nabla \boldsymbol{x_1} \in \boldsymbol{A_1}^{\#}. \ldots \nabla \boldsymbol{x_m} \in \boldsymbol{A_m}^{\#}.\ \tau, \text{ where } \tau \neq \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \sigma_2 \\ e\ \backslash\boldsymbol{\Gamma} & \equiv & e\ \backslash \boldsymbol{x_1} \backslash \ldots \backslash \boldsymbol{x_m} \\ \nu\boldsymbol{\Gamma}.\ c & \equiv & \nu \boldsymbol{x_1} \in \boldsymbol{A_1}^{\#}. \ldots \nu \boldsymbol{x_m} \in \boldsymbol{A_m}^{\#}.\ c \\ \Omega, \boldsymbol{\Gamma} & \equiv & \Omega, \boldsymbol{x_1} \in \boldsymbol{A_1}, \ldots, \boldsymbol{x_m} \in \boldsymbol{A_m} \\ \boldsymbol{\Gamma}[\omega] & \equiv & \boldsymbol{x_1}{:}\boldsymbol{A_1}[\|\omega\|], \ldots, \boldsymbol{x_m}{:}\boldsymbol{A_m}[\|\omega + (\cdot, \boldsymbol{x_1}{:}\boldsymbol{A_1}, \ldots, \boldsymbol{x_{m-1}}{:}\boldsymbol{A_{m-1}})\|] \\ \mathrm{id}_{\boldsymbol{\Gamma}} & \equiv & \cdot, \boldsymbol{x_1}/\boldsymbol{x_1}, \ldots, \boldsymbol{x_m}/\boldsymbol{x_m} \end{array}$$

For any (possibly empty) context $\Omega = \cdot, \alpha_1 \in \delta_1, \ldots, \alpha_m \in \delta_m$, we define the following:

$$\begin{array}{lcl} \forall\Omega.\ \tau & \equiv & \forall(\alpha_1 \in \delta_1; \ldots; \alpha_m \in \delta_m).\ \tau \\ \epsilon\Omega.\ c & \equiv & \epsilon\alpha_1 \in \delta_1. \ldots \epsilon\alpha_m \in \delta_m.\ c \\ \Omega[\omega]; f & \equiv & \alpha_1[\omega]; \ldots; \alpha_m[\omega]; f \\ \Omega; f & \equiv & \alpha_1; \ldots; \alpha_m; f \\ \uparrow_{\Omega} \omega & \equiv & \uparrow_{\alpha_m} \ldots \uparrow_{\alpha_1} \omega \end{array}$$

For any nonempty *list of cases $\overline{c} = c_1; \ldots; c_n;$* nil*, we define the following:*

$$\overline{c}\backslash\boldsymbol{x} \quad \equiv \quad (c_1\backslash\boldsymbol{x}); \ldots; (c_n\backslash\boldsymbol{x}); \mathrm{nil}$$

Unification

Our coverage algorithm is designed to work with respect to any unification algorithm and hence our rules abstract away from the details of any such algorithm.

Definition A.2.6 (Unifiers)**.** *We write* $\omega \in (\boldsymbol{A} \approx \boldsymbol{B})$ *to express that* ω *is a substitution such that* $\boldsymbol{A}[\omega] \equiv_{\boldsymbol{\alpha\beta\eta}} \boldsymbol{B}[\omega]$*. We call* ω *a unifier of* $\boldsymbol{A}$ *and* $\boldsymbol{B}$*.*

Definition A.2.7 (Most-General Unifiers)**.** *We write* $\omega = mgu\ (\boldsymbol{A} \approx \boldsymbol{B})$ *if* ω *is a most-general unifier which means that for all other unifiers* ω'*, there must exist an* ω_2 *such that* $\omega' = \omega \circ \omega_2$*.*

Rules for Signature Coverage

$$\frac{}{\cdot \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{scEmpty}$$

$$\frac{\begin{array}{l}\boldsymbol{\Sigma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \text{mgu } (\boldsymbol{B_x} \approx \boldsymbol{B_c}) : \Omega_A, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_c} \\ \omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_c \\ g = \epsilon\Omega'.\ \Omega_A[\omega]; (\boldsymbol{\lambda(\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A])\boldsymbol{.\ (c\ \Gamma_c)}[\omega]) \mapsto f \\ g \text{ in } \overline{c}\end{array}}{\boldsymbol{\Sigma, c{:}(\Pi\Gamma_c.\ B_c)} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{scUnify}$$

$$\frac{\begin{array}{l}\boldsymbol{\Sigma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_c}) \text{ do not unify}\end{array}}{\boldsymbol{\Sigma, c{:}(\Pi\Gamma_c.\ B_c)} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{scSkip}$$

Rules for Local Parameter Coverage

$$\frac{}{\cdot \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{lcEmpty}$$

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \text{mgu } (\boldsymbol{B_x} \approx \boldsymbol{B_p}) : \Omega_A, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_p} \\ \omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_p \\ g = \epsilon\Omega'.\ \Omega_A[\omega]; (\boldsymbol{\lambda(\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A])\boldsymbol{.\ (p\ \Gamma_p)}[\omega]) \mapsto f \\ g \text{ in } \overline{c}\end{array}}{\boldsymbol{\Gamma, p{:}(\Pi\Gamma_p.\ B_p)} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{lcUnify}$$

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_p}) \text{ do not unify}\end{array}}{\boldsymbol{\Gamma, p{:}(\Pi\Gamma_p.\ B_p)} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{lcSkip}$$

Rules for Schematic Parameter Coverage

$$\frac{}{\cdot \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{wcEmpty}$$

$$\frac{\begin{array}{l}\mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \text{mgu } (\boldsymbol{B_x} \approx \boldsymbol{B_b}) : \Omega_b, \Omega_A, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_b} \\ \omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_b \\ g = \epsilon\Omega'.\ \epsilon x_b{\in}(\boldsymbol{\Pi\Gamma_b.\ B_b})[\omega]^{\#}.\ \Omega_A[\omega]; (\boldsymbol{\lambda(\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A])\boldsymbol{.\ (x_b\ \Gamma_b)}[\omega]) \mapsto f \\ g \text{ in } \overline{c}\end{array}}{\mathcal{W}, (\Omega_b,\ (\boldsymbol{\Pi\Gamma_b.\ B_b})) \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{wcUnify}$$

$$\frac{\begin{array}{l}\mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_b}) \text{ do not unify}\end{array}}{\mathcal{W}, (\Omega_b,\ (\boldsymbol{\Pi\Gamma_b.\ B_b})) \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{wcSkip}$$

Rules for Global Parameter Coverage

$$\frac{}{\cdot \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcEmpty}$$

$$\frac{\begin{array}{l}\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_c}) \text{ do not unify}\end{array}}{\Omega_g, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcSkipMismatch}$$

$$\frac{\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}{\Omega_g, \alpha{\in}\delta \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcSkipNonParameter}$$

$$\frac{\begin{array}{l}\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \mathcal{W} \text{ world} \\ \mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau \\ (\Omega_g, (\boldsymbol{\Pi\Gamma_c.\ B_c})) \in \mathcal{W}\end{array}}{\Omega_g, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcSkipWorld}$$

$$\frac{\begin{array}{l}\Omega_g \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ \mathcal{W}_2 \text{ world} \\ \mathcal{W}_2 \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau\end{array}}{\Omega_g, u\overset{\triangledown}{\in}\mathcal{W} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcCheckWorld}$$

Parameter Function Coverage

This judgment is used for determining coverage of parameter functions.

$$\frac{}{\cdot \gg^{\nabla} \overline{c} \text{ covers } \nabla \boldsymbol{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau}\ \textsf{pfEmpty}$$

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \gg^{\nabla} \overline{c} \text{ covers } \nabla \boldsymbol{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau \\ \Omega', \boldsymbol{\Gamma_g} \vdash \omega = \text{mgu } (\boldsymbol{A_p} \approx \boldsymbol{A}) : \boldsymbol{\Gamma_g}, \Omega_A \\ \omega = ((\uparrow_{\Omega'} \cdot) + \boldsymbol{\Gamma_g}), \omega_A \\ g = \epsilon\Omega'.\ \nu\boldsymbol{\Gamma_g}.\ (\Omega_A[\omega]; \boldsymbol{p} \mapsto f) \\ g \text{ in } \overline{c}\end{array}}{\boldsymbol{\Gamma}, \boldsymbol{p}{:}\boldsymbol{A_p} \gg^{\nabla} \overline{c} \text{ covers } \nabla \boldsymbol{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau}\ \textsf{pfUnify}$$

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \gg^{\nabla} \overline{c} \text{ covers } \nabla \boldsymbol{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau \\ (\boldsymbol{A_p} \approx \boldsymbol{A}) \text{ do not unify}\end{array}}{\boldsymbol{\Gamma}, \boldsymbol{p}{:}\boldsymbol{A_p} \gg^{\nabla} \overline{c} \text{ covers } \nabla \boldsymbol{\Gamma_g}.\ \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau}\ \textsf{pfSkip}$$

Complete Coverage

$$\frac{c = \epsilon\alpha{\in}\delta.\ \alpha \mapsto f}{\Omega \vdash c \text{ covers } \forall\alpha{\in}\delta.\ \tau}\ \mathsf{coverSimple}$$

$$\frac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau\ \mathsf{wff} \\ c = \epsilon\boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y}\ \boldsymbol{\Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y}\ \boldsymbol{\Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}$$

$$\frac{\begin{array}{l}\cdot \vdash \exists\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff} \\ c = \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\exists\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}^{\#}$$

$$\frac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)\ \mathsf{wff} \\ c = \epsilon u_1{\in}(\nabla\boldsymbol{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla\boldsymbol{\Gamma}.\ \tau_2).\ (\nu\boldsymbol{\Gamma}.\ (u_1\backslash\boldsymbol{\Gamma},\ u_2\backslash\boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\nabla\boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma}\ \mathsf{coverPairMeta}$$

$$\frac{\Omega_1 \vdash \bar{c} \text{ covers } \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \quad (\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_2)}{\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash \bar{c}\backslash\boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$$

$$\frac{\begin{array}{l}(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}\boldsymbol{.}\ \boldsymbol{B}_{\boldsymbol{x}})\ \mathsf{ctx} \\ \Omega_A \text{ only contains declarations of type } \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#} \\ \boldsymbol{\Sigma} \gg \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}\boldsymbol{.}\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau \\ \boldsymbol{\Gamma}_{\boldsymbol{x}} \gg \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}\boldsymbol{.}\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau \\ \Omega \gg^{\text{world}} \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}\boldsymbol{.}\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau\end{array}}{\Omega \vdash \bar{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}\boldsymbol{.}\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau}\ \mathsf{coverLF}$$

(Rules Continued →)

$$\frac{\begin{array}{l}(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx} \\ \text{All elements of } \Omega_A \text{ occur free in } \boldsymbol{A} \\ g = \epsilon\Omega_A.\ \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \nu\boldsymbol{\Gamma_g}.\ (\Omega_A; \boldsymbol{x} \mapsto f) \\ g \text{ in } \overline{\mathsf{c}} \\ \boldsymbol{\Gamma} \gg^{\nabla} \overline{c} \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)\end{array}}{\Omega \vdash \overline{c} \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverNewLF}^{\#}$$

$$\frac{\cdot \leq. \Omega}{\Omega \vdash \mathit{nil} \text{ covers } (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverEmpty}^{\#}$$

$$\frac{\begin{array}{l}\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type} \\ c = \epsilon\boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \nu\boldsymbol{\Gamma}.\ ((\boldsymbol{x}\ \boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau)}\ \mathsf{coverNewLF}$$

$$\frac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \sigma\ \mathsf{wff} \\ c = \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \sigma).\ \nu\boldsymbol{\Gamma}.\ ((u\backslash\boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\sigma \supset \tau)}\ \mathsf{coverNewMeta}$$

Appendix B

Detailed Proofs

In this chapter we prove that Delphin is a type safe language. The complications arise due to dependencies and especially with respect to the equivalence and manipulation of substitutions (including most-general unifiers).

Our notion of equality (=) is syntactic equality modulo $\alpha\beta\eta$ for LF components. Additionally, notice that substitution application (Def. A.2.4) may get stuck when applied to badly-typed substitutions. For example, $u[\cdot]$ is undefined, while $u[\cdot, e/u] = e$. To be precise, our lemmas will assert the *existence* of a substitution application. For example, we say $\delta[\omega]$ exists if there exists a δ' such that $\delta' = \delta[\omega]$.

B.1 Meta-Theory: Representation Level (LF)

The lemmas in this section focus solely on the representation level, LF. As this is standard, these lemmas are intensionally terse and we direct the reader to Harper and Pfenning (2005) for more details.

Lemma B.1.1 (LF-Level Judgments in Good Context)**.**
If $\mathbf{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{M} : \boldsymbol{A}$ *or* $\mathbf{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{A} : \boldsymbol{K}$ *or* $\mathbf{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{K} : \boldsymbol{kind}$*, then* $\mathbf{\Gamma}\ \mathsf{ctx}_{\mathsf{lf}}$.

Proof. By induction on $\mathcal{E} :: \mathbf{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{M} : \boldsymbol{A}$ and $\mathcal{E} :: \mathbf{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{A} : \boldsymbol{K}$ and $\mathcal{E} :: \mathbf{\Gamma} \vdash^{\mathsf{f}} \boldsymbol{K} : \boldsymbol{kind}$. The condition is proved by the premises in the base cases. Otherwise it is straightforward by induction on the premise (utilizing inversion when induction is on an extended context). □

Lemma B.1.2 (LF-Level Substitution Implies Good Context)**.**
If $\mathbf{\Gamma'} \vdash^{\mathsf{f}} \boldsymbol{\gamma} : \mathbf{\Gamma}$*, then* $\mathbf{\Gamma'}\ \mathsf{ctx}_{\mathsf{lf}}$.

Proof. Straightforward by induction on $\mathcal{E} :: \mathbf{\Gamma'} \vdash^{\mathsf{f}} \boldsymbol{\gamma} : \mathbf{\Gamma}$. □

Lemma B.1.3 (LF-Level id is Well-Typed)**.**
If $\Gamma\ \mathsf{ctx}_{\mathsf{lf}}$*, then* $\Gamma \vdash^{\mathsf{lf}} \mathbf{id}_\Gamma : \Gamma$.

Proof. Straightforward by induction on Γ. □

Lemma B.1.4 (LF-Level id Redundant in Substitution Application)**.**

1. *If* $\Gamma \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{K}$*, then* $\boldsymbol{A}[\mathbf{id}_\Gamma] = \boldsymbol{A}$.
2. *If* $\Gamma \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$*, then* $\boldsymbol{M}[\mathbf{id}_\Gamma] = \boldsymbol{M}$.
3. *If* $\Gamma \vdash^{\mathsf{lf}} \boldsymbol{K} : \boldsymbol{kind}$*, then* $\boldsymbol{K}[\mathbf{id}_\Gamma] = \boldsymbol{K}$.

Proof. By induction on the typing and kinding judgment utilizing the definition of Substitution Application. (Definition A.2.4). □

Lemma B.1.5 (LF-Level Shift Redundant in Substitution Application)**.**
$\boldsymbol{A}[(\uparrow_{\boldsymbol{x}} \gamma), \gamma'] = \boldsymbol{A}[\gamma, \gamma']$ *and* $\boldsymbol{M}[(\uparrow_{\boldsymbol{x}} \gamma), \gamma'] = \boldsymbol{M}[\gamma, \gamma']$ *and* $\boldsymbol{K}[(\uparrow_{\boldsymbol{x}} \gamma), \gamma'] = \boldsymbol{K}[\gamma, \gamma']$.

Proof. By inspection of Substitution Application (Def. A.2.4). □

Lemma B.1.6 (LF-Level Substitution Weakening on Types)**.**
If $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{A}[\gamma, \gamma_2] : \boldsymbol{K}$ *and* $\gamma \leq_{\mathsf{lf}} \gamma'$*, then* $\boldsymbol{A}[\gamma', \gamma_2] = \boldsymbol{A}[\gamma, \gamma_2]$.

Proof. By induction on $\mathcal{E} :: \gamma \leq_{\mathsf{lf}} \gamma'$.

Case: $\mathcal{E} = \dfrac{}{\gamma \leq_{\mathsf{lf}} \gamma}\ \mathsf{leLFsubEq}$

$\boldsymbol{A}[\gamma, \gamma_2] = \boldsymbol{A}[\gamma, \gamma_2]$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \gamma \leq_{\mathsf{lf}} \gamma'\end{array}}{\gamma \leq_{\mathsf{lf}} (\uparrow_{\boldsymbol{x}} \gamma')}\ \mathsf{leLFsubShift}$

$\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{A}[\gamma, \gamma_2] : \boldsymbol{K}$	by assumption
$\boldsymbol{A}[\gamma', \gamma_2] = \boldsymbol{A}[\gamma, \gamma_2]$	by i.h.[1] on $\mathcal{E}_1$
$\boldsymbol{A}[(\uparrow_{\boldsymbol{x}} \gamma'), \gamma_2]$	
$= \boldsymbol{A}[\gamma', \gamma_2]$	by Lemma B.1.5
$= \boldsymbol{A}[\gamma, \gamma_2]$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \gamma \leq_{\mathsf{lf}} \gamma'\end{array}}{\uparrow_{\boldsymbol{x}} \gamma \leq_{\mathsf{lf}} \uparrow_{\boldsymbol{x}} \gamma'}\ \mathsf{leLFsubMiddleShift}$

$\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{A}[(\uparrow_{\boldsymbol{x}} \gamma), \gamma_2] : \boldsymbol{K}$	by assumption
$\boldsymbol{A}[(\uparrow_{\boldsymbol{x}} \gamma), \gamma_2] = \boldsymbol{A}[\gamma, \gamma_2]$	by Lemma B.1.5
$\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{A}[\gamma, \gamma_2] : \boldsymbol{K}$	by above
$\boldsymbol{A}[\gamma', \gamma_2] = \boldsymbol{A}[\gamma, \gamma_2]$	by i.h. on $\mathcal{E}_1$
$\boldsymbol{A}[(\uparrow_{\boldsymbol{x}} \gamma'), \gamma_2]$	
$= \boldsymbol{A}[\gamma', \gamma_2]$	by Lemma B.1.5
$= \boldsymbol{A}[(\uparrow_{\boldsymbol{x}} \gamma), \gamma_2]$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \gamma \leq_{\mathsf{lf}} \gamma'\end{array}}{\gamma \leq_{\mathsf{lf}} \gamma', \boldsymbol{M/x}}\ \mathsf{leLFsubAdd}$

$\boldsymbol{\Gamma' \vdash^{\mathsf{lf}} A[\gamma, \gamma_2] : K}$ by assumption
$\boldsymbol{A[\gamma', \gamma_2] = A[\gamma, \gamma_2]}$ by i.h. on $\mathcal{E}_1$
$\boldsymbol{\Gamma' \vdash^{\mathsf{lf}} A[\gamma', \gamma_2] : K}$ by above
$\boldsymbol{A[\gamma', M/x, \gamma_2] = A[\gamma', \gamma_2]}$
This is a result of us tacitly renaming variable names.
As $\boldsymbol{A[\gamma', \gamma_2]}$ is defined, the only way adding $\boldsymbol{M/x}$ could affect substitution application is if there was already an $\boldsymbol{x}$ in $\boldsymbol{(\gamma', \gamma_2)}$, which we disallow.

$\boldsymbol{A[\gamma', M/x, \gamma_2] = A[\gamma, \gamma_2]}$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \gamma \leq_{\mathsf{lf}} \gamma'\end{array}}{(\gamma, \boldsymbol{M/x}) \leq_{\mathsf{lf}} (\gamma', \boldsymbol{M/x})}\ \mathsf{leLFsubMiddle}$

$\boldsymbol{\Gamma' \vdash^{\mathsf{lf}} A[\gamma, M/x, \gamma_2] : K}$ by assumption
$\boldsymbol{A[\gamma', M/x, \gamma_2] = A[\gamma, M/x, \gamma_2]}$ by i.h. on $\mathcal{E}_1$

□

Lemma B.1.7 (LF-Level Substitution Weakening on Objects)**.**
If $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{M}[\boldsymbol{\gamma}, \boldsymbol{\gamma_2}] : \boldsymbol{A}$ *and* $\boldsymbol{\gamma} \leq_{\mathsf{lf}} \boldsymbol{\gamma'}$*, then* $\boldsymbol{M}[\boldsymbol{\gamma'}, \boldsymbol{\gamma_2}] = \boldsymbol{M}[\boldsymbol{\gamma}, \boldsymbol{\gamma_2}]$.

Proof. By induction on $\mathcal{E} :: \boldsymbol{\gamma} \leq_{\mathsf{lf}} \boldsymbol{\gamma'}$.
This proof proceeds exactly as in Lemma B.1.6. □

Lemma B.1.8 (LF-Level Substitution Weakening on Kinds)**.**
If $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{K}[\boldsymbol{\gamma}, \boldsymbol{\gamma_2}] : \boldsymbol{kind}$ *and* $\boldsymbol{\gamma} \leq_{\mathsf{lf}} \boldsymbol{\gamma'}$*, then* $\boldsymbol{K}[\boldsymbol{\gamma'}, \boldsymbol{\gamma_2}] = \boldsymbol{K}[\boldsymbol{\gamma}, \boldsymbol{\gamma_2}]$.

Proof. By induction on $\mathcal{E} :: \boldsymbol{\gamma} \leq_{\mathsf{lf}} \boldsymbol{\gamma'}$.
This proof proceeds exactly as in Lemma B.1.6. □

Lemma B.1.9 (LF-Level Context Weakening on Typing)**.**
If $\boldsymbol{\Gamma} \leq_{\mathsf{lf}} \boldsymbol{\Gamma'}$ *and* $\boldsymbol{\Gamma'}\ \mathsf{ctx}_{\mathsf{lf}}$ *and* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$*, then* $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$.

Proof. Our LF level is the standard Edinburgh Logical Framework LF, and the reader is directed to Harper and Pfenning (2005) for the proof. □

Lemma B.1.10 (LF-Level Context Weakening on Kinding)**.**
If $\boldsymbol{\Gamma} \leq_{\mathsf{lf}} \boldsymbol{\Gamma'}$ *and* $\Gamma'\ \mathsf{ctx}_{\mathsf{lf}}$ *and* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{K}$*, then* $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{K}$.

Proof. Our LF level is the standard Edinburgh Logical Framework LF, and the reader is directed to Harper and Pfenning (2005) for the proof. □

Lemma B.1.11 (LF-Level Context Weakening on Well-Formed Kinds)**.**
If $\boldsymbol{\Gamma} \leq_{\mathsf{lf}} \boldsymbol{\Gamma'}$ *and* $\boldsymbol{\Gamma'}\ \mathsf{ctx}_{\mathsf{lf}}$ *and* $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{K} : \boldsymbol{kind}$*, then* $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{K} : \boldsymbol{kind}$.

Proof. Our LF level is the standard Edinburgh Logical Framework LF, and the reader is directed to Harper and Pfenning (2005) for the proof. □

Lemma B.1.12 (LF-Level Substitution Property)**.**
If $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}$*, then*

1. *if* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$*, then* $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{M}[\boldsymbol{\gamma}] : \boldsymbol{A}[\boldsymbol{\gamma}]$.
2. *if* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{K}$*, then* $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{A}[\boldsymbol{\gamma}] : \boldsymbol{K}[\boldsymbol{\gamma}]$.

Proof. By induction on $\mathcal{E} :: \boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$ and $\mathcal{E} :: \boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{K}$. Our LF level is the standard Edinburgh Logical Framework LF, and the reader is directed to Harper and Pfenning (2005) for the detailed proof. It is important to note that the substitutions in Harper and Pfenning (2005) lack $\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}$. Therefore, their typing rules have weakening built-in as:

$$\frac{}{\boldsymbol{\Gamma'} \vdash \cdot : \cdot} \qquad \frac{\boldsymbol{\Gamma'} \vdash \boldsymbol{\gamma} : \boldsymbol{\Gamma} \quad \boldsymbol{\Gamma'} \vdash \boldsymbol{M} : \boldsymbol{A}[\boldsymbol{\gamma}]}{\boldsymbol{\Gamma'} \vdash (\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) : \boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A}}$$

However, it is easy to prove that if our substitution is well-typed, the substitution by throwing out all $\uparrow_{\boldsymbol{x}}$ is well-typed in their system (by straightforward induction on our substitution typing rules). Additionally, the result of substitution application is the same as it ignores $\uparrow_{\boldsymbol{x}}$. We prefer our notion of substitution as it allows for composition to be well-typed (Lemma B.1.17), which does not hold in their formulation (However, it is also not necessary, at least for LF). □

Lemma B.1.13 (LF-Level Validity)**.**

- *If* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$*, then* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$.
- *If* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{K}$*, then* $\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{K} : \boldsymbol{kind}$.

Proof. Our LF level is the standard Edinburgh Logical Framework LF, and the reader is directed to Harper and Pfenning (2005) for the proof. □

Lemma B.1.14 (LF-Level Composition Exists)**.**
If $\boldsymbol{\Gamma' \vdash^{\mathbf{lf}} \gamma : \Gamma}$ *and* $\boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} \gamma_2 : \Gamma'}$*, then* $\boldsymbol{(\gamma \circ^{\mathbf{lf}} \gamma_2)}$ *exists.*

Proof. By induction lexicographically on $\mathcal{E} :: \boldsymbol{\Gamma' \vdash^{\mathbf{lf}} \gamma : \Gamma}$ and $\mathcal{F} :: \boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} \gamma_2 : \Gamma'}$.

Case: $\mathcal{E} = \boldsymbol{\Gamma' \vdash^{\mathbf{lf}} \gamma : \Gamma}$ and $\mathcal{F} = \dfrac{\overset{\mathcal{F}_1}{\boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} \gamma_2 : \Gamma'}} \quad \boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} A : \mathit{type}}}{\boldsymbol{\Gamma_2, x{:}A \vdash^{\mathbf{lf}} \uparrow_x \gamma_2 : \Gamma'}}$

$\boldsymbol{(\gamma \circ^{\mathbf{lf}} \gamma_2)}$ exists	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\boldsymbol{(\uparrow_x (\gamma \circ^{\mathbf{lf}} \gamma_2))}$ exists	by above
$\boldsymbol{(\gamma \circ^{\mathbf{lf}} (\uparrow_x \gamma_2))}$ exists	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{}{\boldsymbol{\cdot \vdash^{\mathbf{lf}} \cdot : \cdot}}$ and $\mathcal{F} = \boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} \gamma_2 : \cdot}$, where $\boldsymbol{\gamma_2 \neq \uparrow_{x''} \gamma_2''}$

$\boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} \gamma_2 : \cdot}$	by assumption ($\mathcal{F}$)
$\boldsymbol{\gamma_2}$ exists	by above
$\boldsymbol{(\cdot \circ^{\mathbf{lf}} \gamma_2)}$ exists	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\boldsymbol{\Gamma' \vdash^{\mathbf{lf}} \gamma : \Gamma}} \quad \boldsymbol{\Gamma' \vdash^{\mathbf{lf}} M : A[\gamma]}}{\boldsymbol{\Gamma' \vdash^{\mathbf{lf}} (\gamma, M/x) : (\Gamma, x{:}A)}}$

and $\mathcal{F} = \boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} \gamma_2 : \Gamma'}$, where $\boldsymbol{\gamma_2 \neq \uparrow_{x''} \gamma_2''}$

$\boldsymbol{(\gamma \circ^{\mathbf{lf}} \gamma_2)}$ exists	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$
$\boldsymbol{\Gamma' \vdash^{\mathbf{lf}} M : A[\gamma]}$	by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} M[\gamma_2] : A[\gamma][\gamma_2]}$	by Lemma B.1.12.1
$\boldsymbol{((\gamma \circ^{\mathbf{lf}} \gamma_2), M[\gamma_2]/x)}$ exists	by above
$\boldsymbol{((\gamma, M/x) \circ^{\mathbf{lf}} \gamma_2)}$ exists	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\boldsymbol{\Gamma' \vdash^{\mathbf{lf}} \gamma : \Gamma}} \quad \boldsymbol{\Gamma' \vdash^{\mathbf{lf}} A : \mathit{type}}}{\boldsymbol{\Gamma', x{:}A \vdash^{\mathbf{lf}} \uparrow_x \gamma : \Gamma}}$

and $\mathcal{F} = \dfrac{\overset{\mathcal{F}_1}{\boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} \gamma_2 : \Gamma'}} \quad \boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} M : A[\gamma]}}{\boldsymbol{\Gamma_2 \vdash^{\mathbf{lf}} (\gamma_2, M/x) : (\Gamma', x{:}A)}}$

$\boldsymbol{(\gamma \circ^{\mathbf{lf}} \gamma_2)}$ exists	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\boldsymbol{((\uparrow_x \gamma) \circ^{\mathbf{lf}} (\gamma_2, M/x))}$ exists	by Composition (Def. A.2.3)

□

Lemma B.1.15 (LF-Level Variable under Composed Substitutions)**.**
If $\boldsymbol{\Gamma' \vdash^{\mathrm{lf}} \gamma : \Gamma}$ *and* $\boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \Gamma'}$ *and* $\boldsymbol{\Gamma \vdash^{\mathrm{lf}} x_0 : A_0}$
and $\boldsymbol{x_0[\gamma] = M_0}$*, then* $\boldsymbol{x_0[\gamma \circ^{\mathrm{lf}} \gamma_2] = M_0[\gamma_2]}$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \boldsymbol{\Gamma' \vdash^{\mathrm{lf}} \gamma : \Gamma}$ and $\mathcal{F} :: \boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \Gamma'}$.

Case: $\mathcal{E} = \boldsymbol{\Gamma' \vdash^{\mathrm{lf}} \gamma : \Gamma}$

and $\mathcal{F} = \dfrac{\overset{\mathcal{F}_1}{\boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \Gamma'}} \qquad \boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} A : \mathit{type}}}{\boldsymbol{\Gamma_2, x{:}A \vdash^{\mathrm{lf}} \uparrow_x \gamma_2 : \Gamma'}}\ \textsf{tpLF-SubShift}$

$\boldsymbol{x_0[\gamma] = M_0}$	by assumption
$\boldsymbol{\Gamma \vdash^{\mathrm{lf}} x_0 : A_0}$	by assumption
$\boldsymbol{x_0[\gamma \circ^{\mathrm{lf}} \uparrow_x \gamma_2]}$	
$\quad = \boldsymbol{x_0[\uparrow_x (\gamma \circ^{\mathrm{lf}} \gamma_2)]}$	by Composition (Def. A.2.3)
$\quad = \boldsymbol{x_0[\gamma \circ^{\mathrm{lf}} \gamma_2]}$	by Lemma B.1.5
$\quad = \boldsymbol{M_0[\gamma_2]}$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\quad = \boldsymbol{M_0[\uparrow_x \gamma_2]}$	by Lemma B.1.5

Case: $\mathcal{E} = \dfrac{}{\boldsymbol{\cdot \vdash^{\mathrm{lf}} \cdot : \cdot}}\ \textsf{tpLF-SubBase}$

and $\mathcal{F} = \boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \cdot}$, where $\boldsymbol{\gamma_2 \neq \uparrow_{x''} \gamma_2''}$

$\boldsymbol{x_0[\cdot] = M_0}$	by assumption
$\boldsymbol{x_0[\cdot]}$ is undefined	by Subst. Application (Def. A.2.4)

Contradiction, so this case is vacuously true.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\boldsymbol{\Gamma' \vdash^{\mathrm{lf}} \gamma : \Gamma}} \qquad \boldsymbol{\Gamma' \vdash^{\mathrm{lf}} M_0 : A[\gamma]}}{\boldsymbol{\Gamma' \vdash^{\mathrm{lf}} (\gamma, M_0/x_0) : (\Gamma, x_0{:}A)}}\ \textsf{tpLF-SubInd}$

and $\mathcal{F} = \boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \Gamma'}$, where $\boldsymbol{\gamma_2 \neq \uparrow_{x''} \gamma_2''}$

$\boldsymbol{x_0[\gamma, M_0/x_0] = M_0}$	by Subst. Application (Def. A.2.4)
$\boldsymbol{x_0[(\gamma, M_0/x_0) \circ^{\mathrm{lf}} \gamma_2]}$	
$\quad = \boldsymbol{x_0[(\gamma \circ^{\mathrm{lf}} \gamma_2), M_0[\gamma_2]/x_0]}$	by Composition (Def. A.2.3)
$\quad = \boldsymbol{M_0[\gamma_2]}$	by Subst. Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}} \qquad \boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{M} : \boldsymbol{A}[\gamma]}{\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} (\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) : (\boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A})}\ \textsf{tpLF-SubInd}$, and $\boldsymbol{x} \neq \boldsymbol{x_0}$

and $\mathcal{F} = \boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'}$, where $\boldsymbol{\gamma_2} \neq \uparrow_{\boldsymbol{x''}} \boldsymbol{\gamma_2''}$

$\boldsymbol{x_0}[\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}] = \boldsymbol{x_0}[\boldsymbol{\gamma}] = \boldsymbol{M_0}$ — by assumption and Subst. Application (Def. A.2.4)

$\boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathrm{lf}} \boldsymbol{x_0} : \boldsymbol{A_0}$ — by assumption

$\boldsymbol{\Gamma} \vdash^{\mathrm{lf}} \boldsymbol{x_0} : \boldsymbol{A_0}$ — by inversion using **LF_Var** and **ctxLFAdd** and Application of **LF_Var**

$\boldsymbol{x_0}[(\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}]$
$= \boldsymbol{x_0}[(\boldsymbol{\gamma} \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}), \boldsymbol{M}[\boldsymbol{\gamma_2}]/\boldsymbol{x}]$ — by Composition (Def. A.2.3)
$= \boldsymbol{x_0}[\boldsymbol{\gamma} \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}]$ — by Subst. Application (Def. A.2.4)
$= \boldsymbol{M_0}[\boldsymbol{\gamma_2}]$ — by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}} \qquad \boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{A} : \textit{type}}{\boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathrm{lf}} \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}}\ \textsf{tpLF-SubShift}$

and $\boldsymbol{\mathcal{F}} = \dfrac{\overset{\mathcal{F}_1}{\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'}} \qquad \boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{M} : \boldsymbol{A}[\gamma_2]}{\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} (\boldsymbol{\gamma_2}, \boldsymbol{M}/\boldsymbol{x}) : (\boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A})}\ \textsf{tpLF-SubInd}$

$\boldsymbol{x_0}[\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}] = \boldsymbol{x_0}[\boldsymbol{\gamma}] = \boldsymbol{M_0}$ — by assumption and Lemma B.1.5

$\boldsymbol{\Gamma} \vdash^{\mathrm{lf}} \boldsymbol{x_0} : \boldsymbol{A_0}$ — by assumption

$\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{M_0} : \boldsymbol{A_0}[\boldsymbol{\gamma}]$ — by Lemma B.1.12.1

$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{M_0}[\boldsymbol{\gamma_2}] : \boldsymbol{A_0}[\boldsymbol{\gamma}][\boldsymbol{\gamma_2}]$ — by Lemma B.1.12.1

$\boldsymbol{\gamma_2} \leq (\boldsymbol{\gamma_2}, \boldsymbol{M}/\boldsymbol{x})$ — by **leLFsubAdd**

$\boldsymbol{x_0}[(\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma}) \circ^{\mathrm{lf}} (\boldsymbol{\gamma_2}, \boldsymbol{M}/\boldsymbol{x})]$
$= \boldsymbol{x_0}[\boldsymbol{\gamma} \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}]$ — by Composition (Def. A.2.3)
$= \boldsymbol{M_0}[\boldsymbol{\gamma_2}]$ — by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$= \boldsymbol{M_0}[\boldsymbol{\gamma_2}, \boldsymbol{M}/\boldsymbol{x}]$ — by Lemma B.1.7

□

Lemma B.1.16 (LF-Level Equality w.r.t. Substitution Composition)**.**
If $\Gamma_1 \vdash^{\mathsf{lf}} \gamma_1 : \Gamma_0$ *and* $\Gamma_2 \vdash^{\mathsf{lf}} \gamma_2 : \Gamma_1$*, then*

1. *if* $\Gamma_0 \vdash^{\mathsf{lf}} A : K$*, then* $A[\gamma_1][\gamma_2] = A[\gamma_1 \circ^{\mathbf{lf}} \gamma_2]$.
2. *if* $\Gamma_0 \vdash^{\mathsf{lf}} M : A$*, then* $M[\gamma_1][\gamma_2] = M[\gamma_1 \circ^{\mathbf{lf}} \gamma_2]$.
3. *if* $\Gamma_0 \vdash^{\mathsf{lf}} K : \mathit{kind}$*, then* $K[\gamma_1][\gamma_2] = K[\gamma_1 \circ^{\mathbf{lf}} \gamma_2]$.

Proof. By induction on A, M, and K.

Case: $A = a$

$\Gamma_1 \vdash^{\mathsf{lf}} \gamma_1 : \Gamma_0$	by assumption
$\Gamma_2 \vdash^{\mathsf{lf}} \gamma_2 : \Gamma_1$	by assumption
$\Gamma_0 \vdash^{\mathsf{lf}} a : K$	by assumption
$\cdot \vdash^{\mathsf{lf}} a : K$	by inversion since Σ **sig**
$a = a[\cdot]$	by Lemma B.1.4.1 and Def. of **id**
$\cdot \leq_{\mathsf{lf}} \gamma_1$	by weakening rules
$a[\gamma_1] = a$	by Lemma B.1.6 and above
$\cdot \leq_{\mathsf{lf}} \gamma_2$	by weakening rules
$a[\gamma_2] = a$	by Lemma B.1.6 and above
$a[\gamma_1][\gamma_2] = a$	by above
$(\gamma_1 \circ^{\mathbf{lf}} \gamma_2)$ exists	by Lemma B.1.14
$\cdot \leq_{\mathsf{lf}} (\gamma_1 \circ^{\mathbf{lf}} \gamma_2)$	by weakening rules
$a[\gamma_1 \circ^{\mathbf{lf}} \gamma_2] = a$	by Lemma B.1.6 and above
$a[\gamma_1][\gamma_2] = a[\gamma_1 \circ^{\mathbf{lf}} \gamma_2]$	by above

Case: $A = B\ M$

$\Gamma_1 \vdash^{\mathsf{lf}} \gamma_1 : \Gamma_0$	by assumption
$\Gamma_2 \vdash^{\mathsf{lf}} \gamma_2 : \Gamma_1$	by assumption
$\Gamma_0 \vdash^{\mathsf{lf}} B\ M : K[\mathbf{id}_{\Gamma_0}, M/x]$	by assumption and inversion
$\Gamma_0 \vdash^{\mathsf{lf}} B : \Pi x{:}A'.\ K$	by inversion using **LF_TypeApp**
$\Gamma_0 \vdash^{\mathsf{lf}} M : A'$	by inversion using **LF_TypeApp**
$(B\ M)[\gamma_1][\gamma_2]$	
$= (B[\gamma_1][\gamma_2])\ (M[\gamma_1][\gamma_2])$	by Subst. Application (Def. A.2.4)
$= (B[\gamma_1 \circ^{\mathbf{lf}} \gamma_2])\ (M[\gamma_1 \circ^{\mathbf{lf}} \gamma_2])$	by i.h. on B and M
$= (B\ M)[\gamma_1 \circ^{\mathbf{lf}} \gamma_2]$	by Subst. Application (Def. A.2.4)

Case: $\boldsymbol{A = \Pi x{:}B_1.\ B_2}$

$\boldsymbol{\Gamma_1 \vdash^{\mathrm{lf}} \gamma_1 : \Gamma_0}$ by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \Gamma_1}$ by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathrm{lf}} \Pi x{:}B_1.\ B_2 : type}$ by assumption and inversion
$\boldsymbol{\Gamma_0 \vdash^{\mathrm{lf}} B_1 : type}$ by inversion using **LF_Pi**
$\boldsymbol{\Gamma_0, x{:}B_1 \vdash^{\mathrm{lf}} B_2 : type}$ by inversion using **LF_Pi**
$\boldsymbol{\Gamma_1 \vdash^{\mathrm{lf}} B_1[\gamma_1] : type}$ by Lemma B.1.12.2 and Subst. Application (Def. A.2.4)
$\boldsymbol{\Gamma_1, x{:}B_1[\gamma_1] \vdash^{\mathrm{lf}} (\uparrow_x \gamma_1, x/x) : \Gamma_1, x{:}B_1}$ by typing rules and Lemma B.1.5
$\boldsymbol{\Gamma_1 \vdash^{\mathrm{lf}} B_1[\gamma_1][\gamma_2] : type}$ by Lemma B.1.12.2 and Subst. Application (Def. A.2.4)
$\boldsymbol{\Gamma_1, x{:}B_1[\gamma_1][\gamma_2] \vdash^{\mathrm{lf}} (\uparrow_x \gamma_2, x/x) : \Gamma_1, x{:}B_1[\gamma]}$ by typing rules and Lemma B.1.5

$$
\begin{aligned}
&(\Pi x{:}B_1.\ B_2)[\gamma_1][\gamma_2] \\
&\quad = \Pi x{:}B_1[\gamma_1][\gamma_2].\ B_2[\uparrow_x \gamma_1, x/x][\uparrow_x \gamma_2, x/x] && \text{by Subst. Application (Def. A.2.4)} \\
&\quad = \Pi x{:}B_1[\gamma_1 \circ^{\mathrm{lf}} \gamma_2].\ B_2[\uparrow_x \gamma_1, x/x][\uparrow_x \gamma_2, x/x] && \text{by i.h. on } B_1 \\
&\quad = \Pi x{:}B_1[\gamma_1 \circ^{\mathrm{lf}} \gamma_2].\ B_2[(\uparrow_x \gamma_1, x/x) \circ^{\mathrm{lf}} (\uparrow_x \gamma_2, x/x)] && \text{by i.h. on } B_2 \\
&\quad = \Pi x{:}B_1[\gamma_1 \circ^{\mathrm{lf}} \gamma_2].\ B_2[\uparrow_x (\gamma_1 \circ^{\mathrm{lf}} \gamma_2), x/x] \\
&\qquad\qquad \text{by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)} \\
&\quad = (\Pi x{:}B_1.\ B_2)[\gamma_1 \circ^{\mathrm{lf}} \gamma_2] && \text{by Subst. Application (Def. A.2.4)}
\end{aligned}
$$

. . . . . .

Case: $\boldsymbol{M = x}$

$\boldsymbol{\Gamma_1 \vdash^{\mathrm{lf}} \gamma_1 : \Gamma_0}$ by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \Gamma_1}$ by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathrm{lf}} x : A}$ by assumption
$\boldsymbol{(x{:}A) \text{ in } \Gamma_0}$ by inversion using **LF_Var**
$\boldsymbol{x[\gamma_1] = M_0}$ by inspection of subst. typing rules and Subst. Application (Def. A.2.4)

$$
\begin{aligned}
&x[\gamma_1][\gamma_2] \\
&\quad = M_0[\gamma_2] && \text{by above} \\
&\quad = x[\gamma_1 \circ^{\mathrm{lf}} \gamma_2] && \text{by Lemma B.1.15}
\end{aligned}
$$

Case: $\boldsymbol{M = c}$

$\boldsymbol{\Gamma_1 \vdash^{\mathsf{lf}} \gamma_1 : \Gamma_0}$ by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathsf{lf}} \gamma_2 : \Gamma_1}$ by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathsf{lf}} c : A}$ by assumption
$\boldsymbol{\cdot \vdash^{\mathsf{lf}} c : A}$ by inversion since $\boldsymbol{\Sigma}$ **sig**
$\boldsymbol{c = c[\cdot]}$ by Lemma B.1.4.2 and Def. of **id**
$\boldsymbol{\cdot \leq_{\mathsf{lf}} \gamma_1}$ by weakening rules
$\boldsymbol{c[\gamma_1] = c}$ by Lemma B.1.6 and above
$\boldsymbol{\cdot \leq_{\mathsf{lf}} \gamma_2}$ by weakening rules
$\boldsymbol{c[\gamma_2] = c}$ by Lemma B.1.6 and above
$\boldsymbol{c[\gamma_1][\gamma_2] = c}$ by above
$\boldsymbol{(\gamma_1 \circ^{\mathsf{lf}} \gamma_2)}$ exists by Lemma B.1.14
$\boldsymbol{\cdot \leq_{\mathsf{lf}} (\gamma_1 \circ^{\mathsf{lf}} \gamma_2)}$ by weakening rules
$\boldsymbol{c[\gamma_1 \circ^{\mathsf{lf}} \gamma_2] = c}$ by Lemma B.1.6 and above
$\boldsymbol{c[\gamma_1][\gamma_2] = c[\gamma_1 \circ^{\mathsf{lf}} \gamma_2]}$ by above

Case: $\boldsymbol{M = \lambda x{:}A.\ N}$

$\boldsymbol{\Gamma_1 \vdash^{\mathsf{lf}} \gamma_1 : \Gamma_0}$ by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathsf{lf}} \gamma_2 : \Gamma_1}$ by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathsf{lf}} \lambda x{:}B.\ N : A}$ by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathsf{lf}} B : type}$ by inversion using **LF_Lam**
$\boldsymbol{\Gamma_0, x{:}B \vdash^{\mathsf{lf}} N : A}$ by inversion using **LF_Lam**
$\boldsymbol{\Gamma_1 \vdash^{\mathsf{lf}} B[\gamma_1] : type}$ by Lemma B.1.12.2 and Subst. Application (Def. A.2.4)
$\boldsymbol{\Gamma_1, x{:}B[\gamma_1] \vdash^{\mathsf{lf}} (\uparrow_x \gamma_1, x/x) : \Gamma_1, x{:}B}$ by typing rules and Lemma B.1.5
$\boldsymbol{\Gamma_1 \vdash^{\mathsf{lf}} B[\gamma_1][\gamma_2] : type}$ by Lemma B.1.12.2 and Subst. Application (Def. A.2.4)
$\boldsymbol{\Gamma_1, x{:}B[\gamma_1][\gamma_2] \vdash^{\mathsf{lf}} (\uparrow_x \gamma_2, x/x) : \Gamma_1, x{:}B[\gamma]}$ by typing rules and Lemma B.1.5

$\boldsymbol{(\lambda x{:}B.\ N)[\gamma_1][\gamma_2]}$
$\boldsymbol{= \lambda x{:}B[\gamma_1][\gamma_2].\ N[\uparrow_x \gamma_1, x/x][\uparrow_x \gamma_2, x/x]}$ by Subst. Application (Def. A.2.4)
$\boldsymbol{= \lambda x{:}B[\gamma_1 \circ^{\mathsf{lf}} \gamma_2].\ N[\uparrow_x \gamma_1, x/x][\uparrow_x \gamma_2, x/x]}$ by i.h. on $\boldsymbol{B}$
$\boldsymbol{= \lambda x{:}B[\gamma_1 \circ^{\mathsf{lf}} \gamma_2].\ N[(\uparrow_x \gamma_1, x/x) \circ^{\mathsf{lf}} (\uparrow_x \gamma_2, x/x)]}$ by i.h. on $\boldsymbol{N}$
$\boldsymbol{= \lambda x{:}B[\gamma_1 \circ^{\mathsf{lf}} \gamma_2].\ N[\uparrow_x (\gamma_1 \circ^{\mathsf{lf}} \gamma_2), x/x]}$ by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)
$\boldsymbol{= (\lambda x{:}B.\ N)[\gamma_1 \circ^{\mathsf{lf}} \gamma_2]}$ by Subst. Application (Def. A.2.4)

Case: $\boldsymbol{M = N_1\ N_2}$

$\boldsymbol{\Gamma_1 \vdash^{\mathsf{lf}} \gamma_1 : \Gamma_0}$	by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathsf{lf}} \gamma_2 : \Gamma_1}$	by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathsf{lf}} N_1\ N_2 : B[\mathrm{id}_{\Gamma_0}, N_2/x]}$	by assumption and inversion
$\boldsymbol{\Gamma_0 \vdash^{\mathsf{lf}} N_1 : \Pi x{:}A.\ B}$	by inversion using **LF_App**
$\boldsymbol{\Gamma_0 \vdash^{\mathsf{lf}} N_2 : A}$	by inversion using **LF_App**
$(N_1\ N_2)[\gamma_1][\gamma_2]$	
$= (N_1[\gamma_1][\gamma_2])\ (N_2[\gamma_1][\gamma_2])$	by Subst. Application (Def. A.2.4)
$= (N_1[\gamma_1 \circ^{\mathsf{lf}} \gamma_2])\ (N_2[\gamma_1 \circ^{\mathsf{lf}} \gamma_2])$	by i.h. on N_1 and N_2
$= (N_1\ N_2)[\gamma_1 \circ^{\mathsf{lf}} \gamma_2]$	by Subst. Application (Def. A.2.4)

. . . . . .

Case: $\boldsymbol{K = type}$

$\boldsymbol{\Gamma_1 \vdash^{\mathsf{lf}} \gamma_1 : \Gamma_0}$	by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathsf{lf}} \gamma_2 : \Gamma_1}$	by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathsf{lf}} type : kind}$	by assumption
$(\gamma_1 \circ^{\mathsf{lf}} \gamma_2)$ exists	by Lemma B.1.14
$type[\gamma_1][\gamma_2]$	
$= type$	by Subst. Application (Def. A.2.4)
$= type[\gamma_1 \circ^{\mathsf{lf}} \gamma_2]$	by Subst. Application (Def. A.2.4)

Case: $\boldsymbol{K = \Pi x{:}A.\ K'}$

$\boldsymbol{\Gamma_1 \vdash^{\mathrm{lf}} \gamma_1 : \Gamma_0}$ by assumption
$\boldsymbol{\Gamma_2 \vdash^{\mathrm{lf}} \gamma_2 : \Gamma_1}$ by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathrm{lf}} \Pi x{:}A.\ K' : kind}$ by assumption
$\boldsymbol{\Gamma_0 \vdash^{\mathrm{lf}} A : type}$ by inversion
$\boldsymbol{\Gamma_0, x{:}A \vdash^{\mathrm{lf}} K' : kind}$ by inversion
$\boldsymbol{\Gamma_1 \vdash^{\mathrm{lf}} A[\gamma_1] : type}$ by Lemma B.1.12.2 and Subst. Application (Def. A.2.4)
$\boldsymbol{\Gamma_1, x{:}A[\gamma_1] \vdash^{\mathrm{lf}} (\uparrow_x \gamma_1, x/x) : \Gamma_1, x{:}A}$ by typing rules and Lemma B.1.5
$\boldsymbol{\Gamma_1 \vdash^{\mathrm{lf}} A[\gamma_1][\gamma_2] : type}$ by Lemma B.1.12.2 and Subst. Application (Def. A.2.4)
$\boldsymbol{\Gamma_1, x{:}A[\gamma_1][\gamma_2] \vdash^{\mathrm{lf}} (\uparrow_x \gamma_2, x/x) : \Gamma_1, x{:}A[\gamma]}$ by typing rules and Lemma B.1.5

$$
\begin{aligned}
&(\Pi x{:}A.\ K)[\gamma_1][\gamma_2] \\
&\quad = \Pi x{:}A[\gamma_1][\gamma_2].\ K'[\uparrow_x \gamma_1, x/x][\uparrow_x \gamma_2, x/x] && \text{by Subst. Application (Def. A.2.4)} \\
&\quad = \Pi x{:}A[\gamma_1 \circ^{\mathrm{lf}} \gamma_2].\ K'[\uparrow_x \gamma_1, x/x][\uparrow_x \gamma_2, x/x] && \text{by i.h. on } A \\
&\quad = \Pi x{:}A[\gamma_1 \circ^{\mathrm{lf}} \gamma_2].\ K'[(\uparrow_x \gamma_1, x/x) \circ^{\mathrm{lf}} (\uparrow_x \gamma_2, x/x)] && \text{by i.h. on } K' \\
&\quad = \Pi x{:}A[\gamma_1 \circ^{\mathrm{lf}} \gamma_2].\ K'[\uparrow_x (\gamma_1 \circ^{\mathrm{lf}} \gamma_2), x/x] \\
&\qquad\qquad \text{by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)} \\
&\quad = (\Pi x{:}A.\ K')[\gamma_1 \circ^{\mathrm{lf}} \gamma_2] && \text{by Subst. Application (Def. A.2.4)}
\end{aligned}
$$

□

Lemma B.1.17 (LF-Level Substitution Composition is Well-Typed)**.**
If $\boldsymbol{\Gamma}\ \mathsf{ctx}_{\mathsf{lf}}$ *and* $\boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}$ *and* $\boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'}$*, then* $\boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} (\boldsymbol{\gamma} \circ^{\mathbf{lf}} \boldsymbol{\gamma_2}) : \boldsymbol{\Gamma}$.

Proof. By induction lexicographically on $\mathcal{E} :: \boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}$ and $\mathcal{F} :: \boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'}$.

Case: $\mathcal{E} = \boldsymbol{\Gamma'} \vdash^{\mathsf{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}$

and $\mathcal{F} = \dfrac{\mathcal{F}_1 \quad \boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'} \qquad \boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}}{\boldsymbol{\Gamma_2}, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathsf{lf}} \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'}}\ \mathsf{tpLF\text{-}SubShift}$

$\boldsymbol{\Gamma}\ \mathsf{ctx}_{\mathsf{lf}}$	by assumption
$\boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{\gamma} \circ^{\mathbf{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma}$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$	by assumption
$\boldsymbol{\Gamma_2}, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathsf{lf}} \uparrow_{\boldsymbol{x}} (\boldsymbol{\gamma} \circ^{\mathbf{lf}} \boldsymbol{\gamma_2}) : \boldsymbol{\Gamma}$	by **tpLF-SubShift**
$\boldsymbol{\Gamma_2}, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathsf{lf}} \boldsymbol{\gamma} \circ^{\mathbf{lf}} (\uparrow_{\boldsymbol{x}} \boldsymbol{\gamma_2}) : \boldsymbol{\Gamma}$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash^{\mathsf{lf}} \cdot : \cdot}\ \mathsf{tpLF\text{-}SubBase}$

and $\mathcal{F} = \boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{\gamma_2} : \cdot$, where $\boldsymbol{\gamma_2} \neq \uparrow_{\boldsymbol{x''}} \boldsymbol{\gamma_2''}$

$\boldsymbol{\Gamma}\ \mathsf{ctx}_{\mathsf{lf}}$	by assumption
$\boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} \boldsymbol{\gamma_2} : \cdot$	by assumption ($\mathcal{F}$)
$\boldsymbol{\Gamma_2} \vdash^{\mathsf{lf}} (\cdot \circ^{\mathbf{lf}} \boldsymbol{\gamma_2}) : \cdot$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}} \qquad \boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{M} : \boldsymbol{A}[\gamma]}{\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} (\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) : (\boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A})}\ \textsf{tpLF-SubInd}$

and $\mathcal{F} = \boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'}$, where $\boldsymbol{\gamma_2} \neq \uparrow_{\boldsymbol{x''}} \boldsymbol{\gamma_2''}$

$(\boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A})$ $\textsf{ctx}_{\textsf{lf}}$	by assumption
$\boldsymbol{\Gamma}$ $\textsf{ctx}_{\textsf{lf}}$	by inversion using **ctxLFAdd**
$\boldsymbol{\Gamma} \vdash^{\mathrm{lf}} \boldsymbol{A} : \boldsymbol{type}$	by inversion using **ctxLFAdd**
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} (\gamma \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}) : \boldsymbol{\Gamma}$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$
$\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{M} : \boldsymbol{A}[\gamma]$	by assumption
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{M}[\boldsymbol{\gamma_2}] : \boldsymbol{A}[\gamma][\boldsymbol{\gamma_2}]$	by Lemma B.1.12.1
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{A}[\gamma][\boldsymbol{\gamma_2}] : \boldsymbol{type}$	by Lemma B.1.13
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{M}[\boldsymbol{\gamma_2}] : \boldsymbol{A}[\gamma \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}]$	by Lemma B.1.16
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} ((\gamma \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}), \boldsymbol{M}[\boldsymbol{\gamma_2}]/\boldsymbol{x}) : \boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A}$	by **tpLF-SubInd**
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} ((\boldsymbol{\gamma}, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathrm{lf}} \boldsymbol{\gamma_2}) : \boldsymbol{\Gamma}, \boldsymbol{x}{:}\boldsymbol{A}$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}} \qquad \boldsymbol{\Gamma'} \vdash^{\mathrm{lf}} \boldsymbol{A} : \boldsymbol{type}}{\boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathrm{lf}} \uparrow_{\boldsymbol{x}} \boldsymbol{\gamma} : \boldsymbol{\Gamma}}\ \textsf{tpLF-SubShift}$

and $\mathcal{F} = \dfrac{\overset{\mathcal{F}_1}{\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma'}} \qquad \boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \boldsymbol{M} : \boldsymbol{A}[\gamma]}{\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} (\boldsymbol{\gamma_2}, \boldsymbol{M}/\boldsymbol{x}) : (\boldsymbol{\Gamma'}, \boldsymbol{x}{:}\boldsymbol{A})}\ \textsf{tpLF-SubInd}$

$\boldsymbol{\Gamma}$ $\textsf{ctx}_{\textsf{lf}}$	by assumption
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} \gamma \circ^{\mathrm{lf}} \boldsymbol{\gamma_2} : \boldsymbol{\Gamma}$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\boldsymbol{\Gamma_2} \vdash^{\mathrm{lf}} (\uparrow_{\boldsymbol{x}} \gamma) \circ^{\mathrm{lf}} (\boldsymbol{\gamma_2}, \boldsymbol{M}/\boldsymbol{x}) : \boldsymbol{\Gamma}$	by Composition (Def. A.2.3)

□

B.2 Alternate Formulation of World Inclusion

For the proofs, it is easier to use the following definition of world inclusion, which we will show is equivalent to the one we defined (Lemmas B.9.1 and B.9.2). The "cast" operator that follows is also used in the proof of liveness (Lemma B.20.4).

Worlds will only contain declarations of type $\boldsymbol{A}$ or $\boldsymbol{A}^{\#}$. Substitutions cannot traverse world boundaries, so we define a special cast operation which throws out computation-level information and the distinction between uninstantiable parameters and instantiable parameters, but saves the distinction between $\boldsymbol{A}$ and $\boldsymbol{A}^{\#}$.

This alternative formulation is particularly important for utilizing the most-general unifier from the coverage rules for proving liveness (Lemma B.20.4).

Definition B.2.1 (Casting Saving Parameters)**.**

$$\|\Omega\|^{\nabla} = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ \|\Omega'\|^{\nabla}, \boldsymbol{x}{:}\boldsymbol{A} & \text{if } \Omega = \Omega', \boldsymbol{x}{:}\boldsymbol{A} \\ \|\Omega'\|^{\nabla}, \boldsymbol{x}{:}\boldsymbol{A}^{\#} & \text{if } \Omega = \Omega', \boldsymbol{x}{:}\boldsymbol{A}^{\#} \\ \|\Omega'\|^{\nabla} & \text{if } \Omega = \Omega', u{:}\tau \\ \|\Omega'\|^{\nabla}, \boldsymbol{x}{:}\boldsymbol{A}^{\#} & \text{if } \Omega = \Omega', \boldsymbol{x} \overset{\nabla}{\in} \boldsymbol{A}^{\#} \\ \|\Omega'\|^{\nabla} & \text{if } \Omega = \Omega', u \overset{\nabla}{\in} \mathcal{W} \end{cases}$$

We similarly define a casting operation on substitutions that results in computation-level substitutions rather than LF ones.

$$\begin{array}{lcl} \|\cdot\|^{\nabla} & = & \cdot \\ \|\omega, \boldsymbol{M}/\boldsymbol{x}\|^{\nabla} & = & \|\omega\|^{\nabla}, \boldsymbol{M}/\boldsymbol{x} \\ \|\omega, e/u\|^{\nabla} & = & \|\omega\|^{\nabla} \end{array} \qquad \begin{array}{lcl} \| \uparrow_{\boldsymbol{x}} \omega\|^{\nabla} & = & \uparrow_{\boldsymbol{x}} \|\omega\|^{\nabla} \\ \| \uparrow_{u} \omega\|^{\nabla} & = & \|\omega\|^{\nabla} \end{array}$$

$$\frac{\|\Omega\|^{\nabla} \vdash \omega : \Omega'}{(\Omega, \boldsymbol{A'}[\omega]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesAltYes}$$

$$\frac{(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}'}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesAltOther}$$

$$\frac{}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} *}\ \mathsf{includesAltAny}$$

B.3 Meta-Theory: Basics 1

Lemma B.3.1 (Typing Implies Well-Formed Context)**.**

- *If* $\Omega \vdash e \in \delta$*, then* Ω ctx*.*
- *If* $\Omega \vdash c \in \tau$*, then* Ω ctx*.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash e \in \delta$ and $\mathcal{E} :: \Omega \vdash c \in \tau$. This proof is very straightforward and holds either by the premises or by the i.h. on one of the premises. After the i.h. we need to do inversion using ctxAdd for rules new, newW, fix, cEps, and cNew. □

Lemma B.3.2 (Valid Substitution Implies Good Context)**.**
If $\Omega' \vdash \omega : \Omega$*, then* Ω' ctx*.*

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$. This is straightforward via calls to the i.h. and the rules for valid contexts, Ω' ctx. □

Lemma B.3.3 (Computation Level Doesn't Influence Well-Formedness of Types)**.** *This lemma shows that only LF objects can be indexed in types δ.*

1. *If $\Omega_1, u'{\in}\tau', \Omega_2 \vdash \delta$ wff, then $\Omega_1, \Omega_2 \vdash \delta$ wff.*
2. *If $\Omega_1, u'{\overset{\triangledown}{\in}}\mathcal{W}', \Omega_2 \vdash \delta$ wff, then $\Omega_1, \Omega_2 \vdash \delta$ wff.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash \delta$ wff. We only show Part 1 as Part 2 directly follows the same proof. We present these two lemmas together, but they do not need to be mutually recursive.

Case: $\mathcal{E} = \dfrac{\|\Omega_1, u'{\in}\tau', \Omega_2\| \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \boldsymbol{A}\ \mathsf{wff}}\ \mathsf{LF\_wff}$

$\|\Omega_1, u'{\in}\tau', \Omega_2\| \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$	by assumption
$\|\Omega_1, u'{\in}\tau', \Omega_2\| = \|\Omega_1, \Omega_2\|$	by Casting (Def. A.2.1)
$\|\Omega_1, \Omega_2\| \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$	by above
$\Omega_1, \Omega_2 \vdash \boldsymbol{A}\ \mathsf{wff}$	by $\mathsf{LF\_wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, u'{\in}\tau', \Omega_2 \vdash \boldsymbol{A}\ \mathsf{wff}\end{array}}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}}\ \mathsf{param\_wff}$

$\Omega_1, \Omega_2 \vdash \boldsymbol{A}\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega_1, \Omega_2 \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$	by $\mathsf{param\_wff}$

Case: $\mathcal{E} = \dfrac{}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \mathrm{unit}\ \mathsf{wff}}\ \mathrm{unit}\mathsf{wff}$

$\Omega_1, \Omega_2 \vdash \mathrm{unit}\ \mathsf{wff}$	by $\mathrm{unit}\mathsf{wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, u'{\in}\tau', \Omega_2 \vdash \tau\ \mathsf{wff} \qquad \mathcal{W}\ \mathsf{world}\end{array}}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \forall nil.\ \tau\ \mathsf{wff}}\ \forall_b\mathsf{wff}$

$\Omega_1, \Omega_2 \vdash \tau\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega_1, \Omega_2 \vdash \forall nil.\ \tau\ \mathsf{wff}$	by $\forall_b\mathsf{wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega_1, u'{\in}\tau', \Omega_2 \vdash \delta_1\ \mathsf{wff} & \Omega_1, u'{\in}\tau', \Omega_2, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}\end{array}}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \forall\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}\ \forall_i\mathsf{wff}$

$\Omega_1, \Omega_2 \vdash \delta_1\ \mathsf{wff}$ by i.h. on $\mathcal{E}_1$
$\Omega_1, \Omega_2, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}$ by i.h. on $\mathcal{E}_2$
$\Omega_1, \Omega_2 \vdash \forall\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}$ by $\forall_i\mathsf{wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega_1, u'{\in}\tau', \Omega_2 \vdash \delta\ \mathsf{wff} & \Omega_1, u'{\in}\tau', \Omega_2, \alpha{\in}\delta \vdash \tau\ \mathsf{wff}\end{array}}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}}\ \exists\mathsf{wff}$

$\Omega_1, \Omega_2 \vdash \delta\ \mathsf{wff}$ by i.h. on $\mathcal{E}_1$
$\Omega_1, \Omega_2, \alpha{\in}\delta \vdash \tau\ \mathsf{wff}$ by i.h. on $\mathcal{E}_2$
$\Omega_1, \Omega_2 \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}$ by $\exists\mathsf{wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega_1, u'{\in}\tau', \Omega_2 \vdash \boldsymbol{A}^{\#}\ \mathsf{wff} & \Omega_1, u'{\in}\tau', \Omega_2, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}\end{array}}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}}\ \nabla\mathsf{wff}$

$\Omega_1, \Omega_2 \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$ by i.h. on $\mathcal{E}_1$
$\Omega_1, \Omega_2, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}$ by i.h. on $\mathcal{E}_2$
$\Omega_1, \Omega_2 \vdash \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}$ by $\nabla\mathsf{wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \mathcal{W}\ \mathsf{world} \qquad \Omega_1, u'{\in}\tau', \Omega_2 \vdash \tau\ \mathsf{wff}\end{array}}{\Omega_1, u'{\in}\tau', \Omega_2 \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}}\ \nabla_{\text{world}}\mathsf{wff}$

$\Omega_1, \Omega_2 \vdash \tau\ \mathsf{wff}$ by i.h. on $\mathcal{E}_1$
$\mathcal{W}\ \mathsf{world}$ by assumption
$\Omega_1, \Omega_2 \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}$ by $\nabla_{\text{world}}\mathsf{wff}$

□

B.4 Meta-Theory: Casting 1

Lemma B.4.1 (Casting Context to LF)**.**
If Ω ctx*, then* $\|\Omega\|$ $\mathbf{ctx_{lf}}$.

Proof. By induction on $\mathcal{E} :: \Omega$ ctx. This proof is straightforward. The conditions for Ω ctx appeal to the well-formedness judgment $\Omega \vdash \delta$ wff. If $\delta = \boldsymbol{A}$ or $\delta = \boldsymbol{A}^{\#}$, then by inversion $\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$ and we can build up the context using rules for LF valid contexts, $\boldsymbol{\Gamma}$ $\mathbf{ctx_{lf}}$. All other cases are thrown out in $\|\Omega\|$ by the definition of casting (Definition A.2.1).

□

Lemma B.4.2 (Relationship between Castings)**.**
$\|\Omega\| = \|\|\Omega\|^{\nabla}\|$ *and* $\|\omega\| = \|\|\omega\|^{\nabla}\|$.

Proof. Straightforward by induction on Ω or ω utilizing the appropriate definitions (Definitions A.2.1, A.2.2, and B.2.1).

□

Lemma B.4.3 (Casting Param Context is Well-Formed)**.**
If Ω ctx*, then* $\|\Omega\|^{\nabla}$ ctx*.*

Proof. By induction on $\mathcal{E} :: \Omega$ ctx. This proof is straightforward. The conditions for Ω ctx appeal to the well-formedness judgment $\Omega \vdash \delta$ wff. If $\delta = \boldsymbol{A}$ or $\delta = \boldsymbol{A}^{\#}$, then by inversion $\|\Omega\| \vdash^{\mathsf{f}} \boldsymbol{A} : \boldsymbol{type}$, and $\|\Omega\| = \|\|\Omega\|^{\nabla}\|$ by Lemma B.4.2. We can therefore build up the context again. All other cases are thrown out in $\|\Omega\|^{\nabla}$ by definition (Def B.2.1). □

Lemma B.4.4 ($\leq_{\mathcal{W}}$ and $\leq$ Relations between Ω and $\boldsymbol{\Gamma}$)**.**

- *If* $\Omega \leq_{\mathcal{W}} \Omega'$*, then* $\|\Omega\| \leq_{\mathbf{lf}} \|\Omega'\|$*.*
- *If* $\Omega \leq \Omega'$*, then* $\|\Omega\| \leq_{\mathbf{lf}} \|\Omega'\|$*.*

Proof. By induction on $\mathcal{E} :: \Omega \leq_{\mathcal{W}} \Omega'$ or $\mathcal{E} :: \Omega \leq \Omega'$ utilizing the definition of $\|\Omega\|$. □

Lemma B.4.5 ($\leq_{\mathcal{W}}$ Relation to id)**.**
If $\Omega \leq_{\mathcal{W}} \Omega'$*, then* $\mathrm{id}_{\Omega} \leq \mathrm{id}_{\Omega'}$*.*

Proof. By induction on $\mathcal{E} :: \Omega \leq_{\mathcal{W}} \Omega'$.

□

Lemma B.4.6 (Id Property between Ω and $\boldsymbol{\Gamma}$)**.**
$\|\mathrm{id}_{\Omega}\| = \mathbf{id}_{(\|\Omega\|)}$.

Proof. By induction on Ω using the appropriate definitions.

□

We will now prove that $\|\omega\|$ and behaves appropriately.

Lemma B.4.7 (Casting Substitution to LF)**.**
If $\Omega' \vdash \omega : \Omega$*, then* $\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega\| : \|\Omega\|$.

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$.

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

$\|\cdot\| = \cdot$	by Casting (Defs. A.2.1 and A.2.2)
$\|\cdot\| \vdash^{\mathsf{lf}} \|\cdot\| : \|\cdot\|$	by Definition and **tpLF-SubBase**

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

Subcase: $(\alpha{\in}\delta)$ is $(\boldsymbol{x}{\in}\boldsymbol{A})$

$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega\| : \|\Omega\|$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash e \in \boldsymbol{A}[\omega]$	by assumption
$\Omega' \vdash e \in \boldsymbol{A}[\|\omega\|]$	by Subst. Application (Def. A.2.4)
$\boldsymbol{e = M}$ and $\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{M : A}[\|\omega\|]$	by inversion using isLF
$\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{(\|\omega\|, e/x) : (\|\Omega\|, x{:}A)}$	by **tpLF-SubInd**
$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega, e/\boldsymbol{x}\| : \|\Omega, \boldsymbol{x}{\in}\boldsymbol{A}\|$	by Casting (Defs. A.2.1 and A.2.2)
$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega, e/\alpha\| : \|\Omega, \alpha{\in}\delta\|$	by above

Subcase: $(\alpha{\in}\delta)$ is $(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$

$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega\| : \|\Omega\|$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash e \in \boldsymbol{A}^{\#}[\omega]$	by assumption
$\Omega' \vdash e \in \boldsymbol{A}[\|\omega\|]^{\#}$	by Subst. Application (Def. A.2.4)
$e = \boldsymbol{x'}$ and $(\boldsymbol{x'}{\in}\boldsymbol{A}[\|\omega\|]^{\#})$ or $(\boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}[\|\omega\|]^{\#})$ in Ω'	by inversion using $\mathsf{var}^{\#}$
$\Omega'\ \mathsf{ctx}$	by Lemma B.3.2 on $\mathcal{E}_1$
$\|\Omega'\|\ \mathbf{ctx_{lf}}$	by Lemma B.4.1
$\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{x' : A}[\|\omega\|]$	by Casting (Def. A.2.1) and **LF_Var**
$\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{(\|\omega\|,\ e/x) : (\|\Omega\|, x{:}A)}$	by **tpLF-SubInd**
$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega, e/\boldsymbol{x}\| : \|\Omega, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}\|$	by Casting (Defs. A.2.1 and A.2.2)
$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega, e/\alpha\| : \|\Omega, \alpha{\in}\delta\|$	by above

Subcase: $(\alpha{\in}\delta)$ is $(u{\in}\tau)$

$\|\Omega'\| \vdash^{\mathsf{f}} \|\omega\| : \|\Omega\|$ by i.h. on $\mathcal{E}_1$
$\|\Omega, \alpha{\in}\delta\| = \|\Omega\|$ by Def. of Casting (Def. A.2.1)
$\|\omega, e/\alpha\| = \|\omega\|$ by Def. of Casting (Def. A.2.2)
$\|\Omega'\| \vdash^{\mathsf{f}} \|\omega, e/\alpha\| : \|\Omega, \alpha{\in}\delta\|$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

$\Omega'\ \mathsf{ctx}$ by Lemma B.3.2 on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}$ by assumption
$\|\Omega'\| \vdash^{\mathsf{f}} \boldsymbol{A}[\|\omega\|] : \boldsymbol{type}$ by Subst. Application (Def. A.2.4) and inversion using $\mathsf{param\_wff}$ and $\mathsf{LF\_wff}$
$\|\Omega', \boldsymbol{x'}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega]\| = \|\Omega'\|, \boldsymbol{x'}{:}\boldsymbol{A}[\|\omega\|]$ by Def. of Casting (Def. A.2.1) and Subst. Application (Def. A.2.4)
$\|\Omega'\|\ \mathbf{ctx}_{\mathsf{lf}}$ by Lemma B.4.1
$(\|\Omega'\|, \boldsymbol{x'}{:}\boldsymbol{A}[\|\omega\|])\ \mathbf{ctx}_{\mathsf{lf}}$ by **ctxLFAdd**
$\|\Omega'\| \vdash^{\mathsf{f}} \|\omega\| : \|\Omega\|$ by i.h. on $\mathcal{E}_1$
$\|\Omega'\|, \boldsymbol{x'}{:}\boldsymbol{A}[\|\omega\|] \vdash^{\mathsf{f}} \uparrow_{\boldsymbol{x'}} \|\omega\| : \|\Omega\|$ by **tpLF-SubShift**
$\|\Omega'\|, \boldsymbol{x'}{:}\boldsymbol{A}[\|\omega\|] \vdash^{\mathsf{f}} \boldsymbol{x'} : \boldsymbol{A}[\|\omega\|]$ by **LF_Var**
$\boldsymbol{A}[\uparrow_{\boldsymbol{x'}} \|\omega\|] = \boldsymbol{A}[\|\omega\|]$ by Lemma B.1.5
$\|\Omega'\|, \boldsymbol{x'}{:}\boldsymbol{A}[\|\omega\|] \vdash^{\mathsf{f}} (\uparrow_{\boldsymbol{x'}} \|\omega\|), \boldsymbol{x'}/\boldsymbol{x} : \|\Omega\|, \boldsymbol{x}{:}\boldsymbol{A}$ by **tpLF-SubInd**
$\|\Omega', \boldsymbol{x'}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega]\| \vdash^{\mathsf{f}} \| \uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}\| : \|\Omega, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}\|$
by above and Casting (Defs A.2.1 and A.2.2)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u'{\overset{\triangledown}{\in}}\mathcal{W}) \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u{\overset{\triangledown}{\in}}\mathcal{W})}\ \mathsf{tpSubWorld}$

$\|\Omega'\| \vdash \|\omega\| : \|\Omega\|$ by i.h. on $\mathcal{E}_1$
$\| \uparrow_{u'} \omega, u'/u\| = \|\omega\|$ by Casting (Def. A.2.2)
$\|\Omega', u'{\overset{\triangledown}{\in}}\mathcal{W}\| = \|\Omega'\|$ by Casting (Def. A.2.1)
$\|\Omega, u{\overset{\triangledown}{\in}}\mathcal{W}\| = \|\Omega\|$ by Casting (Def. A.2.1)
$\|\Omega', u'{\overset{\triangledown}{\in}}\mathcal{W}\| \vdash \| \uparrow_{u'} \omega, u'/u\| : \|\Omega, u{\overset{\triangledown}{\in}}\mathcal{W}\|$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta \ \mathsf{wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

Subcase: $\alpha = u$ and $\delta = \tau$

$\|\Omega'\| \vdash^{\mathsf{f}} \|\omega\| : \|\Omega\|$	by i.h. on $\mathcal{E}_1$
$\| \uparrow_\alpha \omega\| = \|\omega\|$	by Casting (Def. A.2.2)
$\|\Omega', \alpha{\in}\delta\| = \|\Omega'\|$	by Casting (Def. A.2.1)
$\|\Omega', \alpha{\in}\delta\| \vdash^{\mathsf{f}} \| \uparrow_\alpha \omega\| : \|\Omega\|$	by above

Subcase: $\alpha = \boldsymbol{x}$ and $\delta = \boldsymbol{A}$ or $\boldsymbol{A}^{\#}$

$\|\Omega'\| \vdash^{\mathsf{f}} \|\omega\| : \|\Omega\|$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \delta \ \mathsf{wff}$	by assumption
$\|\Omega', \alpha{\in}\delta\| = \|\Omega'\|, \boldsymbol{x}{:}\boldsymbol{A}$	by Casting (Def. A.2.1)
$\|\Omega'\| \vdash^{\mathsf{f}} \boldsymbol{A} : \boldsymbol{type}$	by inversion using $\mathsf{LF\_wff}$
	(possibly first with $\mathsf{param\_wff}$ if $\delta = \boldsymbol{A}^{\#}$)
$\|\Omega'\|, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathsf{f}} \uparrow_{\boldsymbol{x}} \|\omega\| : \|\Omega\|$	by **tpLF-SubShift**
$\| \uparrow_\alpha \omega\| = \uparrow_{\boldsymbol{x}} \|\omega\|$	by Casting (Def. A.2.2)
$\|\Omega', \alpha{\in}\delta\| \vdash^{\mathsf{f}} \| \uparrow_\alpha \omega\| : \|\Omega\|$	by above

□

Lemma B.4.8 (Casting Substitution with Params)**.**
If $\Omega' \vdash \omega : \Omega$*, then* $\|\Omega'\|^{\nabla} \vdash \|\omega\|^{\nabla} : \|\Omega\|^{\nabla}$.

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$.

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

$\|\cdot\|^{\nabla} = \cdot$	by Casting (Defs. A.2.1 and B.2.1)
$\|\cdot\|^{\nabla} \vdash \|\cdot\|^{\nabla} : \|\cdot\|^{\nabla}$	by Definition and $\mathsf{tpSubBase}$

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

Subcase: $(\alpha{\in}\delta)$ is $(\boldsymbol{x}{\in}\boldsymbol{A})$

$\|\Omega'\|^{\nabla} \vdash \|\omega\|^{\nabla} : \|\Omega\|^{\nabla}$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash e \in \boldsymbol{A}[\omega]$	by assumption
$\Omega' \vdash e \in \boldsymbol{A}[\|\omega\|]$	by Subst. Application (Def. A.2.4)
$e = \boldsymbol{M}$ and $\|\Omega'\| \vdash^{\mathsf{uf}} \boldsymbol{M} : \boldsymbol{A}[\|\omega\|]$	by inversion using isLF
$\Omega'\ \mathsf{ctx}$	by inversion using isLF
$\|\Omega'\|^{\nabla}\ \mathsf{ctx}$	by Lemma B.4.3
$\|\Omega'\| = \|\|\Omega'\|^{\nabla}\|$ and $\|\omega\| = \|\|\omega\|^{\nabla}\|$	by Lemma B.4.2
$\|\Omega'\|^{\nabla} \vdash \boldsymbol{M} \in \boldsymbol{A}[\|\omega\|^{\nabla}]$	by isLF and Subst. App.
$\|\Omega'\|^{\nabla} \vdash \|\omega\|^{\nabla}, \boldsymbol{M}/\boldsymbol{x} : \|\Omega\|^{\nabla}, \boldsymbol{x}{\in}\boldsymbol{A}$	by $\mathsf{tpSubInd}$
$\|\Omega'\|^{\nabla} \vdash \|\omega, e/\alpha\|^{\nabla} : \|\Omega, \alpha{\in}\delta\|^{\nabla}$	by Casting (Def. B.2.1) and above

Subcase: $(\alpha{\in}\delta)$ is $(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$

$\|\Omega'\|^{\nabla} \vdash \|\omega\|^{\nabla} : \|\Omega\|^{\nabla}$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash e \in \boldsymbol{A}^{\#}[\omega]$	by assumption
$\Omega' \vdash e \in \boldsymbol{A}[\|\omega\|]^{\#}$	by Subst. Application (Def. A.2.4)
$e = \boldsymbol{x'}$ and Ω' ctx and	
$(\boldsymbol{x'}{\in}\boldsymbol{A}[\|\omega\|]^{\#})$ or $(\boldsymbol{x'}{\overset{\nabla}{\in}}\boldsymbol{A}[\|\omega\|]^{\#})$ in Ω'	by inversion using $\mathsf{var}^{\#}$
$\|\Omega'\|^{\nabla}$ ctx	by Lemma B.4.3
$(\boldsymbol{x'}{\in}\boldsymbol{A}[\|\omega\|]^{\#})$ in $\|\Omega'\|^{\nabla}$	by Casting (Def. B.2.1)
$\|\Omega'\|^{\nabla} \vdash \boldsymbol{x'} \in \boldsymbol{A}[\|\omega\|]^{\#}$	by $\mathsf{var}^{\#}$
$\boldsymbol{A}[\|\omega\|]^{\#} = \boldsymbol{A}^{\#}[\|\omega\|^{\nabla}]$	by Subst. App. and Lemma B.4.2
$\|\Omega'\|^{\nabla} \vdash \|\omega\|^{\nabla}, \boldsymbol{x'}/\boldsymbol{x} : \|\Omega\|^{\nabla}, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}$	by $\mathsf{tpSubInd}$
$\|\Omega'\|^{\nabla} \vdash \|\omega, e/\alpha\|^{\nabla} : \|\Omega, \alpha{\in}\delta\|^{\nabla}$	by Casting (Def. B.2.1) and above

Subcase: $(\alpha{\in}\delta)$ is $(u{\in}\tau)$

$\|\Omega'\|^{\nabla} \Vdash^{\mathsf{f}} \|\omega\|^{\nabla} \mathbf{:} \|\Omega\|^{\nabla}$	by i.h. on $\mathcal{E}_1$
$\|\Omega, \alpha{\in}\delta\|^{\nabla} = \|\Omega\|^{\nabla}$	by Def. of Casting (Def. B.2.1)
$\|\omega, e/\alpha\|^{\nabla} = \|\omega\|^{\nabla}$	by Def. of Casting (Def. B.2.1)
$\|\Omega'\|^{\nabla} \Vdash^{\mathsf{f}} \|\omega, e/\alpha\|^{\nabla} \mathbf{:} \|\Omega, \alpha{\in}\delta\|^{\nabla}$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'}^{\triangledown}\!\in\!\boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x}^{\triangledown}\!\in\!\boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

$\Omega'\ \mathsf{ctx}$ by Lemma B.3.2 on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}$ by assumption
$\Omega' \vdash \boldsymbol{A}[\|\omega\|]^{\#}\ \mathsf{wff}$ by Subst. Application (Def. A.2.4)
$\|\Omega'\| \vdash^{\mathsf{f}} \boldsymbol{A}[\|\omega\|]$ **:** ***type*** by inversion using $\mathsf{param\_wff}$ and $\mathsf{LF\_wff}$
$\|\Omega'\|^{\nabla}\ \mathsf{ctx}$ by Lemma B.4.3
$\|\Omega'\| = \|\|\Omega'\|^{\nabla}\|$ and $\|\omega\| = \|\|\omega\|^{\nabla}\|$ by Lemma B.4.2
$\|\Omega'\|^{\nabla} \vdash \boldsymbol{A}^{\#}[\|\omega\|^{\nabla}]\ \mathsf{wff}$ by Subst. App. and $\mathsf{LF\_wff}$ and $\mathsf{param\_wff}$
$(\|\Omega'\|^{\nabla}, \boldsymbol{x'}\!\in\!\boldsymbol{A}^{\#}[\|\omega\|^{\nabla}])\ \mathsf{ctx}$ by ctxAdd
$(\|\Omega'\|^{\nabla}, \boldsymbol{x'}\!\in\!\boldsymbol{A}^{\#}[\|\omega\|^{\nabla}]) \vdash \boldsymbol{x'} \in \boldsymbol{A}^{\#}[\|\omega\|^{\nabla}]$ by $\mathsf{var}^{\#}$
$\|\Omega'\|^{\nabla} \vdash \|\omega\|^{\nabla} : \|\Omega\|^{\nabla}$ by i.h. on $\mathcal{E}_1$
$\|\Omega'\|^{\nabla}, \boldsymbol{x'}\!\in\!\boldsymbol{A}^{\#}[\|\omega\|^{\nabla}] \vdash \uparrow_{\boldsymbol{x'}} \|\omega\|^{\nabla} : \|\Omega\|^{\nabla}$ by $\mathsf{tpSubShift}$
$\boldsymbol{A}^{\#}[\uparrow_{\boldsymbol{x'}} \|\omega\|^{\nabla}] = \boldsymbol{A}[\|\omega\|]^{\#} = \boldsymbol{A}^{\#}[\|\omega\|^{\nabla}]$
by Lemma B.1.5, Subst. App., and Casting
$\|\Omega'\|^{\nabla}, \boldsymbol{x'}\!\in\!\boldsymbol{A}^{\#}[\|\omega\|^{\nabla}] \vdash \uparrow_{\boldsymbol{x'}} \|\omega\|^{\nabla}, \boldsymbol{x'}/\boldsymbol{x} : \|\Omega\|^{\nabla}, \boldsymbol{x}\!\in\!\boldsymbol{A}^{\#}$ by $\mathsf{tpSubInd}$
$\|\Omega', \boldsymbol{x'}^{\triangledown}\!\in\!\boldsymbol{A}^{\#}[\omega]\|^{\nabla} \vdash^{\mathsf{f}} \| \uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}\|^{\nabla}$ **:** $\|\Omega, \boldsymbol{x}^{\triangledown}\!\in\!\boldsymbol{A}^{\#}\|^{\nabla}$
by above and Casting (Defs B.2.1)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u'^{\triangledown}\!\in\!\mathcal{W}) \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u^{\triangledown}\!\in\!\mathcal{W})}\ \mathsf{tpSubWorld}$

$\|\Omega'\|^{\nabla} \vdash \|\omega\|^{\nabla} : \|\Omega\|^{\nabla}$ by i.h. on $\mathcal{E}_1$
$\| \uparrow_{u'} \omega, u'/u\|^{\nabla} = \|\omega\|^{\nabla}$ by Casting (Def. B.2.1)
$\|\Omega', u'^{\triangledown}\!\in\!\mathcal{W}\|^{\nabla} = \|\Omega'\|^{\nabla}$ by Casting (Def. B.2.1)
$\|\Omega, u^{\triangledown}\!\in\!\mathcal{W}\|^{\nabla} = \|\Omega\|^{\nabla}$ by Casting (Def. B.2.1)
$\|\Omega', u'^{\triangledown}\!\in\!\mathcal{W}\|^{\nabla} \vdash \| \uparrow_{u'} \omega, u'/u\|^{\nabla} : \|\Omega, u^{\triangledown}\!\in\!\mathcal{W}\|^{\nabla}$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta \ \mathsf{wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

Subcase: $\alpha = u$ and $\delta = \tau$

$\|\Omega'\|^\nabla \vdash \|\omega\|^\nabla : \|\Omega\|^\nabla$ by i.h. on $\mathcal{E}_1$
$\| \uparrow_\alpha \omega\|^\nabla = \|\omega\|^\nabla$ by Casting (Def. B.2.1)
$\|\Omega', \alpha{\in}\delta\|^\nabla = \|\Omega'\|^\nabla$ by Casting (Def. B.2.1)
$\|\Omega', \alpha{\in}\delta\|^\nabla \vdash \| \uparrow_\alpha \omega\|^\nabla : \|\Omega\|^\nabla$ by above

Subcase: $\alpha = \boldsymbol{x}$ and $\delta = \boldsymbol{A}$

$\|\Omega'\|^\nabla \Vdash^{\mathsf{f}} \|\omega\|^\nabla \mathbf{:} \|\Omega\|^\nabla$ by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{A}\ \mathsf{wff}$ by assumption
$\|\Omega', \boldsymbol{x}{\in}\boldsymbol{A}\|^\nabla = \|\Omega'\|^\nabla, \boldsymbol{x}{\in}\boldsymbol{A}$ by Casting (Def. B.2.1)
$\|\Omega'\| \Vdash^{\mathsf{f}} \boldsymbol{A} \mathbf{:} \boldsymbol{type}$ by inversion using $\mathsf{LF\_wff}$
$\|\Omega'\| = \|\|\Omega'\|^\nabla\|$ by Lemma B.4.2
$\|\Omega'\|^\nabla \vdash \boldsymbol{A}\ \mathsf{wff}$ by $\mathsf{LF\_wff}$
$\|\Omega'\|^\nabla, \boldsymbol{x}{\in}\boldsymbol{A} \vdash \uparrow_{\boldsymbol{x}} \|\omega\|^\nabla : \|\Omega\|^\nabla$ by $\mathsf{tpSubShift}$
$\| \uparrow_{\boldsymbol{x}} \omega\|^\nabla = \uparrow_{\boldsymbol{x}} \|\omega\|^\nabla$ by Casting (Def. B.2.1)
$\|\Omega', \alpha{\in}\delta\|^\nabla \Vdash^{\mathsf{f}} \| \uparrow_\alpha \omega\|^\nabla \mathbf{:} \|\Omega\|^\nabla$ by above

Subcase: $\alpha = \boldsymbol{x}$ and $\delta = \boldsymbol{A}^\#$

$\|\Omega'\|^\nabla \Vdash^{\mathsf{f}} \|\omega\|^\nabla \mathbf{:} \|\Omega\|^\nabla$ by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{A}^\#\ \mathsf{wff}$ by assumption
$\|\Omega', \boldsymbol{x}{\in}\boldsymbol{A}^\#\|^\nabla = \|\Omega'\|^\nabla, \boldsymbol{x}{\in}\boldsymbol{A}^\#$ by Casting (Def. B.2.1)
$\|\Omega'\| \Vdash^{\mathsf{f}} \boldsymbol{A} \mathbf{:} \boldsymbol{type}$ by inversion using $\mathsf{param\_wff}$ and $\mathsf{LF\_wff}$
$\|\Omega'\| = \|\|\Omega'\|^\nabla\|$ by Lemma B.4.2
$\|\Omega'\|^\nabla \vdash \boldsymbol{A}^\#\ \mathsf{wff}$ by $\mathsf{LF\_wff}$ and $\mathsf{param\_wff}$
$\|\Omega'\|^\nabla, \boldsymbol{x}{\in}\boldsymbol{A}^\# \vdash \uparrow_{\boldsymbol{x}} \|\omega\|^\nabla : \|\Omega\|^\nabla$ by $\mathsf{tpSubShift}$
$\| \uparrow_{\boldsymbol{x}} \omega\|^\nabla = \uparrow_{\boldsymbol{x}} \|\omega\|^\nabla$ by Casting (Def. B.2.1)
$\|\Omega', \alpha{\in}\delta\|^\nabla \Vdash^{\mathsf{f}} \| \uparrow_\alpha \omega\|^\nabla \mathbf{:} \|\Omega\|^\nabla$ by above

□

B.5 Meta-Theory: Basics 2

Lemma B.5.1 (Access to Last (non-world) Element in Context)**.**

- *If* $(\Omega, \alpha{\in}\delta)\ \mathsf{ctx}$*, then* $\Omega, \alpha{\in}\delta \vdash \alpha \in \delta$*.*
- *If* $(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$*, then* $\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}$*.*

Proof. By case analysis.

Case: $(\alpha{\in}\delta)$ is $(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$

$(\Omega, \boldsymbol{x}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$	by assumption
$\Omega, \boldsymbol{x}{\in}\boldsymbol{A}^{\#} \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}$	by $\mathsf{var}^{\#}$

Case: $(\alpha{\in}\delta)$ is $(u{\in}\tau)$

$(\Omega, u{\in}\tau)\ \mathsf{ctx}$	by assumption
$(\Omega, u{\in}\tau) \leq (\Omega, u{\in}\tau)$	by leEq
$\Omega, u{\in}\tau \vdash u \in \tau$	by var

Case: $(\alpha{\in}\delta)$ is $(\boldsymbol{x}{\in}\boldsymbol{A})$

$(\Omega, \boldsymbol{x}{\in}\boldsymbol{A})\ \mathsf{ctx}$	by assumption
$(\|\Omega\|, \boldsymbol{x}{:}\boldsymbol{A})\ \mathsf{ctx}_{\mathsf{lf}}$	by Lemma B.4.1 and Casting (Def. A.2.1)
$\|\Omega\|, \boldsymbol{x}{:}\boldsymbol{A} \vdash^{\mathsf{lf}} \boldsymbol{x} : \boldsymbol{A}$	by **LF_Var**
$\Omega, \boldsymbol{x}{\in}\boldsymbol{A} \vdash \boldsymbol{x} \in \boldsymbol{A}$	by isLF

Case: $(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$

$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$	by assumption
$\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}$	by $\mathsf{var}^{\#}$

□

Lemma B.5.2 (id Substitution Does Nothing when Applied to Types)**.**
If $\Omega \vdash \delta$ wff*, then* $\delta[\mathrm{id}_\Omega] = \delta$*.*

Proof. By induction on the type utilizing the definition of Substitution Application (Definition A.2.4). We also use Lemma B.4.6 to turn it into an $\mathbf{id}_\Gamma$ when appropriate and use Lemma B.1.4.1 to claim that $\boldsymbol{A}[\mathbf{id}_\Gamma] = \boldsymbol{A}$. □

Lemma B.5.3 (id Substitution Does Nothing when Applied to e and c)**.**

- *If* $\Omega \vdash e \in \delta$*, then* $e[\mathrm{id}_\Omega] = e$*.*
- *If* $\Omega \vdash c \in \tau$*, then* $c[\mathrm{id}_\Omega] = c$*.*

Proof. By induction on the typing rules utilizing the definition of Substitution Application (Definition A.2.4). We also use Lemma B.4.6 to turn it into an $\mathbf{id}_\Gamma$ when appropriate and use Lemma B.1.4.2 to claim that $\boldsymbol{M}[\mathbf{id}_\Gamma] = \boldsymbol{M}$. □

Lemma B.5.4 (Shift Substitution Does Nothing w.r.t. Subst. Application)**.**
$\delta[(\uparrow_\alpha \omega), \omega_2] = \delta[\omega, \omega_2]$ *and* $e[(\uparrow_\alpha \omega), \omega_2] = e[\omega, \omega_2]$*.*

Proof. By inspection of Substitution Application (Def. A.2.4) and using Lemma B.1.5 to handle the cases when we cast the substitution to LF (Def. A.2.2). □

Lemma B.5.5 (id Substitution is Well-Typed)**.**
If Ω ctx*, then* $\Omega \vdash \mathrm{id}_\Omega : \Omega$.

Proof. By induction on Ω.

Case: $\Omega = \cdot$

$\cdot \vdash \cdot : \cdot$	by tpSubBase

Case: $\Omega = \Omega_s, \alpha{\in}\delta$

$(\Omega_s, \alpha{\in}\delta)$ ctx	by assumption
$\Omega_s \vdash \delta$ wff	by inversion using ctxAdd
Ω_s ctx	by inversion using ctxAdd
$\Omega_s \vdash \mathrm{id}_{\Omega_s} : \Omega_s$	by i.h. on Ω_s
$\Omega_s, \alpha{\in}\delta \vdash \uparrow_\alpha \mathrm{id}_{\Omega_s} : \Omega_s$	by tpSubShift
$\Omega_s, \alpha{\in}\delta \vdash \alpha \in \delta$	by Lemma B.5.1
$\delta[\uparrow_\alpha \mathrm{id}_{\Omega_s}] = \delta[\mathrm{id}_{\Omega_s}] = \delta$	by Lemmas B.5.4 and B.5.2
$\Omega_s, \alpha{\in}\delta \vdash (\uparrow_\alpha \mathrm{id}_{\Omega_s}, \alpha/\alpha) : \Omega_s, \alpha{\in}\delta$	by tpSubInd
$\Omega \vdash \mathrm{id}_\Omega : \Omega$	by Definition of id

Case: $\Omega = \Omega_s, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#$

$(\Omega_s, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#)$ ctx	by assumption
$\Omega_s \vdash \boldsymbol{A}^\#$ wff	by inversion using ctxAddNew
Ω_s ctx	by inversion using ctxAddNew
$\Omega_s \vdash \mathrm{id}_{\Omega_s} : \Omega_s$	by i.h. on Ω_s
$\boldsymbol{A}^\#[\mathrm{id}_{\Omega_s}] = \boldsymbol{A}^\#$	by Lemma B.5.2
$\Omega_s, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\# \vdash (\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_s}, \boldsymbol{x}/\boldsymbol{x}) : \Omega_s, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#$	by tpSubIndNew
$\Omega \vdash \mathrm{id}_\Omega : \Omega$	by Definition of id

Case: $\Omega = \Omega_s, u\overset{\triangledown}{\in}\mathcal{W}$

$(\Omega_s, u\overset{\triangledown}{\in}\mathcal{W})$ ctx	by assumption
$\mathcal{W}$ world wff	by inversion using ctxAddWorld
Ω_s ctx	by inversion using ctxAddWorld
$\Omega_s \vdash \mathrm{id}_{\Omega_s} : \Omega_s$	by i.h. on Ω_s
$\Omega_s, u\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_u \mathrm{id}_{\Omega_s}, u/u) : \Omega_s, u\overset{\triangledown}{\in}\mathcal{W}$	by tpSubWorld
$\Omega \vdash \mathrm{id}_\Omega : \Omega$	by Definition of id

□

B.6 Meta-Theory: Substitution on Well-Formedness

Lemma B.6.1 (Substitution Preserves Well-Formedness of Types)**.**
If $\Omega \vdash \delta$ wff *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash \delta[\omega]$ wff*.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash \delta$ wff.

Case: $\mathcal{E} = \dfrac{\|\Omega\| \vdash^{\mathsf{f}} \boldsymbol{A} : \boldsymbol{type}}{\Omega \vdash \boldsymbol{A}\ \mathsf{wff}}\ \mathsf{LF\_wff}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\|\Omega'\| \vdash^{\mathsf{f}} \|\omega\| : \|\Omega\|$	by Lemma B.4.7
$\|\Omega'\| \vdash^{\mathsf{f}} \boldsymbol{A}[\|\omega\|] : \boldsymbol{type}$	by Lemma B.1.12.2 and Subst. Application (Def. A.2.4)
$\Omega' \vdash \boldsymbol{A}[\|\omega\|]$ wff	by LF_wff
$\Omega' \vdash \boldsymbol{A}[\omega]$ wff	by Substitution Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash \boldsymbol{A}\ \mathsf{wff}\end{array}}{\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}}\ \mathsf{param\_wff}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash \boldsymbol{A}[\omega]$ wff	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{A}[\omega]^{\#}$ wff	by param_wff
$\Omega' \vdash \boldsymbol{A}^{\#}[\omega]$ wff	by Substitution Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{}{\Omega \vdash \mathrm{unit}\ \mathsf{wff}}\ \mathrm{unit}\mathsf{wff}$

$\Omega' \vdash \mathrm{unit}$ wff	by unitwff
$\mathrm{unit}[\omega] = \mathrm{unit}$	by Substitution Application (Def. A.2.4)
$\Omega' \vdash \mathrm{unit}[\omega]$ wff	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \\ \Omega \vdash \tau\ \mathsf{wff} & \mathcal{W}\ \mathsf{world}\end{array}}{\Omega \vdash \forall nil.\ \tau\ \mathsf{wff}}\ \forall_b\mathsf{wff}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash \tau[\omega]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}\ \mathsf{world}$	by assumption
$\Omega' \vdash \forall nil.\ \tau[\omega]\ \mathsf{wff}$	by $\forall_b\mathsf{wff}$
$\Omega' \vdash (\forall nil.\ \tau)[\omega]\ \mathsf{wff}$	by Substitution Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash \delta_1\ \mathsf{wff} & \Omega, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \forall\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}\ \forall_i\mathsf{wff}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash \delta[\omega]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega', \alpha_1{\in}\delta_1[\omega] \vdash \uparrow_{\alpha_1}\omega : \Omega$	by tpSubShift
$(\Omega', \alpha_1{\in}\delta_1[\omega])\ \mathsf{ctx}$	by ctxAdd
$\Omega', \alpha_1{\in}\delta_1[\omega] \vdash \alpha_1 \in \delta_1[\omega]$	by Lemma B.5.1
$\delta_1[\uparrow_\alpha \omega] = \delta_1[\omega]$	by Lemma B.5.4
$\Omega', \alpha_1{\in}\delta_1[\omega] \vdash (\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1) : \Omega, \alpha_1{\in}\delta$	by tpSubInd
$\Omega', \alpha_1{\in}\delta_1[\omega] \vdash (\forall\overline{\alpha{\in}\delta}.\ \tau)[\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_2$
$\Omega', \alpha_1{\in}\delta_1[\omega] \vdash \forall\overline{\alpha{\in}\delta}[\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1].\ \tau[(\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1) + \overline{\alpha{\in}\delta}]\ \mathsf{wff}$	by Substitution Application (Def. A.2.4)
$\Omega' \vdash \forall\alpha_1{\in}\delta_1[\omega]; \overline{\alpha{\in}\delta}[\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1].\ \tau[(\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1) + \overline{\alpha{\in}\delta}]\ \mathsf{wff}$	by $\forall_i\mathsf{wff}$
$\Omega' \vdash (\forall\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau)[\omega]\ \mathsf{wff}$	by Substitution Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash \delta\ \mathsf{wff} & \Omega, \alpha{\in}\delta \vdash \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}}\ \exists\mathsf{wff}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash \delta[\omega]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega', \alpha{\in}\delta[\omega] \vdash \uparrow_\alpha \omega : \Omega$	by tpSubShift
$(\Omega', \alpha{\in}\delta[\omega])\ \mathsf{ctx}$	by ctxAdd
$\Omega', \alpha{\in}\delta[\omega] \vdash \alpha \in \delta[\omega]$	by Lemma B.5.1
$\delta[\uparrow_\alpha \omega] = \delta[\omega]$	by Lemma B.5.4
$\Omega', \alpha{\in}\delta[\omega] \vdash (\uparrow_\alpha \omega, \alpha/\alpha) : \Omega, \alpha{\in}\delta$	by tpSubInd
$\Omega', \alpha{\in}\delta[\omega] \vdash \tau[\uparrow_\alpha \omega, \alpha/\alpha]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_2$
$\Omega' \vdash \exists\alpha{\in}\delta[\omega].\ \tau[\uparrow_\alpha \omega, \alpha/\alpha]\ \mathsf{wff}$	by $\exists$wff
$\Omega' \vdash (\exists\alpha{\in}\delta.\ \tau)[\omega]\ \mathsf{wff}$	by Substitution Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff} & \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}}\ \nabla\mathsf{wff}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega] \vdash (\uparrow_{\boldsymbol{x}} \omega, \boldsymbol{x}/\boldsymbol{x}) : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by tpSubIndNew
$\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega] \vdash \tau[\uparrow_{\boldsymbol{x}} \omega, \boldsymbol{x}/\boldsymbol{x}]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_2$
$\Omega' \vdash \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega].\ \tau[\uparrow_{\boldsymbol{x}} \omega, \boldsymbol{x}/\boldsymbol{x}]\ \mathsf{wff}$	by ∇wff
$\Omega \vdash (\nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)[\omega]\ \mathsf{wff}$	by Substitution Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \mathcal{W}\ \mathsf{world} \qquad \Omega \vdash \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}}\ \nabla_{\text{world}}\mathsf{wff}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash \tau[\omega]\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}\ \mathsf{world}$	by assumption
$\Omega' \vdash \nabla\mathcal{W}.\ \tau[\omega]\ \mathsf{wff}$	by ∇_{world}wff
$\Omega \vdash (\nabla\mathcal{W}.\ \tau)[\omega]\ \mathsf{wff}$	by Substitution Application (Def. A.2.4)

□

B.7 Meta-Theory: Weakening 1

Lemma B.7.1 (Context Weakening over Well-Formedness with Worlds)**.**
If $\Omega \vdash \delta$ wff *and* $\Omega \leq_{\mathcal{W}} \Omega'$ *and* Ω' ctx*, then* $\Omega' \vdash \delta$ wff*.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash \delta$ wff.

Case: $\mathcal{E} = \dfrac{\|\Omega\| \vdash^{\mathrm{lf}} \boldsymbol{A} : \boldsymbol{type}}{\Omega \vdash \boldsymbol{A} \text{ wff}}\ \mathsf{LF\_wff}$

$\Omega \leq_{\mathcal{W}} \Omega'$	by assumption
Ω' ctx	by assumption
$\|\Omega'\|$ $\mathbf{ctx_{lf}}$	by Lemma B.4.1
$\|\Omega\| \leq_{\mathrm{lf}} \|\Omega'\|$	by Lemma B.4.4
$\|\Omega'\| \vdash^{\mathrm{lf}} \boldsymbol{A} : \boldsymbol{type}$	by Lemma B.1.10
$\Omega' \vdash \boldsymbol{A}$ wff	by LF_wff

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash \boldsymbol{A} \text{ wff}\end{array}}{\Omega \vdash \boldsymbol{A}^{\#} \text{ wff}}\ \mathsf{param\_wff}$

$\Omega \leq_{\mathcal{W}} \Omega'$	by assumption
Ω' ctx	by assumption
$\Omega' \vdash \boldsymbol{A}$ wff	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{A}^{\#}$ wff	by param_wff

Case: $\mathcal{E} = \dfrac{}{\Omega \vdash \text{unit wff}}\ \text{unitwff}$

$\Omega' \vdash \text{unit wff}$	by unitwff

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash \tau \text{ wff} \qquad \mathcal{W} \text{ world}\end{array}}{\Omega \vdash \forall nil.\ \tau \text{ wff}}\ \forall_b\mathsf{wff}$

$\Omega \leq_{\mathcal{W}} \Omega'$	by assumption
Ω' ctx	by assumption
$\Omega' \vdash \tau$ wff	by i.h. on $\mathcal{E}_1$
$\mathcal{W}$ world	by assumption
$\Omega' \vdash \forall nil.\ \tau$ wff	by $\forall_b$wff

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash \delta_1 \text{ wff} & \Omega, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau \text{ wff}\end{array}}{\Omega \vdash \forall \alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau \text{ wff}}\ \forall_i\mathsf{wff}$

$\Omega \leq_{\mathcal{W}} \Omega'$ by assumption
Ω' ctx by assumption
$\Omega' \vdash \delta_1$ wff by i.h. on $\mathcal{E}_1$
$\Omega, \alpha_1{\in}\delta_1 \leq_{\mathcal{W}} \Omega', \alpha_1{\in}\delta_1$ by leWorldMiddle
$(\Omega', \alpha_1{\in}\delta_1)$ ctx by ctxAdd
$\Omega', \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau$ wff by i.h. on $\mathcal{E}_2$
$\Omega' \vdash \forall\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau$ wff by $\forall$wff

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash \delta \text{ wff} & \Omega, \alpha{\in}\delta \vdash \tau \text{ wff}\end{array}}{\Omega \vdash \exists\alpha{\in}\delta.\ \tau \text{ wff}}\ \exists\text{wff}$

$\Omega \leq_{\mathcal{W}} \Omega'$ by assumption
Ω' ctx by assumption
$\Omega' \vdash \delta$ wff by i.h. on $\mathcal{E}_1$
$\Omega, \alpha{\in}\delta \leq_{\mathcal{W}} \Omega', \alpha{\in}\delta$ by leWorldMiddle
$(\Omega', \alpha{\in}\delta)$ ctx by ctxAdd
$\Omega', \alpha{\in}\delta \vdash \tau$ wff by i.h. on $\mathcal{E}_2$
$\Omega' \vdash \exists\alpha{\in}\delta.\ \tau$ wff by $\exists$wff

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash \boldsymbol{A}^{\#} \text{ wff} & \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau \text{ wff}\end{array}}{\Omega \vdash \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau \text{ wff}}\ \nabla\text{wff}$

$\Omega \leq_{\mathcal{W}} \Omega'$ by assumption
Ω' ctx by assumption
$\Omega' \vdash \boldsymbol{A}^{\#}$ wff by i.h. on $\mathcal{E}_1$
$\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \leq_{\mathcal{W}} \Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ by leWorldMiddle
$(\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ ctx by ctxAdd
$\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau$ wff by i.h. on $\mathcal{E}_2$
$\Omega' \vdash \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$ wff by ∇wff

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{W}\ \mathsf{world} & \Omega \vdash \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}}\ \nabla_{\text{world}}\mathsf{wff}$ $\quad\begin{array}{cc}\mathcal{E}_1\end{array}$

$\Omega \leq_{\mathcal{W}} \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\Omega' \vdash \tau\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}\ \mathsf{world}$	by assumption
$\Omega' \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}$	by $\nabla_{\text{world}}\mathsf{wff}$

□

Lemma B.7.2 (Converting $\leq$ into $\leq_{\mathcal{W}}$)**.**
If $\Omega \leq \Omega'$*, then* $\Omega \leq_{\mathcal{W}} \Omega'$*.*

Proof. Straightforward by induction on $\mathcal{E} :: \Omega \leq \Omega'$. □

Lemma B.7.3 (Context Weakening over Well-Formedness without Worlds)**.**
If $\Omega \vdash \delta$ wff *and* $\Omega \leq \Omega'$ *and* Ω' ctx*, then* $\Omega' \vdash \delta$ wff*.*

Proof.

$\Omega \vdash \delta$ wff	by assumption
$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$\Omega \leq_{\cdot} \Omega'$	by Lemma B.7.2
$\Omega' \vdash \delta$ wff	by Lemma B.7.1

□

Lemma B.7.4 (Substitution Weakening on Types)**.**
If $\delta[\omega, \omega_2]$ exists and $\omega \leq \omega'$, then $\delta[\omega', \omega_2] = \delta[\omega, \omega_2]$.
We say $\delta[\omega, \omega_2]$ exists if there exists a type $\delta' = \delta[\omega, \omega_2]$. In other words, the substitution application is defined. Note that the existence of $\delta[\omega, \omega_2]$ is entailed by its use in a judgment, i.e. If $\Omega' \vdash \delta[\omega, \omega_2]$ wff, then $\delta[\omega, \omega_2]$ exists.

Proof. By induction on $\mathcal{E} :: \omega \leq \omega'$.

Case: $\mathcal{E} = \dfrac{}{\omega \leq \omega}\,\mathsf{leSubEq}$

$\delta[\omega, \omega_2] = \delta[\omega, \omega_2]$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega \leq \omega'\end{array}}{\omega \leq (\uparrow_\alpha \omega')}\,\mathsf{leSubShift}$

$\delta[\omega, \omega_2]$ exists	by assumption
$\delta[\omega', \omega_2] = \delta[\omega, \omega_2]$	by i.h. on $\mathcal{E}_1$
$\delta[(\uparrow_\alpha \omega'), \omega_2]$	
$\quad = \delta[\omega', \omega_2]$	by Lemma B.5.4
$\quad = \delta[\omega, \omega_2]$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega \leq \omega'\end{array}}{\uparrow_\alpha \omega \leq \uparrow_\alpha \omega'}\,\mathsf{leSubMiddleShift}$

$\delta[(\uparrow_\alpha \omega), \omega_2]$ exists	by assumption
$\delta[(\uparrow_\alpha \omega), \omega_2] = \delta[\omega, \omega_2]$	by Lemma B.5.4
$\delta[\omega, \omega_2]$ exists	by above
$\delta[\omega', \omega_2] = \delta[\omega, \omega_2]$	by i.h. on $\mathcal{E}_1$
$\delta[(\uparrow_\alpha \omega'), \omega_2]$	
$\quad = \delta[\omega', \omega_2]$	by Lemma B.5.4
$\quad = \delta[(\uparrow_\alpha \omega), \omega_2]$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega \leq \omega'\end{array}}{\omega \leq \omega', e/\alpha}\,\mathsf{leSubAdd}$

$\delta[\omega, \omega_2]$ exists	by assumption
$\delta[\omega', \omega_2] = \delta[\omega, \omega_2]$	by i.h. on $\mathcal{E}_1$
$\delta[\omega', \omega_2]$ exists	by above
$\delta[\omega', e/\alpha, \omega_2] = \delta[\omega', \omega_2]$	

This is a result of us tacitly renaming variable names.
As $\delta[\omega', \omega_2]$ exists, the only way adding e/α could affect substitution application is if there was already an α in (ω', ω_2), which we disallow.

$\delta[\omega', e/\alpha, \omega_2] = \delta[\omega, \omega_2]$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega \leq \omega'\end{array}}{(\omega, e/\alpha) \leq (\omega', e/\alpha)}\,\mathsf{leSubMiddle}$

$\delta[\omega, e/\alpha, \omega_2]$ exists	by assumption
$\delta[\omega', e/\alpha, \omega_2] = \delta[\omega, e/\alpha, \omega_2]$	by i.h. on $\mathcal{E}_1$

□

Lemma B.7.5 (Substitution Weakening on Expressions)**.**
If $f[\omega, \omega_2]$ exists and $\omega \leq \omega'$, then $f[\omega', \omega_2] = f[\omega, \omega_2]$.
Note that the existence of $f[\omega, \omega_2]$ is entailed by its use in a judgment, i.e. If $\Omega' \vdash f[\omega, \omega_2] \in \delta$, then $f[\omega, \omega_2]$ exists.

Proof. By induction on $\mathcal{E} :: \omega \leq \omega'$.
This proof proceeds exactly as in Lemma B.7.4. □

Lemma B.7.6 (Substitution Weakening on Cases)**.**
If $c[\omega, \omega_2]$ exists and $\omega \leq \omega'$, then $c[\omega', \omega_2] = c[\omega, \omega_2]$.
Note that the existence of $c[\omega, \omega_2]$ is entailed by its use in a judgment, i.e. If $\Omega' \vdash c[\omega, \omega_2] \in \tau$, then $c[\omega, \omega_2]$ exists.

Proof. By induction on $\mathcal{E} :: \omega \leq \omega'$.
This proof proceeds exactly as in Lemma B.7.4. □

Lemma B.7.7 (Context Weakening over $\leq_{\mathcal{W}}$ to $\leq_*$)**.**
If $\Omega \leq_{\mathcal{W}} \Omega'$, then $\Omega \leq_* \Omega'$.

Proof. By induction on $\mathcal{E} :: \Omega \leq_{\mathcal{W}} \Omega'$. In rule leWorldAddNew we get to exploit that anything is in world $*$ (includesAny). □

Lemma B.7.8 ($\leq_{\mathcal{W}}$ Relation between Casting Param)**.**
If $\Omega \leq_{\mathcal{W}} \Omega'$*, then* $\|\Omega\|^{\nabla} \leq_{\mathcal{W}} \|\Omega'\|^{\nabla}$*.*

Proof. By induction on $\mathcal{E} :: \Omega \leq_{\mathcal{W}} \Omega'$. The interesting case is leWorldAddNew, so we just show that one below.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}} \Omega' \qquad (\Omega', \boldsymbol{A}) \in \mathcal{W}\end{array}}{\Omega \leq_{\mathcal{W}} \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\ \mathsf{leWorldAddNew}$

$$
\begin{array}{lr}
\|\Omega\|^{\nabla} \leq_{\mathcal{W}} \|\Omega'\|^{\nabla} & \text{by i.h. on } \mathcal{E}_1 \\
\|\Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}\|^{\nabla} = \|\Omega'\|^{\nabla}, \boldsymbol{x} \in \boldsymbol{A}^{\#} & \text{by Def. of Casting (Def. B.2.1)} \\
\|\Omega\|^{\nabla} \leq_{\mathcal{W}} \|\Omega'\|^{\nabla}, \boldsymbol{x} \in \boldsymbol{A}^{\#} & \text{by leWorldAdd} \\
\|\Omega\|^{\nabla} \leq_{\mathcal{W}} \|\Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}\|^{\nabla} & \text{by above}
\end{array}
$$

□

Lemma B.7.9 (Context Weakening over Typing Limited to LF Types)**.**
If $\Omega \leq_* \Omega'$ *and* Ω' ctx *and* δ *is either* $\boldsymbol{A}$ *or* $\boldsymbol{A}^{\#}$ *and* $\Omega \vdash e \in \delta$*, then* $\Omega' \vdash e \in \delta$*.*

Proof. By case analysis on $\mathcal{E} :: \Omega \vdash e \in \delta$ when δ is $\boldsymbol{A}$ or $\boldsymbol{A}^{\#}$.

Case: $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad ((\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\ \mathsf{var}^{\#}$

$\Omega \leq_* \Omega'$	by assumption
Ω' ctx	by assumption
$(\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ in Ω	by assumption
$(\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ in Ω'	by inspection of Weakening Rules
$\Omega' \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}$	by $\mathsf{var}^{\#}$

Case: $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad \|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$\Omega \leq_* \Omega'$	by assumption
$\|\Omega\| \leq_{\mathsf{lf}} \|\Omega'\|$	by Lemma B.4.4
Ω' ctx	by assumption
$\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by assumption
$\|\Omega'\|$ $\mathsf{ctx}_{\mathsf{lf}}$	by Lemma B.4.1
$\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by Lemma B.1.9
$\Omega' \vdash \boldsymbol{M} \in \boldsymbol{A}$	by isLF

□

B.8 Meta-Theory: Basics 3

Lemma B.8.1 (Typing Implies Well-Formed Type)**.**

- *If* $\Omega \vdash e \in \delta$*, then* $\Omega \vdash \delta$ wff*.*
- *If* $\Omega \vdash c \in \tau$*, then* $\Omega \vdash \tau$ wff*.*

Proof. By induction on $\mathcal{E} :: \Omega \vdash e \in \delta$ and $\mathcal{E} :: \Omega \vdash c \in \tau$.

Case: $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx}}{\Omega \vdash () \in \mathrm{unit}}\ \mathsf{top}$

$\Omega \vdash \mathrm{unit}$ wff	by unitwff

Case: $\mathcal{E} = \dfrac{(\Omega_1, u{\in}\tau, \Omega_2)\ \mathsf{ctx} \quad (\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)}{\Omega_1, u{\in}\tau, \Omega_2 \vdash u \in \tau}\ \tau\mathsf{var}$

$(\Omega_1, u{\in}\tau, \Omega_2)$ ctx	by assumption
$\Omega_1 \vdash \tau$ wff	by inversion using ctxAdd
$\Omega_1 \leq \Omega_1, u{\in}\tau$	by leAdd
$\Omega_1, u{\in}\tau \vdash \tau$ wff	by Lemma B.7.3
$(\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)$	by assumption
$\Omega_1, u{\in}\tau, \Omega_2 \vdash \tau$ wff	by Lemma B.7.3

Case: $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad ((\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\ \mathsf{var}^{\#}$

Ω ctx	by assumption
$((\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}))$ in Ω	by assumption
$\Omega = \Omega_0, D, \Omega_1$ where $D = ((\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}))$	by inversion
$\Omega_0 \leq_* \Omega_0, D, \Omega_1$	by weakening rules
$\Omega_0 \vdash \boldsymbol{A}^{\#}$ wff	by inversion using ctxAdd
$\Omega \vdash \boldsymbol{A}^{\#}$ wff	by Lemma B.7.1

Case: $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad \|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by assumption
$\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$	by Lemma B.1.13
$\Omega \vdash \boldsymbol{A}\ \mathsf{wff}$	by $\mathsf{LF\_wff}$

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \Omega\ \mathsf{ctx} \quad \Omega \vdash \tau\ \mathsf{wff} \quad \text{for all } {}_{c_i \in \overline{c}}(\Omega \vdash c_i \in \tau) \quad \Omega \vdash \overline{c}\ \text{covers}\ \tau}{\Omega \vdash \text{fn}\ \overline{c} \in \tau}\ \mathsf{impI}$

$\Omega \vdash \tau\ \mathsf{wff}$	by assumption

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \Omega \vdash e \in \forall\overline{\alpha \in \delta}.\ \tau \quad \Omega \vdash \text{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}}{\Omega \vdash e\ \overline{f} \in \tau[\text{id}_\Omega, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$

$\Omega \vdash \forall\overline{\alpha \in \delta}.\ \tau\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega, \overline{\alpha \in \delta} \vdash \tau\ \mathsf{wff}$	by inversion using wff rules
$\Omega \vdash \text{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$	by assumption
$\Omega \vdash \tau[\text{id}_\Omega, f/\alpha]\ \mathsf{wff}$	by Lemma B.6.1

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash e \in \tau}{\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau}\ \mathsf{new}$

$(\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$	by inversion using ctxAdd
$\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega \vdash \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}$	by $\nabla\mathsf{wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash e \in \tau\end{array}}{\Omega \vdash \nu u{\in}\mathcal{W}.\, e \in \nabla\mathcal{W}.\, \tau}\,\mathsf{newW}$

$(\Omega, u\overset{\triangledown}{\in}\mathcal{W})$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\mathcal{W}$ world	by inversion using ctxAddWorld
$\Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash \tau$ wff	by i.h. on $\mathcal{E}_1$
$\Omega \vdash \tau$ wff	by Lemma B.3.3
$\Omega \vdash \nabla\mathcal{W}.\, \tau$ wff	by ∇_{world}wff

Case: $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\, \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\,\mathsf{pop}$

$\Omega_1 \vdash \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\, \tau$ wff	by i.h. on $\mathcal{E}_1$
$\Omega_1 \vdash \boldsymbol{A}^{\#}$ wff	by inversion using ∇wff
$\Omega_1, \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau$ wff	by inversion using ∇wff
$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ ctx	by assumption
Ω_1 ctx	by inversion using ctxAdd
$\Omega_1 \vdash \mathrm{id}_{\Omega_1} : \Omega_1$	by Lemma B.5.5
$\boldsymbol{A}^{\#}[\mathrm{id}_{\Omega_1}] = \boldsymbol{A}^{\#}$	by Lemma B.5.2
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash (\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}) : \Omega_1, \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by tpSubIndNew
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$ wff	by Lemma B.6.1
$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$	by assumption
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$ wff	by Lemma B.7.3

Case: $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ (\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}, \Omega_2)\end{array}}{\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}, \Omega_2 \vdash e\backslash u \in \tau}\ \mathsf{popW}$

$(\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}, \Omega_2)\ \mathsf{ctx}$	by assumption
$(\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W})\ \mathsf{ctx}$	by inversion
$\mathcal{W}\ \mathsf{world}$	by inversion using ctxAddWorld
$\Omega_1 \vdash \nabla\mathcal{W}_2.\ \tau\ \mathsf{wff}$	by i.h. on $\mathcal{E}_1$
$\Omega_1 \vdash \tau\ \mathsf{wff}$	by inversion using ∇_{world}wff
$\mathcal{W} \leq \mathcal{W}_2$	by assumption
$\Omega_1 \leq_{\mathcal{W}_2} \Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}$	by leWorldAddMarker
$\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W} \vdash \tau\ \mathsf{wff}$	by Lemma B.7.1
$(\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}, \Omega_2)$	by assumption
$\Omega_1, u{\overset{\nabla}{\in}}\mathcal{W}, \Omega_2 \vdash \tau\ \ \mathsf{wff}$	by Lemma B.7.1

Case: $\mathcal{E} = \dfrac{\Omega \vdash (\exists\alpha{\in}\delta.\ \tau)\ \mathsf{wff} \qquad \overset{\mathcal{E}_1}{\Omega \vdash e \in \delta} \qquad \overset{\mathcal{E}_2}{\Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]}}{\Omega \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\ \mathsf{pairI}$

$\Omega \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by assumption

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Omega, u{\in}\tau \vdash e \in \tau}}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\ \mathsf{fix}$

$(\Omega, u{\in}\tau)\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \tau\ \mathsf{wff}$	by inversion using ctxAdd

. . . . . .

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash \tau\ \mathsf{wff} \qquad \Omega, \alpha{\in}\delta \vdash c \in \tau\end{array}}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\ \mathsf{cEps}$

$\Omega \vdash \tau\ \mathsf{wff}$ by assumption

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau\end{array}}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{cNew}$

$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$ by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$ by inversion using ctxAdd
$\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}$ by i.h. on $\mathcal{E}_1$
$\Omega \vdash \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}$ by $\nabla\mathsf{wff}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}] \qquad \Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta} \qquad \Omega \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \overline{e} \mapsto f \in \forall\overline{\alpha{\in}\delta}.\ \tau}\ \mathsf{cMatch}$

$\Omega \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}$ by assumption

Case: $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$

See case for pop (above)

□

Lemma B.8.2 (Distribute Shift over LF Substitution)**.**
If $\Omega, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}}((\uparrow_\Omega \cdot), \omega_2) : \Omega'$ *and* $\Omega' = \cdot, \boldsymbol{x_1}{\in}\delta_1, \ldots, \boldsymbol{x_n}{\in}\delta_n$*,*
then $\Omega, \boldsymbol{x}{\in}\delta \vdash (\uparrow_{\boldsymbol{x}} \uparrow_\Omega \cdot), \omega_2 : \Omega'$*.*

Proof. By induction on Ω'.

Case: $\cdot$

$\Omega, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}}((\uparrow_\Omega \cdot), \omega_2) : \cdot$	by assumption
$\omega_2 = \cdot$	by inversion
$\Omega, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}} \uparrow_\Omega \cdot : \cdot$	by above

Case: $\Omega', \boldsymbol{x_{(n+1)}}{\in}\delta_{(n+1)}$

$\Omega, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}}((\uparrow_\Omega \cdot), \omega_2, e/\boldsymbol{x_{(n+1)}}) : \Omega', \boldsymbol{x_{(n+1)}}{\in}\delta_{(n+1)}$	by assumption
$\Omega' = \cdot, \boldsymbol{x_1}{\in}\delta_1, \ldots, \boldsymbol{x_n}{\in}\delta_n$	by assumption
$\Omega \vdash (\uparrow_\Omega \cdot), \omega_2, e/\boldsymbol{x_{(n+1)}} : \Omega', \boldsymbol{x_{(n+1)}}{\in}\delta_{(n+1)}$	by inversion using tpSubShift
$\Omega \vdash \delta_{(n+1)}$ wff	by inversion using tpSubShift
$\Omega \vdash ((\uparrow_\Omega \cdot), \omega_2) : \Omega'$	by inversion using tpSubInd
$\Omega \vdash e \in \delta_{(n+1)}[(\uparrow_\Omega \cdot), \omega_2]$	by inversion using tpSubInd
$\Omega, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}}((\uparrow_\Omega \cdot), \omega_2) : \Omega'$	by tpSubShift
$\Omega, \boldsymbol{x}{\in}\delta \vdash (\uparrow_{\boldsymbol{x}} \uparrow_\Omega \cdot), \omega_2 : \Omega'$	by i.h. on Ω'
Since $\boldsymbol{x_{(n+1)}}{\in}\delta_{(n+1)}$ we know that $\delta_{(n+1)} = \boldsymbol{A}$ or $\delta_{(n+1)} = \boldsymbol{A}^{\#}$	
$\Omega \leq_* \Omega, \boldsymbol{x}{\in}\delta$	by leWorldAdd
$(\Omega, \boldsymbol{x}{\in}\delta)$ ctx	by Lemma B.3.2
$\Omega, \boldsymbol{x}{\in}\delta \vdash e \in \delta_{(n+1)}[(\uparrow_\Omega \cdot), \omega_2]$	by Lemma B.7.9
$\delta_{(n+1)}[(\uparrow_\Omega \cdot), \omega_2] = \delta_{(n+1)}[(\uparrow_{\boldsymbol{x}} \uparrow_\Omega \cdot), \omega_2]$	by Subst. App. (Def. A.2.4) and Lemma B.1.5
$\Omega, \boldsymbol{x}{\in}\delta \vdash (\uparrow_{\boldsymbol{x}} \uparrow_\Omega \cdot), \omega_2, e/\boldsymbol{x_{(n+1)}} : \Omega', \boldsymbol{x_{(n+1)}}{\in}\delta_{(n+1)}$	by tpSubInd

□

Lemma B.8.3 (Substitution Property for Liveness)**.** *In this lemma we translate a substitution over the domain $\Omega, \overline{\boldsymbol{x}\in\delta}$ into one over the domain $\overline{\boldsymbol{x}\in\delta}$.*
If $\overline{\boldsymbol{x}\in\delta}$ ctx and $\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}} : \Omega, \overline{\boldsymbol{x}\in\delta}$, then $\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{e}/\overline{\boldsymbol{x}}) : \overline{\boldsymbol{x}\in\delta}$.

Proof. By induction on $\overline{\boldsymbol{x}\in\delta}$.

Case: *nil*

$\cdot$ ctx	by assumption
$\Omega \vdash \mathrm{id}_\Omega : \Omega$	by assumption
Ω ctx	by lemma B.3.2
$\|\Omega\|^\nabla$ ctx	by Lemma B.4.3
$\|\Omega\|^\nabla \vdash \uparrow_{\|\Omega\|^\nabla} \cdot : \cdot$	by multiple uses of $\mathsf{tpSubShift}$ (proof by induction on Ω)

Case: $\overline{\boldsymbol{x}\in\delta}; \boldsymbol{x_2}\in\delta_2$

$(\overline{\boldsymbol{x}\in\delta}, \boldsymbol{x_2}\in\delta_2)$ ctx	by assumption
$\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}}, e_2/\boldsymbol{x_2} : \Omega, \overline{\boldsymbol{x}\in\delta}, \boldsymbol{x_2}\in\delta_2$	by assumption
$\overline{\boldsymbol{x}\in\delta}$ ctx	by inversion using ctxAdd
$\overline{\boldsymbol{x}\in\delta} \vdash \delta_2$ wff	by inversion using ctxAdd
$\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}} : \Omega, \overline{\boldsymbol{x}\in\delta}$	by inversion using $\mathsf{tpSubInd}$
$\Omega \vdash e_2 \in \delta_2[\mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}}]$	by inversion using $\mathsf{tpSubInd}$
$\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{e}/\overline{\boldsymbol{x}}) : \overline{\boldsymbol{x}\in\delta}$	by i.h. on $\overline{\boldsymbol{x}\in\delta}$
$\|\Omega\|^\nabla \vdash \delta_2[\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{e}/\overline{\boldsymbol{x}}]$ wff	by Lemma B.6.1
$(\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{e}/\overline{\boldsymbol{x}}) \leq (\mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}})$	by weakening rules
$\delta_2[\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{e}/\overline{\boldsymbol{x}}] = \delta_2[\mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}}]$	by Lemma B.7.4
$\|\Omega\|^\nabla \vdash e_2 \in \delta_2[\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{e}/\overline{\boldsymbol{x}}]$	by case analysis on e_2 as it is an $\boldsymbol{M}$ utilizing Lemma B.4.2 o (and possibly Definition (Def. B.2.1) if it is an $\boldsymbol{x}$)
$\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{e}/\overline{\boldsymbol{x}}, e_2/\delta_2) : (\overline{\boldsymbol{x}\in\delta}; \boldsymbol{x_2}\in\delta_2)$	by $\mathsf{tpSubInd}$

□

Lemma B.8.4 (Substitution Property for Liveness Reversed)**.** *This lemma is the reverse direction of Lemma B.8.3.*
If $\overline{\boldsymbol{x}\in\delta}$ ctx *and* Ω ctx *and* $\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot, \overline{e}/\overline{\boldsymbol{x}}) : \overline{\boldsymbol{x}\in\delta}$*, then* $\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}} : \Omega, \overline{\boldsymbol{x}\in\delta}$.

Proof. By induction on $\overline{\boldsymbol{x}\in\delta}$.

Case: *nil*

Ω ctx	by assumption
$\Omega \vdash \mathrm{id}_\Omega : \Omega$	by Lemma B.5.5

Case: $\overline{\boldsymbol{x}\in\delta}; \boldsymbol{x_2}\in\delta_2$

$(\overline{\boldsymbol{x}\in\delta}, \boldsymbol{x_2}\in\delta_2)$ ctx	by assumption
Ω ctx	by assumption
$\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot, \overline{e}/\overline{\boldsymbol{x}}, e_2/\delta_2) : (\overline{\boldsymbol{x}\in\delta}; \boldsymbol{x_2}\in\delta_2)$	by assumption
$\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot, \overline{e}/\overline{\boldsymbol{x}}) : \overline{\boldsymbol{x}\in\delta}$	by inversion using $\mathsf{tpSubInd}$
$\|\Omega\|^{\nabla} \vdash e_2 \in \delta_2[\uparrow_{\|\Omega\|^{\nabla}} \cdot, \overline{e}/\overline{\boldsymbol{x}}]$	by inversion using $\mathsf{tpSubInd}$
$\overline{\boldsymbol{x}\in\delta}$ ctx	by inversion using ctxAdd
$\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}} : \Omega, \overline{\boldsymbol{x}\in\delta}$	by i.h. on $\overline{\boldsymbol{x}\in\delta}$
$(\uparrow_{\|\Omega\|^{\nabla}} \cdot, \overline{e}/\overline{\boldsymbol{x}}) \leq (\mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}})$	by weakening rules
$\delta_2[\uparrow_{\|\Omega\|^{\nabla}} \cdot, \overline{e}/\overline{\boldsymbol{x}}] = \delta_2[\mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}}]$	by Lemma B.7.4
$\Omega \vdash e_2 \in \delta_2[\mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}}]$	by case analysis on e_2 as it is an $\boldsymbol{M}$. straightforward utilizing Lemma B.4.2
$\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\boldsymbol{x}}, e_2/\boldsymbol{x_2} : \Omega, (\overline{\boldsymbol{x}\in\delta}; \boldsymbol{x_2}\in\delta_2)$	by $\mathsf{tpSubInd}$

□

Lemma B.8.5 (Generation of an Equivalent Substitution in Desired Form)**.** *If* $\|\Omega\|^{\nabla} \vdash \omega : \Omega'$ *and* $\Omega' = \cdot, \boldsymbol{x_1}{\in}\delta_1, \ldots, \boldsymbol{x_n}{\in}\delta_n$*, then there exists an* ω_2 *such that* $\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2 : \Omega'$ *and for all* δ'*,* $\delta'[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2] = \delta'[\omega]$*.*

Proof. By induction on $\mathcal{E} :: \|\Omega\|^{\nabla} \vdash \omega : \Omega'$ utilizing casting (Def. B.2.1).

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

$\cdot$ is of the desired form. by assumption

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \|\Omega\|^{\nabla} \vdash \omega : \Omega' \qquad \|\Omega\|^{\nabla} \vdash e \in \delta_{(n+1)}[\omega]\end{array}}{\|\Omega\|^{\nabla} \vdash (\omega, e/\boldsymbol{x_{(n+1)}}) : \Omega', \boldsymbol{x_{(n+1)}}{\in}\delta_{(n+1)}}\ \mathsf{tpSubInd}$

$\Omega' = \cdot, \boldsymbol{x_1}{\in}\delta_1, \ldots, \boldsymbol{x_n}{\in}\delta_n$ by assumption
There exists an ω_2 such that
$\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2 : \Omega'$
and for all δ', $\delta'[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2] = \delta'[\omega]$ by i.h. on $\mathcal{E}_1$
$\|\Omega\|^{\nabla} \vdash e \in \delta[\omega]$ by assumption
$\delta_{(n+1)}[\omega] = \delta_{(n+1)}[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2]$ by above
$\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2, e/\boldsymbol{x_{(n+1)}} : \Omega', \boldsymbol{x_{(n+1)}}{\in}\delta_{(n+1)}$ by $\mathsf{tpSubInd}$
for all δ', $\delta'[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2, e/\boldsymbol{x_{(n+1)}}] = \delta'[\omega, e/\boldsymbol{x_{(n+1)}}]$
by above and Subst. Application (Def A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \|\Omega\|^{\nabla} \vdash \omega : \Omega' \qquad \|\Omega\|^{\nabla} \vdash \delta\ \mathsf{wff}\end{array}}{\|\Omega\|^{\nabla}, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}} \omega : \Omega'}\ \mathsf{tpSubShift}$

$\Omega' = \cdot, \boldsymbol{x_1}{\in}\delta_1, \ldots, \boldsymbol{x_n}{\in}\delta_n$ by assumption
There exists an ω_2 such that
$\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2 : \Omega'$
and for all δ', $\delta'[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2] = \delta'[\omega]$ by i.h. on $\mathcal{E}_1$
$\|\Omega\|^{\nabla} \vdash \delta\ \mathsf{wff}$ by assumption
$\|\Omega\|^{\nabla}, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}}((\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2) : \Omega'$ by $\mathsf{tpSubShift}$
$\|\Omega\|^{\nabla}, \boldsymbol{x}{\in}\delta \vdash ((\uparrow_{\boldsymbol{x}} \uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2) : \Omega'$ by Lemma B.8.2
$\|\Omega\|^{\nabla}, \boldsymbol{x}{\in}\delta \vdash ((\uparrow_{\|\Omega, \boldsymbol{x}\in\delta\|^{\nabla}} \cdot), \omega_2) : \Omega'$ by Definition
for all δ', $\delta'[(\uparrow_{\|\Omega, \boldsymbol{x}\in\delta\|^{\nabla}} \cdot), \omega_2] = \delta'[\uparrow_{\boldsymbol{x}} \omega]$ by above and Lemma B.5.4

□

B.9 Meta-Theory: Equivalence of World Inclusions

Lemma B.9.1 (Alt World Inclusion Entails the Current)**.**
If Ω ctx *and* $\mathcal{W}$ world *and* $(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$*, then* $(\Omega, \boldsymbol{A}) \in \mathcal{W}$*.*

Proof. By induction on $\mathcal{E} :: (\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$.

Case: $\mathcal{E} = \dfrac{\|\Omega\|^{\nabla} \vdash \omega : \Omega'}{(\Omega, \boldsymbol{A'}[\omega]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesAltYes}$

$(\mathcal{W}', (\Omega', \boldsymbol{A'}))$ world	by assumption
Ω ctx	by assumption
Ω' ctx	by inversion using worldAdd
$\Omega' = \cdot, \boldsymbol{x_1} \in \delta_1, \ldots, \boldsymbol{x_n} \in \delta_n$	by inversion using worldAdd
There exists a substitution such that $\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot), e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n} : \Omega'$ and $\boldsymbol{A'}[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}] = \boldsymbol{A'}[\omega]$	by Lemma B.8.5
$\Omega \vdash (\text{id}_\Omega, e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}) : \Omega, \Omega'$	by Lemma B.8.4
$((\uparrow_{\|\Omega\|^{\nabla}} \cdot), e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}) \leq (\text{id}_\Omega, e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n})$	by weakening rules and properties of id and casting
$\boldsymbol{A'}[\text{id}_\Omega, e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}] = \boldsymbol{A'}[\omega]$	by above and Lemma B.7.4
$(\Omega, \boldsymbol{A'}[\omega]) \in (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by includesYes

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ (\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}'\end{array}}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesAltOther}$

$(\mathcal{W}', (\Omega', \boldsymbol{A'}))$ world	by assumption
Ω ctx	by assumption
$\mathcal{W}'$ world	by inversion using worldAdd
$(\Omega, \boldsymbol{A}) \in \mathcal{W}'$	by i.h. on $\mathcal{E}_1$
$(\Omega, \boldsymbol{A}) \in (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by includesOther

Case: $\mathcal{E} = \dfrac{}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} *}\ \mathsf{includesAltAny}$

$(\Omega, \boldsymbol{A}) \in *$	by includesAny

□

Lemma B.9.2 (Current World Inclusion Entails the Alt)**.**
If $\mathcal{W}$ world *and* $(\Omega, \boldsymbol{A}) \in \mathcal{W}$*, then* $(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$*.*

Proof. By induction on $\mathcal{E} :: (\Omega, \boldsymbol{A}) \in \mathcal{W}$.

Case: $\mathcal{E} = \dfrac{\Omega \vdash (\text{id}_\Omega, \omega) : \Omega, \Omega'}{(\Omega, \boldsymbol{A'}[\text{id}_\Omega, \omega]) \in (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesYes}$

$(\mathcal{W}', (\Omega', \boldsymbol{A'}))$ world	by assumption
Ω ctx	by Lemma B.3.2
$\|\Omega\|^\nabla$ ctx	by Lemma B.4.3
Ω' ctx and $\Omega' \vdash \boldsymbol{A}$ wff	by inversion using $\mathsf{worldAdd}$
$\Omega' = \cdot, \boldsymbol{x_1} \in \delta_1, \ldots, \boldsymbol{x_n} \in \delta_n$	by inversion using $\mathsf{worldAdd}$
$\omega = e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}$	by inversion
$\|\Omega\|^\nabla \vdash ((\uparrow_{\|\Omega\|^\nabla} \cdot), e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}) : \Omega'$	by Lemma B.8.3
$\Omega \vdash \boldsymbol{A'}[(\uparrow_{\|\Omega\|^\nabla} \cdot), e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}]$ wff	by Lemma B.6.1
$((\uparrow_{\|\Omega\|^\nabla} \cdot), e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}) \leq (\text{id}_\Omega, e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n})$	by weakening rules and properties of id and casting
$\boldsymbol{A'}[(\uparrow_{\|\Omega\|^\nabla} \cdot), e_1/\boldsymbol{x_1}, \ldots, e_n/\boldsymbol{x_n}] = \boldsymbol{A'}[\text{id}_\Omega, \omega]$	by above and Lemma B.7.4
$(\Omega, \boldsymbol{A'}[\text{id}_\Omega, \omega]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by $\mathsf{includesAltYes}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ (\Omega, \boldsymbol{A}) \in \mathcal{W}'\end{array}}{(\Omega, \boldsymbol{A}) \in (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesOther}$

$(\mathcal{W}', (\Omega', \boldsymbol{A'}))$ world	by assumption
$\mathcal{W}'$ world	by inversion using $\mathsf{worldAdd}$
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}'$	by i.h. on $\mathcal{E}_1$
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by $\mathsf{includesAltOther}$

Case: $\mathcal{E} = \dfrac{}{(\Omega, \boldsymbol{A}) \in *}\ \mathsf{includesAny}$

$(\Omega, \boldsymbol{A}) \in_{\text{alt}} *$	by $\mathsf{includesAltAny}$

□

B.10 Meta-Theory: Weakening 2

Lemma B.10.1 (Context Weakening for World Inclusion)**.**

1. *If* Ω_2 ctx *and* $\Omega \leq_* \Omega_2$ *and* $\Omega \vdash (\uparrow_\Omega \cdot), \omega : \Omega'$ *and all contexts* Ω, Ω_2, Ω' *only have decs. of the form* $\boldsymbol{x}{\in}\delta$, *then* $\Omega_2 \vdash (\uparrow_{\Omega_2} \cdot), \omega : \Omega'$.

2. *If* Ω_2 ctx *and* $\Omega \leq_* \Omega_2$ *and* $\mathcal{W}$ world *and* $(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$, *then* $(\Omega_2, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$.

3. *If* Ω_2 ctx *and* $\Omega \leq_* \Omega_2$ *and* $\mathcal{W}$ world *and* $(\Omega, \boldsymbol{A}) \in \mathcal{W}$, *then* $(\Omega_2, \boldsymbol{A}) \in \mathcal{W}$.

Proof.

1 By induction on ω

Case: $\cdot$

Ω_2 ctx	by assumption
$\Omega \leq_* \Omega_2$	by assumption
$\Omega \vdash (\uparrow_\Omega \cdot) : \cdot$	by assumption
$\Omega_2 \vdash (\uparrow_{\Omega_2} \cdot) : \cdot$	proof by straightforward induction on Ω_2 utilizing tpSubBase, tpSubShift and inversion using ctxAdd

Case: $\omega, e/\boldsymbol{x}$

Ω_2 ctx	by assumption
$\Omega \leq_* \Omega_2$	by assumption
$\Omega \vdash (\uparrow_\Omega \cdot), \omega, e/\boldsymbol{x} : \Omega', \boldsymbol{x}{\in}\delta$	by assumption
All contexts Ω, Ω_2, Ω' only have decs. of the form $\boldsymbol{x'}{\in}\delta'$	by assumption
$\Omega \vdash (\uparrow_\Omega \cdot), \omega : \Omega'$	by inversion using tpSubInd
$\Omega \vdash e \in \delta[(\uparrow_\Omega \cdot), \omega]$	by inversion using tpSubInd
$\Omega_2 \vdash (\uparrow_{\Omega_2} \cdot), \omega : \Omega'$	by i.h. on ω
Since $\boldsymbol{x}{\in}\delta$, we know $\delta = \boldsymbol{A}$ or $\delta = \boldsymbol{A}^\#$	
$\Omega_2 \vdash e \in \delta[(\uparrow_\Omega \cdot), \omega]$	by Lemma B.7.9
$\delta[(\uparrow_\Omega \cdot), \omega] = \delta[(\uparrow_{\Omega_2} \cdot), \omega]$	by Lemma B.1.5
$\Omega_2 \vdash (\uparrow_{\Omega_2} \cdot), \omega, e/\boldsymbol{x} : \Omega', \boldsymbol{x}{\in}\delta$	by tpSubInd

2 By induction on $\mathcal{E} :: (\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$

Case: $\mathcal{E} = \dfrac{\|\Omega\|^{\nabla} \vdash \omega : \Omega'}{(\Omega, \boldsymbol{A'}[\omega]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesAltYes}$

$\Omega \leq_* \Omega_2$	by assumption
Ω_2 ctx	by assumption
$(\mathcal{W}', (\Omega', \boldsymbol{A'}))$ world	by assumption
Ω' ctx	by inversion using worldAdd
Ω' only has decs. of $\boldsymbol{x}{\in}\delta$	by inversion using worldAdd
$\|\Omega\|^{\nabla} \vdash \omega : \Omega'$	by assumption (premise)
$\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot), \omega_2 : \Omega'$ and $\boldsymbol{A'}[\omega] = \boldsymbol{A'}[\uparrow_{\|\Omega\|^{\nabla}} \cdot, \omega_2]$	by Lemma B.8.5
$\|\Omega_2\|^{\nabla}$ ctx	by Lemma B.4.3
$\|\Omega\|^{\nabla} \leq_* \|\Omega_2\|^{\nabla}$	by Lemma B.7.8
$\|\Omega\|^{\nabla}$ and $\|\Omega_2\|^{\nabla}$ only contain decs. of $\boldsymbol{x}{\in}\delta$	by Casting (Def. B.2.1)
$\|\Omega_2\|^{\nabla} \vdash (\uparrow_{\|\Omega_2\|^{\nabla}} \cdot), \omega_2 : \Omega'$	by Part 1
$(\Omega_2, \boldsymbol{A'}[(\uparrow_{\|\Omega_2\|^{\nabla}} \cdot), \omega_2]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by includesAltYes
$\boldsymbol{A'}[\omega] = \boldsymbol{A'}[\uparrow_{\|\Omega\|^{\nabla}} \cdot, \omega_2] = \boldsymbol{A'}[\uparrow_{\|\Omega_2\|^{\nabla}} \cdot, \omega_2]$	by Lemma B.1.5 and above
$(\Omega_2, \boldsymbol{A'}[\omega]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ (\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}'\end{array}}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}\ \mathsf{includesAltOther}$

$\Omega \leq_* \Omega_2$	by assumption
Ω_2 ctx	by assumption
$(\mathcal{W}', (\Omega', \boldsymbol{A'}))$ world	by assumption
$\mathcal{W}'$ world	by inversion using worldAdd
$(\Omega_2, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}'$	by i.h. on $\mathcal{E}_1$
$(\Omega_2, \boldsymbol{A}) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by includesAltOther

Case: $\mathcal{E} = \dfrac{}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} *}\ \mathsf{includesAltAny}$

$(\Omega_2, \boldsymbol{A}) \in_{\text{alt}} *$	by includesAltAny

3 Proof direct (below)

Ω_2 ctx	by assumption
$\Omega \leq_* \Omega_2$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega, \boldsymbol{A}) \in \mathcal{W}$	by assumption
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$	by Lemma B.9.2
$(\Omega_2, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$	by this Lemma, Part 2
$(\Omega_2, \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.9.1

□

Lemma B.10.2 (Transitivity of $\leq_{\mathcal{W}}$)**.**
If $\Omega_1 \leq_{\mathcal{W}} \Omega_2$ *and* $\Omega_2 \leq_{\mathcal{W}} \Omega_3$*, and* Ω_3 ctx *and* $\mathcal{W}$ world*, then* $\Omega_1 \leq_{\mathcal{W}} \Omega_3$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \Omega_1 \leq_{\mathcal{W}} \Omega_2$ and $\mathcal{F} :: \Omega_2 \leq_{\mathcal{W}} \Omega_3$.

Case: $\mathcal{E} = \Omega_1 \leq_{\mathcal{W}} \Omega_2$ and $\mathcal{F} = \dfrac{}{\Omega_2 \leq_{\mathcal{W}} \Omega_2}\,\mathsf{leWorldEq}$

$\Omega_1 \leq_{\mathcal{W}} \Omega_2$	by $\mathcal{E}$

Case: $\mathcal{E} = \Omega_1 \leq_{\mathcal{W}} \Omega_2$ and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3\end{array}}{\Omega_2 \leq_{\mathcal{W}} \Omega_3, \alpha_1 \in \delta_1}\,\mathsf{leWorldAdd}$

$(\Omega_3, \alpha_1 \in \delta_1)$ ctx	by assumption
$\mathcal{W}$ world	by assumption
Ω_3 ctx	by inversion using ctxAdd
$\Omega_1 \leq_{\mathcal{W}} \Omega_3$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\Omega_1 \leq_{\mathcal{W}} \Omega_3, \alpha_1 \in \delta_1$	by $\mathsf{leWorldAdd}$

Case: $\mathcal{E} = \Omega_1 \leq_{\mathcal{W}} \Omega_2$ and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3 \qquad (\Omega_3, \boldsymbol{A}) \in \mathcal{W}\end{array}}{\Omega_2 \leq_{\mathcal{W}} \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\,\mathsf{leWorldAddNew}$

$(\Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ ctx	by assumption
$\mathcal{W}$ world	by assumption
Ω_3 ctx	by inversion using $\mathsf{ctxAddNew}$
$\Omega_1 \leq_{\mathcal{W}} \Omega_3$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$(\Omega_3, \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\Omega_1 \leq_{\mathcal{W}} \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by $\mathsf{leWorldAddNew}$

Case: $\mathcal{E} = \Omega_1 \leq_{\mathcal{W}} \Omega_2$ and $\mathcal{F} = \dfrac{\begin{array}{cc}\mathcal{F}_1 & \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3 & \mathcal{W}' \leq \mathcal{W}\end{array}}{\Omega_2 \leq_{\mathcal{W}} \Omega_3, u{\overset{\triangledown}{\in}}\mathcal{W}'}\,\mathsf{leWorldAddMarker}$

$(\Omega_3, u{\overset{\triangledown}{\in}}\mathcal{W}')$ ctx	by assumption
$\mathcal{W}$ world	by assumption
Ω_3 ctx	by inversion using ctxAddWorld
$\Omega_1 \leq_{\mathcal{W}} \Omega_3$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\mathcal{W}' \leq \mathcal{W}$	by assumption
$\Omega_1 \leq_{\mathcal{W}} \Omega_3, u{\overset{\triangledown}{\in}}\mathcal{W}'$	by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{}{\Omega_2, D \leq_{\mathcal{W}} \Omega_2, D}\,\mathsf{leWorldEq}$ and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3\end{array}}{\Omega_2, D \leq_{\mathcal{W}} \Omega_3, D}\,\mathsf{leWorldMiddle}$

$\Omega_2, D \leq_{\mathcal{W}} \Omega_3, D$	by $\mathcal{F}$

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \\ \Omega_1 \leq_{\mathcal{W}} \Omega_2 & D = \alpha_1{\in}\delta_1\end{array}}{\Omega_1 \leq_{\mathcal{W}} \Omega_2, D}\,\mathsf{leWorldAdd}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3\end{array}}{\Omega_2, D \leq_{\mathcal{W}} \Omega_3, D}\,\mathsf{leWorldMiddle}$

(Ω_3, D) ctx	by assumption
$\mathcal{W}$ world	by assumption
Ω_3 ctx	by inversion using ctxAdd
$\Omega_1 \leq_{\mathcal{W}} \Omega_3$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\Omega_1 \leq_{\mathcal{W}} \Omega_3, D$	by leWorldAdd

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \leq_{\mathcal{W}} \Omega_2 \qquad (\Omega_2, \boldsymbol{A}) \in \mathcal{W}\end{array}}{\Omega_1 \leq_{\mathcal{W}} \Omega_2, \boldsymbol{x} \mathrel{\overset{\triangledown}{\in}} \boldsymbol{A}^{\#}}\,\mathsf{leWorldAddNew}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3\end{array}}{\Omega_2, \boldsymbol{x} \mathrel{\overset{\triangledown}{\in}} \boldsymbol{A}^{\#} \leq_{\mathcal{W}} \Omega_3, \boldsymbol{x} \mathrel{\overset{\triangledown}{\in}} \boldsymbol{A}^{\#}}\,\mathsf{leWorldMiddle}$

$(\Omega_3, \boldsymbol{x} \mathrel{\overset{\triangledown}{\in}} \boldsymbol{A}^{\#})$ ctx	by assumption
$\mathcal{W}$ world	by assumption
Ω_3 ctx	by inversion using $\mathsf{ctxAddNew}$
$\Omega_1 \leq_{\mathcal{W}} \Omega_3$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$(\Omega_2, \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\Omega_2 \leq_* \Omega_3$	by Lemma B.7.7 on $\mathcal{F}_1$
$(\Omega_3, \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.10.1
$\Omega_1 \leq_{\mathcal{W}} \Omega_3, \boldsymbol{x} \mathrel{\overset{\triangledown}{\in}} \boldsymbol{A}^{\#}$	by $\mathsf{leWorldAddNew}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \leq_{\mathcal{W}} \Omega_2 \end{array} \quad \mathcal{W}' \leq \mathcal{W}}{\Omega_1 \leq_{\mathcal{W}} \Omega_2, u{:}\overset{\triangledown}{\in}\mathcal{W}'}\,\mathsf{leWorldAddMarker}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3 \end{array}}{\Omega_2, u{:}\overset{\triangledown}{\in}\mathcal{W}' \leq_{\mathcal{W}} \Omega_3, u{:}\overset{\triangledown}{\in}\mathcal{W}'}\,\mathsf{leWorldMiddle}$

$(\Omega_3, u{:}\overset{\triangledown}{\in}\mathcal{W}')$ ctx	by assumption
$\mathcal{W}$ world	by assumption
Ω_3 ctx	by inversion using ctxAddWorld
$\Omega_1 \leq_{\mathcal{W}} \Omega_3$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\mathcal{W}' \leq \mathcal{W}$	by assumption
$\Omega_1 \leq_{\mathcal{W}} \Omega_3, u{:}\overset{\triangledown}{\in}\mathcal{W}'$	by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \leq_{\mathcal{W}} \Omega_2 \end{array}}{\Omega_1, D \leq_{\mathcal{W}} \Omega_2, D}\,\mathsf{leWorldMiddle}$ and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \leq_{\mathcal{W}} \Omega_3 \end{array}}{\Omega_2, D \leq_{\mathcal{W}} \Omega_3, D}\,\mathsf{leWorldMiddle}$

(Ω_3, D) ctx	by assumption
$\mathcal{W}$ world	by assumption
Ω_3 ctx	by inversion using ctxAdd or ctxAddNew
$\Omega_1 \leq_{\mathcal{W}} \Omega_3$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\Omega_1, D \leq_{\mathcal{W}} \Omega_3, D$	by leWorldMiddle

□

B.11 Meta-Theory: Casting 2

Lemma B.11.1 (LF Substitution Application on a Substitution Composed with $\cdot$)**.**

1. $\boldsymbol{A}[(\cdot \circ^{\mathbf{lf}} \gamma), \gamma_2] = \boldsymbol{A}[\gamma, \gamma_2]$.
2. $\boldsymbol{M}[(\cdot \circ^{\mathbf{lf}} \gamma), \gamma_2] = \boldsymbol{M}[\gamma, \gamma_2]$.

The proof is the same for $\boldsymbol{A}$ and $\boldsymbol{M}$, so we just show it below for $\boldsymbol{A}$.

Proof. By induction on γ.

Case: $\gamma = \uparrow_{x'} \gamma'$

$$
\begin{aligned}
&\boldsymbol{A}[(\cdot \circ^{\mathbf{lf}} \uparrow_{x'} \gamma'), \gamma_2] \\
&\quad = \boldsymbol{A}[(\uparrow_{x'} (\cdot \circ^{\mathbf{lf}} \gamma')), \gamma_2] && \text{by Composition (Def. A.2.3)} \\
&\quad = \boldsymbol{A}[(\cdot \circ^{\mathbf{lf}} \gamma'), \gamma_2] && \text{by Lemma B.1.5} \\
&\quad = \boldsymbol{A}[\gamma', \gamma_2] && \text{by i.h. on } \gamma' \\
&\quad = \boldsymbol{A}[\uparrow_{x'} \gamma', \gamma_2] && \text{by Lemma B.1.5}
\end{aligned}
$$

Case: $\gamma = \gamma', \boldsymbol{M'}/\boldsymbol{x'}$ or $\gamma = \cdot$

$$
\boldsymbol{A}[(\cdot \circ^{\mathbf{lf}} \gamma), \gamma_2] = \boldsymbol{A}[\gamma, \gamma_2] \qquad \text{by Composition (Def. A.2.3)}
$$

□

Lemma B.11.2 (Equivalence of LF Substitution Application)**.**

1. $\boldsymbol{A}[((\gamma, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathbf{lf}} \gamma_2), \gamma_0] = \boldsymbol{A}[(\gamma \circ^{\mathbf{lf}} \gamma_2), \boldsymbol{M}[\gamma_2]/\boldsymbol{x}, \gamma_0]$.
2. $\boldsymbol{M_0}[((\gamma, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathbf{lf}} \gamma_2), \gamma_0] = \boldsymbol{M_0}[(\gamma \circ^{\mathbf{lf}} \gamma_2), \boldsymbol{M}[\gamma_2]/\boldsymbol{x}, \gamma_0]$.

The proof is the same for $\boldsymbol{A}$ and $\boldsymbol{M_0}$, so we just show it below for $\boldsymbol{A}$.

Proof. By induction on γ_2.

Case: $\gamma_2 = \uparrow_{x'} \gamma_2'$

$$
\begin{aligned}
&\boldsymbol{A}[((\gamma, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathbf{lf}} (\uparrow_{x'} \gamma_2')), \gamma_0] \\
&\quad = \boldsymbol{A}[(\uparrow_{x'} ((\gamma, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathbf{lf}} \gamma_2')), \gamma_0] && \text{by Composition (Def. A.2.3)} \\
&\quad = \boldsymbol{A}[((\gamma, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathbf{lf}} \gamma_2'), \gamma_0] && \text{by Lemma B.1.5} \\
&\quad = \boldsymbol{A}[(\gamma \circ^{\mathbf{lf}} \gamma_2'), \boldsymbol{M}[\gamma_2']/\boldsymbol{x}, \gamma_0] && \text{by i.h. on } \gamma_2' \\
&\quad = \boldsymbol{A}[(\gamma \circ^{\mathbf{lf}} \gamma_2'), \boldsymbol{M}[\uparrow_{x'} \gamma_2']/\boldsymbol{x}, \gamma_0] && \text{by Lemma B.1.5} \\
&\quad = \boldsymbol{A}[\uparrow_{x'} (\gamma \circ^{\mathbf{lf}} \gamma_2'), \boldsymbol{M}[\uparrow_{x'} \gamma_2']/\boldsymbol{x}, \gamma_0] && \text{by Lemma B.1.5} \\
&\quad = \boldsymbol{A}[(\gamma \circ^{\mathbf{lf}} \uparrow_{x'} \gamma_2'), \boldsymbol{M}[\uparrow_{x'} \gamma_2']/\boldsymbol{x}, \gamma_0] && \text{by Composition (Def. A.2.3)}
\end{aligned}
$$

Case: $\gamma_2 = \gamma_2', \boldsymbol{M'}/\boldsymbol{x'}$ or $\gamma_2 = \cdot$

$$
\boldsymbol{A}[((\gamma, \boldsymbol{M}/\boldsymbol{x}) \circ^{\mathbf{lf}} \gamma_2), \gamma_0] = \boldsymbol{A}[(\gamma \circ^{\mathbf{lf}} \gamma_2), \boldsymbol{M}[\gamma_2]/\boldsymbol{x}, \gamma_0] \quad \text{by Composition (Def. A.2.3)}
$$

□

Lemma B.11.3 (Casting Composition to LF for Substitution Application)**.**

1. *If* $\Omega' \vdash \omega : \Omega$ *and* $\Omega_2 \vdash \omega_2 : \Omega'$*, then* $\boldsymbol{A}[\|\omega \circ \omega_2\|, \boldsymbol{\gamma}] = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}]$.
2. *If* $\Omega' \vdash \omega : \Omega$ *and* $\Omega_2 \vdash \omega_2 : \Omega'$*, then* $\boldsymbol{M}[\|\omega \circ \omega_2\|, \boldsymbol{\gamma}] = \boldsymbol{M}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}]$.

The proof is the same for $\boldsymbol{A}$ *and* $\boldsymbol{M}$*, so we just show it below for* $\boldsymbol{A}$*.*

Proof. By induction lexicographically
on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$ and $\mathcal{F} :: \Omega_2 \vdash \omega_2 : \Omega'$.

Case: $\mathcal{E} = \Omega' \vdash \omega : \Omega$

$$\text{and } \mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash \delta\ \mathsf{wff}\end{array}}{\Omega_2, \alpha{\in}\delta \vdash \uparrow_\alpha \omega_2 : \Omega'}\ \mathsf{tpSubShift}$$

Subcase: $\alpha = u$

$$\begin{array}{lr}
\boldsymbol{A}[\|\omega{\circ}\uparrow_u \omega_2\|, \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\|\uparrow_u (\omega \circ \omega_2)\|, \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[\|\omega \circ \omega_2\|, \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by i.h. on } \mathcal{E} \text{ and } \mathcal{F}_1 \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\uparrow_u \omega_2\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}$$

Subcase: $\alpha = \boldsymbol{x}$

$$\begin{array}{lr}
\boldsymbol{A}[\|\omega{\circ}\uparrow_{\boldsymbol{x}} \omega_2\|, \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\|\uparrow_{\boldsymbol{x}} (\omega \circ \omega_2)\|, \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[\uparrow_{\boldsymbol{x}} \|\omega \circ \omega_2\|, \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)} \\
\quad = \boldsymbol{A}[\|\omega \circ \omega_2\|, \boldsymbol{\gamma}] & \text{by Lemma B.1.5} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by i.h. on } \mathcal{E} \text{ and } \mathcal{F}_1 \\
\quad = \boldsymbol{A}[\uparrow_{\boldsymbol{x}} (\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by Lemma B.1.5} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \uparrow_{\boldsymbol{x}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\uparrow_{\boldsymbol{x}} \omega_2\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}$$

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \cdot$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

$$\begin{array}{lr}
\boldsymbol{A}[\|\cdot \circ \omega_2\|, \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\|\omega_2\|, \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[(\cdot \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by Lemma B.11.1} \\
\quad = \boldsymbol{A}[(\|\cdot\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}$$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : \Omega, \alpha{\in}\delta}\ \mathsf{tpSubInd}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega$, where $\omega_2 \neq\!\uparrow_{\alpha''} \omega_2''$

Subcase: $\alpha = u$

$$
\begin{array}{ll}
\boldsymbol{A}[\|(\omega, e/u) \circ \omega_2\|, \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\|(\omega \circ \omega_2), e[\omega_2]/u\|, \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[\|\omega \circ \omega_2\|, \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by i.h. on } \mathcal{E}_1 \text{ and } \mathcal{F} \\
\quad = \boldsymbol{A}[(\|\omega, e/u\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}
$$

Subcase: $\alpha = \boldsymbol{x}$

$$
\begin{array}{ll}
\Omega' \vdash e \in \delta[\omega] & \text{by assumption} \\
\delta = \boldsymbol{B} \text{ or } \boldsymbol{B}^{\#} & \text{by inversion since } \alpha = \boldsymbol{x} \\
e = \boldsymbol{M} \text{ or } \boldsymbol{x'} & \text{by inversion using either } \mathsf{isLF} \text{ or } \mathsf{var}^{\#} \\
e[\omega_2] = e[\|\omega_2\|] & \text{by Subst. Application (Def. A.2.4)} \\
\boldsymbol{A}[\|(\omega, e/\boldsymbol{x}) \circ \omega_2\|, \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\|((\omega \circ \omega_2), e[\|\omega_2\|]/\boldsymbol{x})\|, \boldsymbol{\gamma}] & \\
\qquad & \text{by Composition (Def. A.2.3) and above} \\
\quad = \boldsymbol{A}[\|\omega \circ \omega_2\|, e[\|\omega_2\|]/\boldsymbol{x}, \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), e[\|\omega_2\|]/\boldsymbol{x}, \boldsymbol{\gamma}] & \text{by i.h. on } \mathcal{E}_1 \text{ and } \mathcal{F} \\
\quad = \boldsymbol{A}[((\|\omega\|, e/\boldsymbol{x}) \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by Lemma B.11.2} \\
\quad = \boldsymbol{A}[(\|\omega, e/\boldsymbol{x}\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}
$$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash \boldsymbol{A}^{\#}[\omega][\omega_2]\ \mathsf{wff}\end{array}}{(\Omega_2, \boldsymbol{x''} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega][\omega_2]) \vdash (\uparrow_{\boldsymbol{x''}} \omega_2, \boldsymbol{x''}/\boldsymbol{x'}) : (\Omega', \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega])}\ \mathsf{tpSubIndNew}$

$$
\begin{aligned}
&\boldsymbol{A}[\|(\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x''}} \omega_2, \boldsymbol{x''}/\boldsymbol{x'})\|, \boldsymbol{\gamma}] \\
&\quad = \boldsymbol{A}[\|(\uparrow_{\boldsymbol{x'}} \omega) \circ (\uparrow_{\boldsymbol{x''}} \omega_2, \boldsymbol{x''}/\boldsymbol{x'}), \boldsymbol{x'}[(\uparrow_{\boldsymbol{x''}} \omega_2, \boldsymbol{x''}/\boldsymbol{x'})]/\boldsymbol{x}\|, \boldsymbol{\gamma}] \\
&\qquad\qquad \text{by Composition (Def. A.2.3)} \\
&\quad = \boldsymbol{A}[\|(\uparrow_{\boldsymbol{x'}} \omega) \circ (\uparrow_{\boldsymbol{x''}} \omega_2, \boldsymbol{x''}/\boldsymbol{x'}), \boldsymbol{x''}/\boldsymbol{x}\|, \boldsymbol{\gamma}] \quad \text{by Subst. App. (Def. A.2.4)} \\
&\quad = \boldsymbol{A}[\|(\uparrow_{\boldsymbol{x'}} \omega) \circ (\uparrow_{\boldsymbol{x''}} \omega_2, \boldsymbol{x''}/\boldsymbol{x'})\|, \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by Casting (Def. A.2.2)} \\
&\quad = \boldsymbol{A}[\|\omega \circ \uparrow_{\boldsymbol{x''}} \omega_2\|, \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by Composition (Def. A.2.3)} \\
&\quad = \boldsymbol{A}[\|\uparrow_{\boldsymbol{x''}} (\omega \circ \omega_2)\|, \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by Composition (Def. A.2.3)} \\
&\quad = \boldsymbol{A}[\uparrow_{\boldsymbol{x''}} \|\omega \circ \omega_2\|, \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by Casting (Def. A.2.2)} \\
&\quad = \boldsymbol{A}[\|\omega \circ \omega_2\|, \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by Lemma B.1.5} \\
&\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by i.h. on } \mathcal{E}_1 \text{ and } \mathcal{F}_1 \\
&\quad = \boldsymbol{A}[\uparrow_{\boldsymbol{x''}} (\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by Lemma B.1.5} \\
&\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} (\uparrow_{\boldsymbol{x''}} \|\omega_2\|)), \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \quad \text{by Composition (Def. A.2.3)} \\
&\quad = \boldsymbol{A}[((\uparrow_{\boldsymbol{x'}} \|\omega\|) \circ^{\mathbf{lf}} (\uparrow_{\boldsymbol{x''}} \|\omega_2\|, \boldsymbol{x''}/\boldsymbol{x'})), \boldsymbol{x''}/\boldsymbol{x}, \boldsymbol{\gamma}] \\
&\qquad\qquad \text{by Composition (Def. A.2.3)} \\
&\quad = \boldsymbol{A}[((\uparrow_{\boldsymbol{x'}} \|\omega\|, \boldsymbol{x'}/\boldsymbol{x}) \circ^{\mathbf{lf}} (\uparrow_{\boldsymbol{x''}} \|\omega_2\|, \boldsymbol{x''}/\boldsymbol{x'})), \boldsymbol{\gamma}] \\
&\qquad\qquad \text{by Composition (Def. A.2.3) and Subst. Application (Def. A.2.4)} \\
&\quad = \boldsymbol{A}[(\| \uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}\| \circ^{\mathbf{lf}} \|(\uparrow_{\boldsymbol{x''}} \omega_2, \boldsymbol{x''}/\boldsymbol{x'})\|), \boldsymbol{\gamma}] \quad \text{by Casting (Def. A.2.2)}
\end{aligned}
$$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega_2, u''\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'} \omega_2, u''/u') : (\Omega', u'\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

$$
\begin{array}{lr}
\boldsymbol{A}[\|(\uparrow_{u'} \omega, u'/u) \circ (\uparrow_{u''} \omega_2, u''/u')\|, \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\|(\uparrow_{u'} \omega) \circ (\uparrow_{u''} \omega_2, u''/u'), u'[(\uparrow_{u''} \omega_2, u''/u')]/u\|, \boldsymbol{\gamma}] & \\
& \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[\|(\uparrow_{u'} \omega) \circ (\uparrow_{u''} \omega_2, u''/u'), u''/u\|, \boldsymbol{\gamma}] & \text{by Subst. App. (Def. A.2.4)} \\
\quad = \boldsymbol{A}[\|(\uparrow_{u'} \omega) \circ (\uparrow_{u''} \omega_2, u''/u')\|, \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)} \\
\quad = \boldsymbol{A}[\|\omega \circ \uparrow_{u''} \omega_2\|, \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[\|\uparrow_{u''} (\omega \circ \omega_2)\|, \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[\|\omega \circ \omega_2\|, \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by i.h. on } \mathcal{E}_1 \text{ and } \mathcal{F}_1 \\
\quad = \boldsymbol{A}[(\|\uparrow_{u'} \omega, u'/u\| \circ^{\mathbf{lf}} \|(\uparrow_{u''} \omega_2, u''/u')\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}
$$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta\ \mathsf{wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash e \in \delta[\omega]\end{array}}{\Omega_2 \vdash (\omega_2, e/\alpha) : (\Omega', \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

Subcase: $\alpha = u$

$$
\begin{array}{rr}
\boldsymbol{A}[\!\| \uparrow_u \omega \circ (\omega_2, e/u) \|\!], \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\!\|\omega \circ \omega_2\|\!], \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by i.h. on } \mathcal{E}_1 \text{ and } \mathcal{F}_1 \\
\quad = \boldsymbol{A}[(\!\| \uparrow_u \omega\!\| \circ^{\mathbf{lf}} \|\omega_2, e/u\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}
$$

Subcase: $\alpha = \boldsymbol{x}$

$$
\begin{array}{rr}
\boldsymbol{A}[\!\| \uparrow_{\boldsymbol{x}} \omega \circ (\omega_2, e/\boldsymbol{x}) \|\!], \boldsymbol{\gamma}] & \\
\quad = \boldsymbol{A}[\!\|\omega \circ \omega_2\|\!], \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[(\|\omega\| \circ^{\mathbf{lf}} \|\omega_2\|), \boldsymbol{\gamma}] & \text{by i.h. on } \mathcal{E}_1 \text{ and } \mathcal{F}_1 \\
\quad = \boldsymbol{A}[((\uparrow_{\boldsymbol{x}} \|\omega\|) \circ^{\mathbf{lf}} (\|\omega_2\|, \boldsymbol{e}/\boldsymbol{x})), \boldsymbol{\gamma}] & \text{by Composition (Def. A.2.3)} \\
\quad = \boldsymbol{A}[(\!\| \uparrow_{\boldsymbol{x}} \omega\!\| \circ^{\mathbf{lf}} \|\omega_2, e/\boldsymbol{x}\|), \boldsymbol{\gamma}] & \text{by Casting (Def. A.2.2)}
\end{array}
$$

□

B.12 Meta-Theory: Substitution Properties 1

Lemma B.12.1 (Equality of Substitution Composition Applied to Types)**.** *If* $\Omega_1 \vdash \omega_1 : \Omega_0$ *and* $\Omega_2 \vdash \omega_2 : \Omega_1$ *and* $\Omega_0 \vdash \delta$ wff *and* $(\omega_1 \circ \omega_2)$ *exists, then* $\delta[\omega_1][\omega_2] = \delta(\omega_1 \circ \omega_2)$.

Proof. By induction on $\mathcal{E} :: \Omega_0 \vdash \delta$ wff.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_0 \vdash \boldsymbol{A} \text{ wff}\end{array}}{\Omega_0 \vdash \boldsymbol{A}^{\#} \text{ wff}}\,\text{param\_wff}$

$\Omega_1 \vdash \omega_1 : \Omega_0$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$(\omega_1 \circ \omega_2)$ exists	by assumption
$\boldsymbol{A}^{\#}[\omega_1][\omega_2]$	
$= \boldsymbol{A}[\omega_1][\omega_2]^{\#}$	by Subst. App. (Def. A.2.4)
$= \boldsymbol{A}[\omega_1 \circ \omega_2]^{\#}$	by i.h. on $\mathcal{E}_1$
$= \boldsymbol{A}^{\#}[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\|\Omega_0\| \vdash^{\mathbf{lf}} \boldsymbol{A} : \boldsymbol{type}}{\Omega_0 \vdash \boldsymbol{A} \text{ wff}}\,\text{LF\_wff}$

$\Omega_1 \vdash \omega_1 : \Omega_0$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$(\omega_1 \circ \omega_2)$ exists	by assumption
$\|\Omega_0\| \vdash^{\mathbf{lf}} \boldsymbol{A} : \boldsymbol{type}$	by assumption
$\|\Omega_1\| \vdash^{\mathbf{lf}} \|\omega_1\| : \|\Omega_0\|$	by Lemma B.4.7
$\|\Omega_2\| \vdash^{\mathbf{lf}} \|\omega_2\| : \|\Omega_1\|$	by Lemma B.4.7
$\boldsymbol{A}[\omega_1][\omega_2]$	
$= \boldsymbol{A}[\|\omega_1\|][\|\omega_2\|]$	by Subst. App. (Def. A.2.4)
$= \boldsymbol{A}[\|\omega_1\| \circ^{\mathbf{lf}} \|\omega_2\|]$	by Lemma B.1.16
$= \boldsymbol{A}[\|\omega_1 \circ \omega_2\|]$	by Lemma B.11.3
$= \boldsymbol{A}[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{}{\Omega_0 \vdash \text{unit wff}}\,\text{unitwff}$

$(\omega_1 \circ \omega_2)$ exists	by assumption
$\text{unit}[\omega_1][\omega_2] = \text{unit} = \text{unit}[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega_0 \vdash \tau\ \mathsf{wff}\end{array}}{\Omega_0 \vdash \forall nil.\ \tau\ \mathsf{wff}}\ \forall_b \mathsf{wff}$

$\Omega_1 \vdash \omega_1 : \Omega_0$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$(\omega_1 \circ \omega_2)$ exists by assumption
$(\forall nil.\ \tau)[\omega_1][\omega_2]$
$= \forall nil.\ \tau[\omega_1][\omega_2]$ by Subst. App. (Def. A.2.4)
$= \forall nil.\ \tau[\omega_1 \circ \omega_2]$ by i.h. on $\mathcal{E}_1$
$= (\forall nil.\ \tau)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2\\ \Omega_0 \vdash \delta_1\ \mathsf{wff} & \Omega_0, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta'}.\ \tau\ \mathsf{wff}\end{array}}{\Omega_0 \vdash \forall \alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta'}.\ \tau\ \mathsf{wff}}\ \forall_i \mathsf{wff}$

$\Omega_1 \vdash \omega_1 : \Omega_0$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$(\omega_1 \circ \omega_2)$ exists by assumption
$\Omega_0 \vdash \forall \alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta'}.\ \tau\ \mathsf{wff}$ by assumption
$\Omega_0 \vdash \delta_1\ \mathsf{wff}$ by assumption ($\mathcal{E}_1$)
$\Omega_0, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta'}.\ \tau\ \mathsf{wff}$ by assumption ($\mathcal{E}_2$)
$\Omega_1 \vdash \delta_1[\omega_1]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_1\ \mathsf{ctx}$ by Lemma B.3.2
$(\Omega_1, \alpha_1{\in}\delta_1[\omega_1])\ \mathsf{ctx}$ by ctxAdd
$(\Omega_1, \alpha_1{\in}\delta_1[\omega_1]) \vdash \alpha_1 \in \delta_1[\uparrow_\alpha \omega_1]$ by Lemma B.5.1 and Lemma B.5.4
$(\Omega_1, \alpha_1{\in}\delta_1[\omega_1]) \vdash \uparrow_{\alpha_1} \omega_1 : \Omega_0$ by tpSubShift
$(\Omega_1, \alpha_1{\in}\delta_1[\omega_1]) \vdash (\uparrow_{\alpha_1} \omega_1, \alpha_1/\alpha_1) : (\Omega_0, \alpha_1{\in}\delta_1)$ by tpSubInd
$\Omega_2 \vdash \delta_1[\omega_1][\omega_2]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_2\ \mathsf{ctx}$ by Lemma B.3.2
$(\Omega_2, \alpha_1{\in}\delta_1[\omega_1][\omega_2])\ \mathsf{ctx}$ by ctxAdd
$(\Omega_2, \alpha_1{\in}\delta_1[\omega_1][\omega_2]) \vdash \alpha_1 \in \delta_1[\omega_1][\uparrow_{\alpha_1} \omega_2]$
by Lemma B.5.1 and Lemma B.5.4
$(\Omega_2, \alpha_1{\in}\delta_1[\omega_1][\omega_2]) \vdash \uparrow_{\alpha_1} \omega_2 : \Omega_1$ by tpSubShift
$(\Omega_2, \alpha_1{\in}\delta_1[\omega_1][\omega_2]) \vdash (\uparrow_{\alpha_1} \omega_2, \alpha_1/\alpha_1) : (\Omega_1, \alpha_1{\in}\delta_1[\omega_1])$ by tpSubInd

(Case Continued →)

$(\uparrow_{\alpha_1} \omega_1, \alpha_1/\alpha_1) \circ (\uparrow_{\alpha_1} \omega_2, \alpha_1/\alpha_1)$
$\quad = (\omega_1 \circ \omega_2), \alpha_1/\alpha_1$
by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)

$(\uparrow_{\alpha_1} \omega_1, \alpha_1/\alpha_1) \circ (\uparrow_{\alpha_1} \omega_2, \alpha_1/\alpha_1)$ exists by above

We now define a transformation function, *trans*, such that

$\quad trans\ (\forall \alpha_1 \in \delta_1.\ (\forall \overline{\alpha \in \delta'}.\ \tau)) = \forall \alpha_1 \in \delta_1; \overline{\alpha \in \delta'}.\ \tau$

We will need the following property of *trans*:

for all σ^* and ω^*, $(trans\ \sigma^*)[\omega^*] = trans\ (\sigma^*[\omega^*])$

Proof is straightforward using Subst. App. (Def. A.2.4)

$(\forall \alpha_1 \in \delta_1; \overline{\alpha \in \delta'}.\ \tau)[\omega_1][\omega_2]$
$= (trans\ (\forall \alpha_1 \in \delta_1.\ (\forall \overline{\alpha \in \delta'}.\ \tau)))[\omega_1][\omega_2]$
$= (trans\ (\forall \alpha_1 \in \delta_1[\omega_1].\ (\forall \overline{\alpha \in \delta'}.\ \tau)[\uparrow_{\alpha_1} \omega_1, \alpha_1/\alpha_1]))[\omega_2]$
by Subst. App. (Def. A.2.4)
$= trans\ (\forall \alpha_1 \in \delta_1[\omega_1][\omega_2].\ (\forall \overline{\alpha \in \delta'}.\ \tau)[\uparrow_{\alpha_1} \omega_1, \alpha_1/\alpha_1][\uparrow_{\alpha_1} \omega_2, \alpha_1/\alpha_1])$
by Subst. App. (Def. A.2.4)
$= trans\ (\forall \alpha_1 \in \delta_1[\omega_1 \circ \omega_2].\ (\forall \overline{\alpha \in \delta'}.\ \tau)[\uparrow_{\alpha_1} \omega_1, \alpha_1/\alpha_1][\uparrow_{\alpha_1} \omega_2, \alpha_1/\alpha_1])$
by i.h. on $\mathcal{E}_1$
$= trans\ (\forall \alpha_1 \in \delta_1[\omega_1 \circ \omega_2].\ (\forall \overline{\alpha \in \delta'}.\ \tau)[(\uparrow_{\alpha_1} \omega_1, \alpha_1/\alpha_1) \circ (\uparrow_{\alpha_1} \omega_2, \alpha_1/\alpha_1)])$
by i.h. on $\mathcal{E}_2$
$= trans\ (\forall \alpha_1 \in \delta_1[\omega_1 \circ \omega_2].\ (\forall \overline{\alpha \in \delta'}.\ \tau)[((\uparrow_{\alpha_1} \omega_1) \circ (\uparrow_{\alpha_1} \omega_2, \alpha_1/\alpha_1)), \alpha_1/\alpha_1])$
by Composition (Def. A.2.3)
$= trans\ (\forall \alpha_1 \in \delta_1[\omega_1 \circ \omega_2].\ (\forall \overline{\alpha \in \delta'}.\ \tau)[(\omega_1 \circ (\uparrow_{\alpha_1} \omega_2)), \alpha_1/\alpha_1])$
by Composition (Def. A.2.3)
$= trans\ (\forall \alpha_1 \in \delta_1[\omega_1 \circ \omega_2].\ (\forall \overline{\alpha \in \delta'}.\ \tau)[\uparrow_{\alpha_1} (\omega_1 \circ \omega_2), \alpha_1/\alpha_1])$
by Composition (Def. A.2.3)
$= trans\ (\forall \alpha_1 \in \delta_1.\ (\forall \overline{\alpha \in \delta'}.\ \tau))[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)
$= (\forall \alpha_1 \in \delta_1; \overline{\alpha \in \delta'}.\ \tau)[\omega_1 \circ \omega_2]$ by *trans*

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega_0 \vdash \delta' \ \mathsf{wff} & \Omega_0, \alpha{\in}\delta' \vdash \tau \ \mathsf{wff}\end{array}}{\Omega_0 \vdash \exists \alpha{\in}\delta'.\ \tau \ \mathsf{wff}}\ \exists\mathsf{wff}$

$\Omega_1 \vdash \omega_1 : \Omega_0$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$(\omega_1 \circ \omega_2)$ exists	by assumption
$\Omega_0 \vdash \exists \alpha{\in}\delta'.\ \tau \ \mathsf{wff}$	by assumption
$\Omega_0 \vdash \delta' \ \mathsf{wff}$	by assumption ($\mathcal{E}_1$)
$\Omega_0, \alpha{\in}\delta' \vdash \tau \ \mathsf{wff}$	by assumption ($\mathcal{E}_2$)
$\Omega_1 \vdash \delta'[\omega_1] \ \mathsf{wff}$	by Lemma B.6.1
$\Omega_1 \ \mathsf{ctx}$	by Lemma B.3.2
$(\Omega_1, \alpha{\in}\delta'[\omega_1]) \ \mathsf{ctx}$	by ctxAdd
$(\Omega_1, \alpha{\in}\delta'[\omega_1]) \vdash \alpha \in \delta'[\uparrow_\alpha \omega_1]$	by Lemma B.5.1 and Lemma B.5.4
$(\Omega_1, \alpha{\in}\delta'[\omega_1]) \vdash \uparrow_\alpha \omega_1 : \Omega_0$	by tpSubShift
$(\Omega_1, \alpha{\in}\delta'[\omega_1]) \vdash (\uparrow_\alpha \omega_1, \alpha/\alpha) : (\Omega_0, \alpha{\in}\delta')$	by tpSubInd
$\Omega_2 \vdash \delta'[\omega_1][\omega_2] \ \mathsf{wff}$	by Lemma B.6.1
$\Omega_2 \ \mathsf{ctx}$	by Lemma B.3.2
$(\Omega_2, \alpha{\in}\delta'[\omega_1][\omega_2]) \ \mathsf{ctx}$	by ctxAdd
$(\Omega_2, \alpha{\in}\delta'[\omega_1][\omega_2]) \vdash \alpha \in \delta'[\omega_1][\uparrow_\alpha \omega_2]$	by Lemma B.5.1 and Lemma B.5.4
$(\Omega_2, \alpha{\in}\delta'[\omega_1][\omega_2]) \vdash \uparrow_\alpha \omega_2 : \Omega_1$	by tpSubShift
$(\Omega_2, \alpha{\in}\delta'[\omega_1][\omega_2]) \vdash (\uparrow_\alpha \omega_2, \alpha/\alpha) : (\Omega_1, \alpha{\in}\delta'[\omega_1])$	by tpSubInd
$(\uparrow_\alpha \omega_1, \alpha/\alpha) \circ (\uparrow_\alpha \omega_2, \alpha/\alpha)$	
$\quad = (\omega_1 \circ \omega_2), \alpha/\alpha$	by Comp. (Def. A.2.3) and Subst. App. (Def. A.2.4)
$(\uparrow_\alpha \omega_1, \alpha/\alpha) \circ (\uparrow_\alpha \omega_2, \alpha/\alpha)$ exists	by above
$(\exists \alpha{\in}\delta'.\ \tau)[\omega_1][\omega_2]$	
$\quad = (\exists \alpha{\in}\delta'[\omega_1].\ \tau[\uparrow_\alpha \omega_1, \alpha/\alpha])[\omega_2]$	by Subst. App. (Def. A.2.4)
$\quad = \exists \alpha{\in}\delta'[\omega_1][\omega_2].\ \tau[\uparrow_\alpha \omega_1, \alpha/\alpha][\uparrow_\alpha \omega_2, \alpha/\alpha]$	by Subst. App. (Def. A.2.4)
$\quad = \exists \alpha{\in}\delta'[\omega_1 \circ \omega_2].\ \tau[\uparrow_\alpha \omega_1, \alpha/\alpha][\uparrow_\alpha \omega_2, \alpha/\alpha]$	by i.h. on $\mathcal{E}_1$
$\quad = \exists \alpha{\in}\delta'[\omega_1 \circ \omega_2].\ \tau[(\uparrow_\alpha \omega_1, \alpha/\alpha) \circ (\uparrow_\alpha \omega_2, \alpha/\alpha)]$	by i.h. on $\mathcal{E}_2$
$\quad = \exists \alpha{\in}\delta'[\omega_1 \circ \omega_2].\ \tau[((\uparrow_\alpha \omega_1) \circ (\uparrow_\alpha \omega_2, \alpha/\alpha)), \alpha[\uparrow_\alpha \omega_2, \alpha/\alpha]/\alpha]$	by Composition (Def. A.2.3)
$\quad = \exists \alpha{\in}\delta'[\omega_1 \circ \omega_2].\ \tau[((\uparrow_\alpha \omega_1) \circ (\uparrow_\alpha \omega_2, \alpha/\alpha)), \alpha/\alpha]$	by Subst. App. (Def. A.2.4)
$\quad = \exists \alpha{\in}\delta'[\omega_1 \circ \omega_2].\ \tau[(\omega_1 \circ (\uparrow_\alpha \omega_2)), \alpha/\alpha]$	by Composition (Def. A.2.3)
$\quad = \exists \alpha{\in}\delta'[\omega_1 \circ \omega_2].\ \tau[\uparrow_\alpha (\omega_1 \circ \omega_2), \alpha/\alpha]$	by Composition (Def. A.2.3)
$\quad = (\exists \alpha{\in}\delta'.\ \tau)[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}} \qquad \overset{\mathcal{E}_2}{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}}}{\Omega \vdash \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}}\ \nabla\mathsf{wff}$

$\Omega_1 \vdash \omega_1 : \Omega_0$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$(\omega_1 \circ \omega_2)$ exists by assumption
$\Omega_0 \vdash \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}$ by assumption
$\Omega_0 \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$ by assumption ($\mathcal{E}_1$)
$\Omega_0, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}$ by assumption ($\mathcal{E}_2$)
$\Omega_1 \vdash \boldsymbol{A}^{\#}[\omega_1]\ \mathsf{wff}$ by Lemma B.6.1
$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \vdash (\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}) : (\Omega_0, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ by tpSubIndNew
$\Omega_2 \vdash \boldsymbol{A}^{\#}[\omega_1][\omega_2]\ \mathsf{wff}$ by Lemma B.6.1
$(\Omega_2, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1][\omega_2]) \vdash (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x}) : (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1])$ by tpSubIndNew
$(\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x})$
$\quad = (\omega_1 \circ \omega_2), \boldsymbol{x}/\boldsymbol{x}$
by Comp. (Def. A.2.3) and Subst. App. (Def. A.2.4)
$(\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x})$ exists by above
$(\nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)[\omega_1][\omega_2]$
$\quad = (\nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1].\ \tau[\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}])[\omega_2]$ by Subst. App. (Def. A.2.4)
$\quad = \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1][\omega_2].\ \tau[\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}][\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x}]$
by Subst. App. (Def. A.2.4)
$\quad = \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1 \circ \omega_2].\ \tau[\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}][\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x}]$ by i.h. on $\mathcal{E}_1$
$\quad = \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1 \circ \omega_2].\ \tau[(\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x})]$ by i.h. on $\mathcal{E}_2$
$\quad = \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1 \circ \omega_2].\ \tau[((\uparrow_{\boldsymbol{x}} \omega_1) \circ (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x})), \boldsymbol{x}[\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x}]/\boldsymbol{x}]$
by Composition (Def. A.2.3)
$\quad = \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1 \circ \omega_2].\ \tau[((\uparrow_{\boldsymbol{x}} \omega_1) \circ (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x})), \boldsymbol{x}/\boldsymbol{x}]$
by Subst. App. (Def. A.2.4)
$\quad = \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1 \circ \omega_2].\ \tau[(\omega_1 \circ (\uparrow_{\boldsymbol{x}} \omega_2)), \boldsymbol{x}/\boldsymbol{x}]$ by Composition (Def. A.2.3)
$\quad = \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1 \circ \omega_2].\ \tau[\uparrow_{\boldsymbol{x}} (\omega_1 \circ \omega_2), \boldsymbol{x}/\boldsymbol{x}]$
by Composition (Def. A.2.3)
$\quad = (\nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{W}\ \mathsf{world} & \Omega \vdash \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}}\ \nabla_{\mathrm{world}}\mathsf{wff}$ ($\mathcal{E}_1$)

$\Omega_1 \vdash \omega_1 : \Omega_0$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$(\omega_1 \circ \omega_2)$ exists	by assumption
$\Omega_0 \vdash \nabla\mathcal{W}.\ \tau\ \mathsf{wff}$	by assumption
$\Omega_0 \vdash \tau\ \mathsf{wff}$	by assumption ($\mathcal{E}_1$)
$(\nabla\mathcal{W}.\ \tau)[\omega_1][\omega_2]$	
$= \nabla\mathcal{W}.\ \tau[\omega_1][\omega_2]$	by Subst. App. (Def. A.2.4)
$= \nabla\mathcal{W}.\ \tau[\omega_1 \circ \omega_2]$	by i.h. on $\mathcal{E}_1$
$= (\nabla\mathcal{W}.\ \tau)[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

□

Lemma B.12.2 (Parameter Variables under Substitutions Preserve Typing)**.** *If* $((\boldsymbol{y}{\in}\boldsymbol{B}^{\#})$ *or* $(\boldsymbol{y}\overset{\triangledown}{\in}\boldsymbol{B}^{\#}))$ in Ω *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega]$*.*

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$.

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\,\mathsf{tpSubBase}$

This case is vacuously true, as there is no $\boldsymbol{y}{\in}\boldsymbol{B}^{\#}$ and no $\boldsymbol{y}\overset{\triangledown}{\in}\boldsymbol{B}^{\#}$ in $\cdot$.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\,\mathsf{tpSubInd}$

$((\boldsymbol{y}{\in}\boldsymbol{B}^{\#})$ or $(\boldsymbol{y}\overset{\triangledown}{\in}\boldsymbol{B}^{\#}))$ in $\Omega, \alpha{\in}\delta$	by assumption

Subcase: $\alpha = \boldsymbol{y}$ and $\delta = \boldsymbol{B}^{\#}$

$\boldsymbol{y}[\omega, e/\alpha] = \boldsymbol{y}[\omega, e/\boldsymbol{y}] = e$	by assumption and Subst. App. (Def. A.2.4)
$\Omega' \vdash e \in \delta[\omega]$	by assumption
$\Omega' \vdash \delta[\omega]$ wff	by Lemma B.8.1
$\omega \leq (\omega, e/\alpha)$	by leSubAdd
$\Omega' \vdash e \in \delta[\omega, e/\alpha]$	by Lemma B.7.4
$\Omega' \vdash \boldsymbol{y}[\omega, e/\alpha] \in \boldsymbol{B}^{\#}[\omega, e/\alpha]$	by above

Subcase: $\alpha \neq \boldsymbol{y}$

$\boldsymbol{y}[\omega, e/\alpha] = \boldsymbol{y}[\omega]$	by Subst. App. (Def. A.2.4)
$((\boldsymbol{y}{\in}\boldsymbol{B}^{\#})$ or $(\boldsymbol{y}\overset{\triangledown}{\in}\boldsymbol{B}^{\#}))$ in Ω	by above
$\Omega' \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega]$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{B}^{\#}[\omega]$ wff	by Lemma B.8.1
$\omega \leq (\omega, e/\alpha)$	by leSubAdd
$\Omega' \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega, e/\alpha]$	by Lemma B.7.4
$\Omega' \vdash \boldsymbol{y}[\omega, e/\alpha] \in \boldsymbol{B}^{\#}[\omega, e/\alpha]$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}) : (\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

$((\boldsymbol{y} \in \boldsymbol{B}^{\#})$ or $(\boldsymbol{y} \overset{\triangledown}{\in} \boldsymbol{B}^{\#}))$ in $(\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ by assumption

Subcase: $\boldsymbol{x} = \boldsymbol{y}$ and $\boldsymbol{A}^{\#} = \boldsymbol{B}^{\#}$

$\boldsymbol{y}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}] = \boldsymbol{y}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{y}] = \boldsymbol{x_2}$ by assumption and Subst. App. (Def. A.2.4)

$(\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega])\ \mathsf{ctx}$ by Lemma B.3.2 on $\mathcal{E}$

$\Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}$ by inversion using ctxAdd

$\Omega' \leq_* \Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega]$ by leWorldAddNew with includesAny

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}$ by Lemma B.7.1

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{A}^{\#}[\uparrow_{\boldsymbol{x_2}} \omega]\ \mathsf{wff}$ by Lemma B.5.4

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{x_2} \in \boldsymbol{A}^{\#}[\omega]$ by Lemma B.5.1

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{x_2} \in \boldsymbol{A}^{\#}[\uparrow_{\boldsymbol{x_2}} \omega]$ by Lemma B.5.4

$(\uparrow_{\boldsymbol{x_2}} \omega) \leq (\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x})$ By leSubAdd

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{x_2} \in \boldsymbol{A}^{\#}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}]$ by Lemma B.7.4

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{y}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}] \in \boldsymbol{B}^{\#}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}]$ by above

Subcase: $\boldsymbol{x} \neq \boldsymbol{y}$

$\boldsymbol{y}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}]$
$= \boldsymbol{y}[\uparrow_{\boldsymbol{x_2}} \omega]$ by Subst. App. (Def. A.2.4)
$= \boldsymbol{y}[\omega]$ by Lemma B.5.4

$((\boldsymbol{y} \in \boldsymbol{B}^{\#})$ or $(\boldsymbol{y} \overset{\triangledown}{\in} \boldsymbol{B}^{\#}))$ in Ω by above

$\Omega' \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega]$ by i.h. on $\mathcal{E}_1$

$\Omega' \vdash \boldsymbol{B}^{\#}[\omega]\ \mathsf{wff}$ by Lemma B.8.1

$(\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega])\ \mathsf{ctx}$ by Lemma B.3.2 on $\mathcal{E}$

$\Omega' \leq_* \Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega]$ by leWorldAddNew with includesAny

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega]$ by Lemma B.7.9

$\omega \leq (\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x})$ by leSubShift and leSubAdd

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}]$ by Lemma B.7.4

$\Omega', \boldsymbol{x_2} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega] \vdash \boldsymbol{y}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}] \in \boldsymbol{B}^{\#}[\uparrow_{\boldsymbol{x_2}} \omega, \boldsymbol{x_2}/\boldsymbol{x}]$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u_2 \overset{\triangledown}{\in} \mathcal{W}) \vdash (\uparrow_{u_2} \omega, u_2/u) : (\Omega, u \overset{\triangledown}{\in} \mathcal{W})}\ \mathsf{tpSubWorld}$

$((\boldsymbol{y} \in \boldsymbol{B}^{\#})$ or $(\boldsymbol{y} \overset{\triangledown}{\in} \boldsymbol{B}^{\#}))$ in $(\Omega, \mathrm{u} \overset{\triangledown}{\in} \mathcal{W})$	by assumption
$((\boldsymbol{y} \in \boldsymbol{B}^{\#})$ or $(\boldsymbol{y} \overset{\triangledown}{\in} \boldsymbol{B}^{\#}))$ in Ω	by above
$\boldsymbol{y}[\uparrow_{u_2} \omega, u_2/u]$	
$\quad = \boldsymbol{y}[\uparrow_{u_2} \omega]$	by Subst. App. (Def. A.2.4)
$\quad = \boldsymbol{y}[\omega]$	by Lemma B.5.4
$\Omega' \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega]$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{B}^{\#}[\omega]\ \mathsf{wff}$	by Lemma B.8.1
$(\Omega', u_2 \overset{\triangledown}{\in} \mathcal{W})\ \mathsf{ctx}$	by Lemma B.3.2 on $\mathcal{E}$
$\Omega' \leq_* \Omega', u_2 \overset{\triangledown}{\in} \mathcal{W}$	by leWorldAddNew with includesAny
$\Omega', u_2 \overset{\triangledown}{\in} \mathcal{W} \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega]$	by Lemma B.7.9
$\omega \leq (\uparrow_{u_2} \omega, u_2/u)$	by leSubShift and leSubAdd
$\Omega', u_2 \overset{\triangledown}{\in} \mathcal{W} \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\uparrow_{u_2} \omega, u_2/u]$	by Lemma B.7.4
$\Omega', u_2 \overset{\triangledown}{\in} \mathcal{W} \vdash \boldsymbol{y}[\uparrow_{u_2} \omega, u_2/u] \in \boldsymbol{B}^{\#}[\uparrow_{u_2} \omega, u_2/u]$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta\ \mathsf{wff}\end{array}}{\Omega', \alpha \in \delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

$((\boldsymbol{y} \in \boldsymbol{B}^{\#})$ or $(\boldsymbol{y} \overset{\triangledown}{\in} \boldsymbol{B}^{\#}))$ in Ω	by assumption
$\Omega' \vdash \boldsymbol{y}[\omega] \in \boldsymbol{B}^{\#}[\omega]$	by i.h. on $\mathcal{E}_1$
$\Omega' \vdash \boldsymbol{y}[\uparrow_\alpha \omega] \in \boldsymbol{B}^{\#}[\uparrow_\alpha \omega]$	by Lemma B.5.4
$\Omega' \leq_* \Omega', \alpha \in \delta$	by leWorldAdd
$(\Omega', \alpha \in \delta)\ \mathsf{ctx}$	by Lemma B.3.2 on $\mathcal{E}$
$\Omega', \alpha \in \delta \vdash \boldsymbol{y}[\uparrow_\alpha \omega] \in \boldsymbol{B}^{\#}[\uparrow_\alpha \omega]$	by Lemma B.7.9

□

Lemma B.12.3 (Substitution Preserves Typing (Restricted Form of Context))**.**
If $\Omega \vdash e \in \delta$ *and* $\Omega' \vdash \omega : \Omega$ *and* $\delta = \boldsymbol{A}$ *or* $\delta = \boldsymbol{A}^{\#}$
and Ω *only contains declarations of the from* $\boldsymbol{x'} \in \delta'$*,*
then $\Omega' \vdash e[\omega] \in \delta[\omega]$*.*

Proof. By case analysis on $\mathcal{E} :: \Omega \vdash e \in \tau$.

Case: $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad \|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$\mathcal{F} :: \Omega' \vdash \omega : \Omega$	by assumption
$\|\Omega'\| \vdash^{\mathsf{lf}} \|\omega\| : \|\Omega\|$	by Lemma B.4.7
$\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by assumption
$\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{M}[\|\omega\|] : \boldsymbol{A}[\|\omega\|]$	by Lemma B.1.12.1
$\Omega'\ \mathsf{ctx}$	by Lemma B.3.2 on $\mathcal{F}$
$\Omega' \vdash \boldsymbol{M}[\|\omega\|] \in \boldsymbol{A}[\|\omega\|]$	by isLF
$\Omega' \vdash \boldsymbol{M}[\omega] \in \boldsymbol{A}[\omega]$	by Subst. Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad ((\boldsymbol{x} \in \boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\ \mathsf{var}^{\#}$

$\mathcal{F} :: \Omega' \vdash \omega : \Omega$	by assumption
$((\boldsymbol{x} \in \boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}))$ in Ω	by assumption
Ω only contains declarations of the from $\boldsymbol{x'} \in \delta'$	by assumption
$(\boldsymbol{x} \in \boldsymbol{A}^{\#})$ in Ω	by above
$\Omega' \vdash \boldsymbol{x}[\omega] \in \boldsymbol{A}^{\#}[\omega]$	by Lemma B.12.2

□

Lemma B.12.4 (Substitution Composition is Well-Typed (Restricted Form))**.**
If Ω ctx *and* $\Omega' \vdash \omega : \Omega$ *and* $\Omega_2 \vdash \omega_2 : \Omega'$
and all contexts Ω, Ω_2, Ω' *only contain declarations of the form* $\boldsymbol{x}{\in}\delta$,
then $\Omega_2 \vdash (\omega \circ \omega_2) : \Omega$.

Proof. By induction lexicographically on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$ and $\mathcal{F} :: \Omega_2 : \omega_2 : \Omega'$ utilizing that all declarations are of the form $\boldsymbol{x}{\in}\delta$.

Case: $\mathcal{E} = \Omega' \vdash \omega : \Omega$

$$\text{and } \mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash \delta\ \mathsf{wff}\end{array}}{\Omega_2, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}} \omega_2 : \Omega}\ \mathsf{tpSubShift}$$

Ω ctx	by assumption
All contexts only contain declarations of the form $\boldsymbol{x'}{\in}\delta'$	by assumption
$\Omega_2 \vdash \omega \circ \omega_2 : \Omega$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\Omega_2 \vdash \delta\ \mathsf{wff}$	by assumption
$\Omega_2, \boldsymbol{x}{\in}\delta \vdash \uparrow_{\boldsymbol{x}} (\omega \circ \omega_2) : \Omega$	by $\mathsf{tpSubShift}$
$\Omega_2, \boldsymbol{x}{\in}\delta \vdash \omega \circ (\uparrow_{\boldsymbol{x}} \omega_2) : \Omega$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \cdot$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

Ω ctx	by assumption
$\Omega_2 \vdash \omega_2 : \cdot$	by assumption ($\mathcal{F}$)
$\Omega_2 \vdash (\cdot \circ \omega_2) : \cdot$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \\ \Omega' \vdash \omega : \Omega & \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\boldsymbol{x}) : (\Omega, \boldsymbol{x}{\in}\delta)}\,\mathsf{tpSubInd}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq\!\uparrow_{\alpha''} \omega_2''$

$(\Omega, \boldsymbol{x}{\in}\delta)$ ctx	by assumption
All contexts only contain declarations of the form $\boldsymbol{x'}{\in}\delta'$	by assumption
Ω ctx	by inversion using ctxAdd
$\Omega \vdash \delta$ wff	by inversion using ctxAdd
$\Omega_2 \vdash (\omega \circ \omega_2) : \Omega$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$
$\Omega' \vdash e \in \delta[\omega]$	by assumption
$\Omega_2 \vdash e[\omega_2] \in \delta[\omega][\omega_2]$	by Lemma B.12.3
$\Omega_2 \vdash e[\omega_2] \in \delta[\omega \circ \omega_2]$	by Lemma B.12.1
$\Omega_2 \vdash ((\omega \circ \omega_2), e[\omega_2]/\boldsymbol{x}) : \Omega, \boldsymbol{x}{\in}\delta$	by tpSubInd
$\Omega_2 \vdash ((\omega, e/\boldsymbol{x}) \circ \omega_2) : \Omega, \boldsymbol{x}{\in}\delta$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \\ \Omega' \vdash \omega : \Omega & \Omega' \vdash \delta \text{ wff}\end{array}}{\Omega', \boldsymbol{x}{\in}\delta \vdash\!\uparrow_{\boldsymbol{x}} \omega : \Omega}\,\mathsf{tpSubShift}$

and $\mathcal{F} = \dfrac{\begin{array}{cc}\mathcal{F}_1 & \\ \Omega_2 \vdash \omega_2 : \Omega' & \Omega_2 \vdash e \in \delta[\omega_2]\end{array}}{\Omega_2 \vdash (\omega_2, e/\boldsymbol{x}) : (\Omega', \boldsymbol{x}{\in}\delta)}\,\mathsf{tpSubInd}$

Ω ctx	by assumption
All contexts only contain declarations of the form $\boldsymbol{x'}{\in}\delta'$	by assumption
$\Omega_2 \vdash \omega \circ \omega_2 : \Omega$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\Omega_2 \vdash (\uparrow_{\boldsymbol{x}} \omega) \circ (\omega_2, e/\boldsymbol{x}) : \Omega$	by Composition (Def. A.2.3)

□

Lemma B.12.5 (Substitution Property of World Inclusion (Alt))**.**
If $(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$ *and* $\mathcal{W}$ world *and* $\|\Omega_2\|^{\nabla} \vdash \omega_2 : \|\Omega\|^{\nabla}$*, then* $(\Omega_2, \boldsymbol{A}[\omega]) \in_{\text{alt}} \mathcal{W}$*.*

Proof. By induction on $(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$.

Case: $\mathcal{E} = \dfrac{\|\Omega\|^{\nabla} \vdash \omega : \Omega'}{(\Omega, \boldsymbol{A'}[\omega]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}$ includesAltYes

$\|\Omega_2\|^{\nabla} \vdash \omega_2 : \|\Omega\|^{\nabla}$	by assumption
$(\Omega', (\Omega', \boldsymbol{A'}))$ world	by assumption
Ω' ctx	by inversion using worldAdd
Ω', $\|\Omega\|^{\nabla}$, and $\|\Omega_2\|^{\nabla}$ only contains declarations of the form $\boldsymbol{x}{\in}\delta$	by inversion using worldAdd and Casting (Def. B.2.1)
$\|\Omega\|^{\nabla} \vdash \omega : \Omega'$	by assumption (premise)
$\|\Omega_2\|^{\nabla} \vdash \omega \circ \omega_2 : \Omega'$	by Lemma B.12.4
$(\Omega_2, \boldsymbol{A'}[\omega \circ \omega_2]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by includesAltYes
$\boldsymbol{A'}[\omega][\omega_2] = \boldsymbol{A'}[\omega \circ \omega_2]$	by Lemma B.12.1
$(\Omega_2, \boldsymbol{A'}[\omega][\omega_2]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ (\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}'\end{array}}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))}$ includesAltOther

$\|\Omega_2\|^{\nabla} \vdash \omega_2 : \Omega$	by assumption
$(\mathcal{W}', (\Omega', \boldsymbol{A'}))$ world	by assumption
$\mathcal{W}'$ world	by worldAdd
$(\Omega_2, \boldsymbol{A}[\omega_2]) \in_{\text{alt}} \mathcal{W}'$	by i.h. on $\mathcal{E}_1$
$(\Omega_2, \boldsymbol{A}[\omega_2]) \in_{\text{alt}} (\mathcal{W}', (\Omega', \boldsymbol{A'}))$	by includesAltOther

Case: $\mathcal{E} = \dfrac{}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} *}$ includesAltAny

$(\Omega_2, \boldsymbol{A}[\omega_2]) \in_{\text{alt}} *$	by includesAltAny

□

Lemma B.12.6 (Substitution Property of World Inclusion)**.**
If $(\Omega, \boldsymbol{A}) \in \mathcal{W}$ *and* $\mathcal{W}$ world *and* $\Omega_2 \vdash \omega_2 : \Omega$*, then* $(\Omega_2, \boldsymbol{A}[\omega]) \in \mathcal{W}$*.*

Proof.

$(\Omega, \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\mathcal{W}$ world	by assumption
$\Omega_2 \vdash \omega_2 : \Omega$	by assumption
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$	by Lemma B.9.2
$(\Omega_2, \boldsymbol{A}[\omega]) \in_{\text{alt}} \mathcal{W}$	by Lemma B.12.5
Ω_2 ctx	by Lemma B.3.2
$(\Omega_2, \boldsymbol{A}[\omega]) \in \mathcal{W}$	by Lemma B.9.1

□

B.13 Meta-Theory: Weakening with Worlds

Lemma B.13.1 (World Subsumption Preserves Inclusion (Alt))**.**
If $(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_1$ *and* $\mathcal{W}_1 \leq \mathcal{W}_2$ *and* $\mathcal{W}_2$ world*, then* $(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_2$*.*

Proof. By induction on $\mathcal{E} :: (\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_1$.

Case: $\mathcal{E} = \dfrac{\|\Omega\|^{\nabla} \vdash \omega : \Omega'}{(\Omega, \boldsymbol{A'}[\omega]) \in_{\text{alt}} (\mathcal{W}_1, (\Omega', \boldsymbol{A'}))}$ includesAltYes

$\mathcal{W}_1, (\Omega', \boldsymbol{A'}) \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2$ world	by assumption
$(\Omega', \boldsymbol{A'}) \in \mathcal{W}_2$	by inversion using containsSubsume
$(\Omega', \boldsymbol{A'}) \in_{\text{alt}} \mathcal{W}_2$	by Lemma B.9.2
$\|\Omega\|^{\nabla} \vdash \omega : \Omega'$	by assumption
$\|\Omega\|^{\nabla} \vdash \|\omega\|^{\nabla} : \|\Omega'\|^{\nabla}$	by Lemma B.4.8 noting that $\|\|\Omega\|^{\nabla}\|^{\nabla} = \|\Omega\|^{\nabla}$
$(\Omega, \boldsymbol{A'}[\|\omega\|^{\nabla}]) \in_{\text{alt}} \mathcal{W}_2$	by Lemma B.12.5
$\boldsymbol{A'}[\omega] = \boldsymbol{A'}[\|\omega\|] = \boldsymbol{A'}[\|\|\omega\|^{\nabla}\|] = \boldsymbol{A'}[\|\omega\|^{\nabla}]$	by Subst. Application (Def. A.2.4) and Lemma B.4.2
$(\Omega, \boldsymbol{A'}[\omega]) \in_{\text{alt}} \mathcal{W}_2$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ (\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_1\end{array}}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} (\mathcal{W}_1, (\Omega', \boldsymbol{A'}))}$ includesAltOther

$(\mathcal{W}_1, (\Omega', \boldsymbol{A'})) \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2$ world	by assumption
$\mathcal{W}_1 \leq \mathcal{W}_2$	by inversion we see that anySubsume or containsSubsume may apply. If it is anySubsume, then the property holds by the same rule. If it is containsSubsume, then the property holds by inversion.
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_2$	by i.h. on $\mathcal{E}_1$

Case: $\mathcal{E} = \dfrac{}{(\Omega, \boldsymbol{A}) \in_{\text{alt}} *}$ includesAltAny

$* \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2 = *$	by inversion using anySubsume
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_2$	by $\mathcal{E}$ and above

□

Lemma B.13.2 (World Subsumption Preserves Inclusion)**.**
If $(\Omega, \boldsymbol{A}) \in \mathcal{W}_1$ *and* $\mathcal{W}_1 \leq \mathcal{W}_2$ *and* $\mathcal{W}_2$ world*, then* $(\Omega, \boldsymbol{A}) \in \mathcal{W}_2$.

Proof.

$(\Omega, \boldsymbol{A}) \in \mathcal{W}_1$	by assumption
$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2$ world	by assumption
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_1$	by Lemma B.9.2
$(\Omega, \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}_2$	by Lemma B.13.1

By inversion on $(\Omega, \boldsymbol{A}) \in \mathcal{W}_1$ we see that either:

Case: There exists a substitution with a codomain of Ω (from includesYes)

Ω ctx	by Lemma B.3.2
$(\Omega, \boldsymbol{A}) \in \mathcal{W}_2$	by Lemma B.9.1

Case: There exists $*$ in $\mathcal{W}_1$

There exists $*$ in $\mathcal{W}_2$	by inspection of rules for $\mathcal{W}_1 \leq \mathcal{W}_2$
$(\Omega, \boldsymbol{A}) \in \mathcal{W}_2$	by rules utilizing includesOther and includesAny

□

Lemma B.13.3 (Transitivity of World Subsumption)**.**
If $\mathcal{W}_1 \leq \mathcal{W}_2$ *and* $\mathcal{W}_2 \leq \mathcal{W}_3$ *and* $\mathcal{W}_3$ world*, then* $\mathcal{W}_1 \leq \mathcal{W}_3$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \mathcal{W}_1 \leq \mathcal{W}_2$ and $\mathcal{F} :: \mathcal{W}_2 \leq \mathcal{W}_3$.

Case: $\mathcal{E} = \dfrac{}{\cdot \leq \mathcal{W}_2}\,\mathsf{baseSubsume}$ and $\mathcal{F} = \mathcal{W}_2 \leq \mathcal{W}_3$

$\cdot \leq \mathcal{W}_3$	by baseSubsume

Case: $\mathcal{E} = \dfrac{}{\mathcal{W}_1 \leq *}\,\mathsf{anySubsume}$ and $\mathcal{F} = \dfrac{}{* \leq *}\,\mathsf{anySubsume}$

$\mathcal{W}_1 \leq *$	by $\mathcal{E}$

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \mathcal{W}_1 \leq \mathcal{W}_2 \qquad (\Omega, \boldsymbol{A}) \in \mathcal{W}_2}{\mathcal{W}_1, (\Omega, \boldsymbol{A}) \leq \mathcal{W}_2}\,\mathsf{containsSubsume}$ and $\mathcal{F} = \mathcal{W}_2 \leq \mathcal{W}_3$

$\mathcal{W}_3$ world	by assumption
$\mathcal{W}_1 \leq \mathcal{W}_3$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$
$(\Omega, \boldsymbol{A}) \in \mathcal{W}_2$	by assumption
$(\Omega, \boldsymbol{A}) \in \mathcal{W}_3$	by Lemma B.13.2 using $\mathcal{F}$
$\mathcal{W}_1, (\Omega, \boldsymbol{A}) \leq \mathcal{W}_3$	by containsSubsume

□

Lemma B.13.4 (Transitive Property of $\leq$ on Substitutions)**.**
If $\omega_1 \leq \omega_2$ *and* $\omega_2 \leq \omega_3$*, then* $\omega_1 \leq \omega_3$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \omega_1 \leq \omega_2$ and $\mathcal{F} :: \omega_2 \leq \omega_3$. Similar to Lemma B.10.2 except we also have rules to handle the $\uparrow_\alpha$. In Lemma B.10.2 we required significant work with leWorldAddNew because of the assumption $(\Omega', \boldsymbol{A}) \in \mathcal{W}$ that was attached to it. We have no such assumptions in these rules. The proof is straightforward just calling the i.h. and putting it back together.

□

Lemma B.13.5 ($\cdot$ is the Smallest Substitution)**.**
$\cdot \leq \omega$*.*

Proof. Straightforward proof by induction on ω. □

Lemma B.13.6 (Substitution Preserves Signature Extensions)**.**
If $\mathcal{W}$ world *and* $\cdot \leq_{\mathcal{W}} \Omega$ *and* Ω ctx *and* $\Omega' \vdash \omega : \Omega$*, then* $\cdot \leq_{\mathcal{W}} \Omega'$*.*

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$.

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\,\mathsf{tpSubBase}$

$\cdot \leq_{\mathcal{W}} \cdot$	by assumption

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\,\mathsf{tpSubInd}$

$\mathcal{W}$ world	by assumption
$\cdot \leq_{\mathcal{W}} (\Omega, \alpha{\in}\delta)$	by assumption
$(\Omega, \alpha{\in}\delta)$ ctx	by assumption
Ω ctx	by inversion using ctxAdd
$\cdot \leq_{\mathcal{W}} \Omega$	by inversion using leWorldAdd
$\cdot \leq_{\mathcal{W}} \Omega'$	by i.h. on $\mathcal{E}_1$

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 \quad \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}}{(\Omega', \boldsymbol{x'}^{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x}^{\triangledown}{\in}\boldsymbol{A}^{\#})}\,\mathsf{tpSubIndNew}$

$\mathcal{W}$ world	by assumption
$\cdot \leq_{\mathcal{W}} (\Omega, \boldsymbol{x}^{\triangledown}{\in}\boldsymbol{A}^{\#})$	by assumption
$(\Omega, \boldsymbol{x}^{\triangledown}{\in}\boldsymbol{A}^{\#})$ ctx	by assumption
Ω ctx	by inversion using ctxAddNew
$\cdot \leq_{\mathcal{W}} \Omega$	by inversion using leWorldAddNew
$(\Omega, \boldsymbol{A}) \in \mathcal{W}$	by inversion using leWorldAddNew
$(\Omega, \boldsymbol{A}) \in_{\mathrm{alt}} \mathcal{W}$	by Lemma B.9.2
$\lVert\Omega'\rVert^{\nabla} \vdash \lVert\omega\rVert^{\nabla} : \lVert\Omega\rVert^{\nabla}$	by Lemma B.4.8 on $\mathcal{E}_1$
$(\Omega', \boldsymbol{A}[\lVert\omega\rVert^{\nabla}]) \in_{\mathrm{alt}} \mathcal{W}$	by Lemma B.12.5
$\boldsymbol{A}[\omega] = \boldsymbol{A}[\lVert\omega\rVert] = \boldsymbol{A}[\lVert\lVert\omega\rVert^{\nabla}\rVert] = \boldsymbol{A}[\lVert\omega\rVert^{\nabla}]$	by Subst. Application (Def. A.2.4) and Lemma B.4.2
$(\Omega', \boldsymbol{A}[\omega]) \in_{\mathrm{alt}} \mathcal{W}$	by above
$(\Omega', \boldsymbol{A}[\omega]) \in \mathcal{W}$	by Lemma B.9.1
$\cdot \leq_{\mathcal{W}} \Omega'$	by i.h. on $\mathcal{E}_1$
$\cdot \leq_{\mathcal{W}} \Omega', \boldsymbol{x'}^{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega]$	by leWorldAddNew and Subst. App (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}_2 \text{ world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}_2) \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}_2)}\ \mathsf{tpSubWorld}$

$\mathcal{W}$ world	by assumption
$\cdot \leq_{\mathcal{W}} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}_2)$	by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}_2)$ ctx	by assumption
Ω ctx	by inversion using ctxAddWorld
$\cdot \leq_{\mathcal{W}} \Omega$	by inversion using leWorldAddMarker
$\mathcal{W}_2 \leq \mathcal{W}$	by inversion using leWorldAddMarker
$\cdot \leq_{\mathcal{W}} \Omega'$	by i.h. on $\mathcal{E}_1$
$\cdot \leq_{\mathcal{W}} (\Omega', u'\overset{\triangledown}{\in}\mathcal{W}_2)$	by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta \text{ wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_{\alpha} \omega : \Omega}\ \mathsf{tpSubShift}$

$\mathcal{W}$ world	by assumption
$\cdot \leq_{\mathcal{W}} \Omega$	by assumption
Ω ctx	by assumption
$\cdot \leq_{\mathcal{W}} \Omega'$	by i.h. on $\mathcal{E}_1$
$\cdot \leq_{\mathcal{W}} (\Omega', \alpha{\in}\delta)$	by leWorldAdd

□

Lemma B.13.7 (Weakening $\leq_{\mathcal{W}}$ w.r.t. World Subsumption)**.**
If $\Omega \leq_{\mathcal{W}_1} \Omega'$ *and* $\mathcal{W}_1 \leq \mathcal{W}_2$ *and* $\mathcal{W}_2$ world*, then* $\Omega \leq_{\mathcal{W}_2} \Omega'$*.*

Proof. By induction on $\mathcal{E} :: \Omega \leq_{\mathcal{W}_1} \Omega'$.

Case: $\mathcal{E} = \dfrac{}{\Omega \leq_{\mathcal{W}_1} \Omega}\,\mathsf{leWorldEq}$

$\Omega \leq_{\mathcal{W}_2} \Omega$	by leWorldEq

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}_1} \Omega'\end{array}}{\Omega \leq_{\mathcal{W}_1} \Omega', \alpha_1 \in \delta_1}\,\mathsf{leWorldAdd}$

$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2$ world	by assumption
$\Omega \leq_{\mathcal{W}_2} \Omega'$	by i.h. on $\mathcal{E}_1$
$\Omega \leq_{\mathcal{W}_2} \Omega', \alpha \in \delta_1$	by leWorldAdd

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}_1} \Omega' \qquad (\Omega', \boldsymbol{A}) \in \mathcal{W}_1\end{array}}{\Omega \leq_{\mathcal{W}_1} \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\,\mathsf{leWorldAddNew}$

$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2$ world	by assumption
$\Omega \leq_{\mathcal{W}_2} \Omega'$	by i.h. on $\mathcal{E}_1$
$(\Omega', \boldsymbol{A}) \in \mathcal{W}_1$	by assumption (premise)
$(\Omega', \boldsymbol{A}) \in \mathcal{W}_2$	by Lemma B.13.2
$\Omega \leq_{\mathcal{W}_2} \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}_1} \Omega' \qquad \mathcal{W}' \leq \mathcal{W}_1\end{array}}{\Omega \leq_{\mathcal{W}_1} \Omega', u \overset{\triangledown}{\in} \mathcal{W}'}\,\mathsf{leWorldAddMarker}$

$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2$ world	by assumption
$\Omega \leq_{\mathcal{W}_2} \Omega'$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}' \leq \mathcal{W}_1$	by assumption (premise)
$\mathcal{W}' \leq \mathcal{W}_2$	by Lemma B.13.3
$\Omega \leq_{\mathcal{W}_2} \Omega', u \overset{\triangledown}{\in} \mathcal{W}'$	by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}_1} \Omega'\end{array}}{\Omega, D \leq_{\mathcal{W}_1} \Omega', D}\,\mathsf{leWorldMiddle}$

$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$\mathcal{W}_2$ world	by assumption
$\Omega \leq_{\mathcal{W}_2} \Omega'$	by i.h. on $\mathcal{E}_1$
$\Omega, D \leq_{\mathcal{W}_2} \Omega', D$	by leWorldMiddle

□

Lemma B.13.8 (Context Weakening Misc. Property)**.**
If $\Omega_1 \leq_{\mathcal{W}} \Omega_1'$*, then* $\Omega_1, \Omega_2 \leq_{\mathcal{W}} \Omega_1', \Omega_2$*.*

Proof. By induction on Ω_2.

Case: $\cdot$

$\Omega_1 \leq_{\mathcal{W}} \Omega_1'$	by assumption

Case: Ω_2, D

$\Omega_1 \leq_{\mathcal{W}} \Omega_1'$	by assumption
$\Omega_1, \Omega_2 \leq_{\mathcal{W}} \Omega_1', \Omega_2$	by i.h. on Ω_2
$\Omega_1, \Omega_2, D \leq_{\mathcal{W}} \Omega_1', \Omega_2, D$	by leWorldMiddle

□

Lemma B.13.9 (Context Weakening Inversion (Version 1))**.**
If $\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$ *and* $\Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega'$ *and* Ω' ctx *and* $\mathcal{W}$ world*,*
then $\Omega' = \Omega'_1, D, \Omega'_2$ *and* $\Omega_1 \leq_{\mathcal{W}} \Omega'_1$ *and* $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$*.*

Proof. By induction on $\mathcal{E} :: \Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega'$.

Case: $\mathcal{E} = \dfrac{}{\Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega_1, D, \Omega_2}$ leWorldEq

$\Omega_1 \leq_{\mathcal{W}} \Omega_1$	by leWorldEq
$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$	by assumption

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega'\end{array}}{\Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega', \alpha_1 \in \delta_1}$ leWorldAdd

$\mathcal{W}$ world	by assumption
$(\Omega', \alpha_1 \in \delta_1)$ ctx	by assumption
$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$	by assumption
Ω' ctx	by inversion using ctxAdd
$\Omega' = \Omega'_1, D, \Omega'_2$	
and $\Omega_1 \leq_{\mathcal{W}} \Omega'_1$	
and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, \alpha_1 \in \delta_1$	by leWorldAdd

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega' \qquad (\Omega', \boldsymbol{A}) \in \mathcal{W}\end{array}}{\Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}$ leWorldAddNew

$\mathcal{W}$ world	by assumption
$(\Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ ctx	by assumption
$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$	by assumption
Ω' ctx	by inversion using ctxAddNew
$\Omega' = \Omega'_1, D, \Omega'_2$	
and $\Omega_1 \leq_{\mathcal{W}} \Omega'_1$	
and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$((\Omega'_1, D, \Omega'_2), \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega' \qquad \mathcal{W}' \leq \mathcal{W}\end{array}}{\Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega', u{\in}^{\triangledown}\mathcal{W}'}\ \mathsf{leWorldAddMarker}$

$\mathcal{W}$ world	by assumption
$(\Omega', u{\in}^{\triangledown}\mathcal{W}')$ ctx	by assumption
$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$	by assumption
Ω' ctx	by inversion using ctxAddNew
$\Omega' = \Omega'_1, D, \Omega'_2$	
and $\Omega_1 \leq_{\mathcal{W}} \Omega'_1$	
and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}' \leq \mathcal{W}$	by assumption
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, u{\in}^{\triangledown}\mathcal{W}$	by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega'\end{array}}{\Omega_1, D, \Omega_2, \alpha_1{\in}\delta_1 \leq_{\mathcal{W}} \Omega', \alpha_1{\in}\delta_1}\ \mathsf{leWorldMiddle}$

and $\dfrac{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2}{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2, \alpha_1{\in}\delta_1}\ \mathsf{leWorldAdd}$

$\mathcal{W}$ world	by assumption
$(\Omega', \alpha_1{\in}\delta_1)$ ctx	by assumption
Ω' ctx	by inversion using ctxAdd
$\Omega' = \Omega'_1, D, \Omega'_2$	
and $\Omega_1 \leq_{\mathcal{W}} \Omega'_1$	
and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, \alpha_1{\in}\delta_1$	by leWorldAdd

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega'\end{array}}{\Omega_1, D, \Omega_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \leq_{\mathcal{W}} \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\,\mathsf{leWorldMiddle}$

and $\dfrac{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2 \quad ((\Omega_1, D, \Omega_2), \boldsymbol{A}) \in \mathcal{W}}{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\,\mathsf{leWorldAddNew}$

$\mathcal{W}$ world	by assumption
$(\Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ ctx	by assumption
Ω' ctx	by inversion using ctxAddNew
$\Omega' = \Omega'_1, D, \Omega'_2$ and $\Omega_1 \leq_{\mathcal{W}} \Omega'_1$ and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$((\Omega_1, D, \Omega_2), \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\Omega_1, D, \Omega_2 \leq_* \Omega'_1, D, \Omega'_2$	by Lemma B.7.7
$((\Omega'_1, D, \Omega'_2), \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.10.1
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq_{\mathcal{W}} \Omega'\end{array}}{\Omega_1, D, \Omega_2, u \overset{\triangledown}{\in} \mathcal{W}' \leq_{\mathcal{W}} \Omega', u \overset{\triangledown}{\in} \mathcal{W}'}\,\mathsf{leWorldMiddle}$

and $\dfrac{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2 \quad \mathcal{W}' \leq \mathcal{W}}{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2, u \overset{\triangledown}{\in} \mathcal{W}'}\,\mathsf{leWorldAddMarker}$

$\mathcal{W}$ world	by assumption
$(\Omega', u \overset{\triangledown}{\in} \mathcal{W}')$ ctx	by assumption
Ω' ctx	by inversion using ctxAddNew
$\Omega' = \Omega'_1, D, \Omega'_2$ and $\Omega_1 \leq_{\mathcal{W}} \Omega'_1$ and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}' \leq \mathcal{W}$	by assumption
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, u \overset{\triangledown}{\in} \mathcal{W}'$	by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \leq_{\mathcal{W}} \Omega'\end{array}}{\Omega_1, D \leq_{\mathcal{W}} \Omega', D}\ \mathsf{leWorldMiddle}$

$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D$	by assumption
$\mathcal{W}$ world	by assumption
(Ω', D) ctx	by assumption
$\Omega_1 \leq_{\mathcal{W}} \Omega'$	by assumption
$\Omega', D \leq_{\mathcal{W}} \Omega', D$	by leWorldEq

□

Lemma B.13.10 (Context Weakening Inversion (Version 2))**.**
If $\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$ *and* $\Omega_1, D, \Omega_2 \leq \Omega'$ *and* Ω' ctx *and* $\mathcal{W}$ world*,*
then $\Omega' = \Omega'_1, D, \Omega'_2$ *and* $\Omega_1 \leq \Omega'_1$ *and* $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$*.*

Proof. By induction on $\mathcal{E} :: \Omega_1, D, \Omega_2 \leq \Omega'$.

Case: $\mathcal{E} = \dfrac{}{\Omega_1, D, \Omega_2 \leq \Omega_1, D, \Omega_2}\,\mathsf{leEq}$

$\Omega_1 \leq \Omega_1$	by leEq
$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$	by assumption

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq \Omega'\end{array}}{\Omega_1, D, \Omega_2 \leq \Omega', \alpha_1{\in}\delta_1}\,\mathsf{leAdd}$

$\mathcal{W}$ world	by assumption
$(\Omega', \alpha_1{\in}\delta_1)$ ctx	by assumption
$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2$	by assumption
Ω' ctx	by inversion using ctxAdd
$\Omega' = \Omega'_1, D, \Omega'_2$ and $\Omega_1 \leq \Omega'_1$ and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, \alpha_1{\in}\delta_1$	by $\mathsf{leWorldAdd}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq \Omega'\end{array}}{\Omega_1, D, \Omega_2, \alpha_1{\in}\delta_1 \leq \Omega', \alpha_1{\in}\delta_1}\,\mathsf{leMiddle}$

and $\dfrac{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2}{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2, \alpha_1{\in}\delta_1}\,\mathsf{leWorldAdd}$

$\mathcal{W}$ world	by assumption
$(\Omega', \alpha_1{\in}\delta_1)$ ctx	by assumption
Ω' ctx	by inversion using ctxAdd
$\Omega' = \Omega'_1, D, \Omega'_2$ and $\Omega_1 \leq \Omega'_1$ and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, \alpha_1{\in}\delta_1$	by $\mathsf{leWorldAdd}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq \Omega'\end{array}}{\Omega_1, D, \Omega_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \leq \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\ \mathsf{leMiddle}$

and $\dfrac{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2 \quad ((\Omega_1, D, \Omega_2), \boldsymbol{A}) \in \mathcal{W}}{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\ \mathsf{leWorldAddNew}$

$\mathcal{W}$ world	by assumption
$(\Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ ctx	by assumption
Ω' ctx	by inversion using ctxAddNew
$\Omega' = \Omega'_1, D, \Omega'_2$ and $\Omega_1 \leq \Omega'_1$ and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$((\Omega_1, D, \Omega_2), \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\Omega_1, D, \Omega_2 \leq_* \Omega'_1, D, \Omega'_2$	by Lemma B.7.7
$((\Omega'_1, D, \Omega'_2), \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.10.1
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1, D, \Omega_2 \leq \Omega'\end{array}}{\Omega_1, D, \Omega_2, u \overset{\triangledown}{\in} \mathcal{W}' \leq \Omega', u \overset{\triangledown}{\in} \mathcal{W}'}\ \mathsf{leMiddle}$

and $\dfrac{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2 \quad \mathcal{W}' \leq \mathcal{W}}{\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D, \Omega_2, u \overset{\triangledown}{\in} \mathcal{W}'}\ \mathsf{leWorldAddMarker}$

$\mathcal{W}$ world	by assumption
$(\Omega', u \overset{\triangledown}{\in} \mathcal{W}')$ ctx	by assumption
Ω' ctx	by inversion using ctxAddNew
$\Omega' = \Omega'_1, D, \Omega'_2$ and $\Omega_1 \leq \Omega'_1$ and $\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}' \leq \mathcal{W}$	by assumption
$\Omega'_1, D \leq_{\mathcal{W}} \Omega'_1, D, \Omega'_2, u \overset{\triangledown}{\in} \mathcal{W}'$	by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \leq \Omega'\end{array}}{\Omega_1, D \leq \Omega', D}\,\mathsf{leMiddle}$

$\Omega_1, D \leq_{\mathcal{W}} \Omega_1, D$	by assumption
$\mathcal{W}$ world	by assumption
(Ω', D) ctx	by assumption
$\Omega_1 \leq \Omega'$	by assumption
$\Omega', D \leq_{\mathcal{W}} \Omega', D$	by leWorldEq

□

Lemma B.13.11 (Context Weakening Inversion (Version 3))**.**
If $\Omega_1, D \leq \Omega_1, D, \Omega_2$ *and* $\Omega_1, D, \Omega_2 \leq \Omega'$ *and* Ω' ctx *and* $\mathcal{W}$ world*,*
then $\Omega' = \Omega'_1, D, \Omega'_2$ *and* $\Omega_1 \leq \Omega'_1$ *and* $\Omega'_1, D \leq \Omega'_1, D, \Omega'_2$*.*

Proof. By induction on $\mathcal{E} :: \Omega_1, D, \Omega_2 \leq \Omega'$. This proceeds the same way as Lemma B.13.10. □

Lemma B.13.12 (Break Apart $\leq_{\mathcal{W}}$)**.**
If $\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$*, then* $\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$ *and* $\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$*.*

Proof. By induction on Ω_3.

Case: $\cdot$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by assumption
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2$	by leWorldEq

Case: $\Omega_3, \alpha_1 \in \delta_1$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \alpha_1 \in \delta_1$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by inversion using leWorldAdd
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \alpha_1 \in \delta_1$	by leWorldAdd

Case: $\Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by inversion using leWorldAddNew
$((\Omega, \Omega_2, \Omega_3), \boldsymbol{A}) \in \mathcal{W}$	by inversion using leWorldAddNew
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\Omega_3, u \overset{\triangledown}{\in} \mathcal{W}'$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, u \overset{\triangledown}{\in} \mathcal{W}'$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by inversion using leWorldAddMarker
$\mathcal{W}' \leq \mathcal{W}$	by inversion using leWorldAddMarker
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, u \overset{\triangledown}{\in} \mathcal{W}'$	by leWorldAddMarker

□

Lemma B.13.13 (Break Apart $\leq$)**.**
If $\Omega \leq \Omega, \Omega_2, \Omega_3$*, then* $\Omega \leq \Omega, \Omega_2$ *and* $\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3$*.*

Proof. By induction on Ω_3.

Case: $\cdot$

$\Omega \leq \Omega, \Omega_2$	by assumption
$\Omega, \Omega_2 \leq \Omega, \Omega_2$	by leWorldEq

Case: $\Omega_3, \alpha_1 \in \delta_1$

$\Omega \leq \Omega, \Omega_2, \Omega_3, \alpha_1 \in \delta_1$	by assumption
$\Omega \leq \Omega, \Omega_2, \Omega_3$	by inversion using leAdd
$\Omega \leq \Omega, \Omega_2$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3, \alpha_1 \in \delta_1$	by leAdd

Case: $\Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$

$\Omega \leq \Omega, \Omega_2, \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by assumption

This is impossible by inspection of the rules, so this case is vacuously true.

Case: $\Omega_3, u \overset{\triangledown}{\in} \mathcal{W}'$

$\Omega \leq \Omega, \Omega_2, \Omega_3, u \overset{\triangledown}{\in} \mathcal{W}'$	by assumption

This is impossible by inspection of the rules, so this case is vacuously true.

□

Lemma B.13.14 (Context Weakening Property on Left with Worlds)**.** *If* $\Omega \leq_* \Omega'$ *and* $\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$ *and* $\mathcal{W}$ world *and* $(\Omega', \Omega_2, \Omega_3)$ ctx*, then* $\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2, \Omega_3$*.*

Proof. By induction on Ω_3.

Case: $\cdot$

$\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2$	by leWorldEq

Case: $\Omega_3, \alpha{\in}\delta$

$\mathcal{W}$ world	by assumption
$(\Omega', \Omega_2, \Omega_3, \alpha{\in}\delta)$ ctx	by assumption
$(\Omega', \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_* \Omega'$	by assumption
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \alpha{\in}\delta$	by assumption
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by inversion using leWorldAdd
$\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2, \Omega_3, \alpha{\in}\delta$	by leWorldAdd

Case: $\Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$

$\mathcal{W}$ world	by assumption
$(\Omega', \Omega_2, \Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ ctx	by assumption
$(\Omega', \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_* \Omega'$	by assumption
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by assumption
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by inversion using leWorldAddNew
$((\Omega, \Omega_2, \Omega_3), \boldsymbol{A}) \in \mathcal{W}$	by inversion using leWorldAddNew
$\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega, \Omega_2, \Omega_3 \leq_* \Omega', \Omega_2, \Omega_3$	by Weakening Rules (Lemma B.13.8)
$((\Omega', \Omega_2, \Omega_3), \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.10.1
$\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2, \Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$

$\mathcal{W}$ world	by assumption
$(\Omega', \Omega_2, \Omega_3, u\overset{\triangledown}{\in}\mathcal{W}')$ ctx	by assumption
$(\Omega', \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_* \Omega'$	by assumption
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$	by assumption
$\Omega, \Omega_2 \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by inversion using leWorldAddMarker
$\mathcal{W}' \leq \mathcal{W}$	by inversion using leWorldAddMarker
$\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega', \Omega_2 \leq_{\mathcal{W}} \Omega', \Omega_2, \Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$	by leWorldAddMarker

□

Lemma B.13.15 (Context Weakening Property on Left without Worlds)**.**
If $\Omega \leq_* \Omega'$ *and* $\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3$ *and* $(\Omega', \Omega_2, \Omega_3)$ ctx*,*
then $\Omega', \Omega_2 \leq \Omega', \Omega_2, \Omega_3$*.*

Proof. By induction on Ω_3.

Case: $\cdot$

$\Omega', \Omega_2 \leq \Omega', \Omega_2$	by leEq

Case: $\Omega_3, \alpha{\in}\delta$

$(\Omega', \Omega_2, \Omega_3, \alpha{\in}\delta)$ ctx	by assumption
$(\Omega', \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_* \Omega'$	by assumption
$\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3, \alpha{\in}\delta$	by assumption
$\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3$	by inversion using leAdd
$\Omega', \Omega_2 \leq \Omega', \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega', \Omega_2 \leq \Omega', \Omega_2, \Omega_3, \alpha{\in}\delta$	by leAdd

Case: $\Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$

$\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by assumption

This is impossible by inspection of the rules, so this case is vacuously true.

Case: $\Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$

$\Omega, \Omega_2 \leq \Omega, \Omega_2, \Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$	by assumption

This is impossible by inspection of the rules, so this case is vacuously true.

□

Lemma B.13.16 (Context Weakening Property Combining Contexts)**.**
If $\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$ *and* $\Omega \leq_{\mathcal{W}} \Omega, \Omega_3$ *and* $\mathcal{W}$ world *and* $(\Omega, \Omega_2, \Omega_3)$ ctx*,*
then $\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$*.*

Proof. By induction on Ω_3.

Case: $\cdot$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by assumption

Case: $\Omega_3, \alpha_1{\in}\delta_1$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_3, \alpha_1{\in}\delta_1$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_3$	by inversion using leWorldAdd
$\mathcal{W}$ world	by assumption
$(\Omega, \Omega_2, \Omega_3, \alpha_1{\in}\delta_1)$ ctx	by assumption
$(\Omega, \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \alpha{\in}\delta_1$	by leWorldAdd

Case: $\Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_3$	by inversion using leWorldAddNew
$((\Omega, \Omega_3), \boldsymbol{A}) \in \mathcal{W}$	by inversion using leWorldAddNew
$\mathcal{W}$ world	by assumption
$(\Omega, \Omega_2, \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ ctx	by assumption
$(\Omega, \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by i.h. on Ω_3
$(\Omega, \Omega_3) \leq_{\mathcal{W}} (\Omega, \Omega_2, \Omega_3)$	by Lemma B.13.8
$(\Omega, \Omega_3) \leq_{*} (\Omega, \Omega_2, \Omega_3)$	by Lemma B.7.7
$((\Omega, \Omega_2, \Omega_3), \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.10.1
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$

$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$	by assumption
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_3$	by inversion using leWorldAddMarker
$\mathcal{W}' \leq \mathcal{W}$	by inversion using leWorldAddMarker
$\mathcal{W}$ world	by assumption
$(\Omega, \Omega_2, \Omega_3, u\overset{\triangledown}{\in}\mathcal{W}')$ ctx	by assumption
$(\Omega, \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3$	by i.h. on Ω_3
$\Omega \leq_{\mathcal{W}} \Omega, \Omega_2, \Omega_3, u\overset{\triangledown}{\in}\mathcal{W}'$	by leWorldAddMarker

□

B.14 Meta-Theory: Main Weakening Lemmas

Lemma B.14.1 (Context Weakening over Global Parameter Coverage)**.**
If $\mathcal{W} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$
and $\mathcal{W}$ world
and $\Omega \leq_{\mathcal{W}} \Omega'$
and Ω' ctx
and $\Omega \gg^{world} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$,

then $\Omega' \gg^{world} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$.

Proof. By induction on $\mathcal{E} :: \Omega \leq_{\mathcal{W}} \Omega'$.

Case: $\mathcal{E} = \dfrac{}{\Omega \leq_{\mathcal{W}} \Omega}$ leWorldEq

$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1}{\Omega \leq_{\mathcal{W}} \Omega', \alpha_1{\in}\delta_1}$ leWorldAdd

$\mathcal{W} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega', \alpha_1{\in}\delta_1)$ ctx	by assumption
$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption
Ω' ctx	by inversion using ctxAdd
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega', \alpha_1{\in}\delta_1 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by gcSkipNonParameter

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}} \Omega' \qquad (\Omega', \boldsymbol{A}) \in \mathcal{W}\end{array}}{\Omega \leq_{\mathcal{W}} \Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}}\ \mathsf{leWorldAddNew}$

$\mathcal{W} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ ctx	by assumption
$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption
Ω' ctx	by inversion using ctxAddNew
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by i.h. on $\mathcal{E}_1$
$(\Omega', \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\Omega', \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by gcSkipWorld

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}} \Omega' \qquad \mathcal{W}' \leq \mathcal{W}\end{array}}{\Omega \leq_{\mathcal{W}} \Omega', u \overset{\triangledown}{\in} \mathcal{W}'}\ \mathsf{leWorldAddMarker}$

$\mathcal{W} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega', u \overset{\triangledown}{\in} \mathcal{W}')$ ctx	by assumption
$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption
Ω' ctx	by inversion using ctxAddWorld
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by i.h. on $\mathcal{E}_1$
$\mathcal{W}' \leq \mathcal{W}$	by assumption
$\Omega', u \overset{\triangledown}{\in} \mathcal{W}' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by gcCheckWorld

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \leq_{\mathcal{W}} \Omega'\end{array}}{\Omega, D \leq_{\mathcal{W}} \Omega', D}\ \mathsf{leWorldMiddle}$

$\mathcal{W} \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption
$\mathcal{W}$ world	by assumption
(Ω', D) ctx	by assumption
$\Omega, D \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by assumption

Subcase: Inversion using gcSkipMismatch

$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by inversion
$D = \boldsymbol{y} \overset{\triangledown}{\in} (\boldsymbol{\Pi\Gamma_c}.\ \boldsymbol{B_c})^{\#}$	by inversion
$(\boldsymbol{B_x} \approx \boldsymbol{B_c})$ do not unify	by inversion
Ω' ctx	by inversion using ctxAddNew
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega', D \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by gcSkipMismatch

Subcase: Inversion using gcSkipNonParameter

$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by inversion
$D = \alpha{\in}\delta$	by inversion
Ω' ctx	by inversion using ctxAdd
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega', D \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by gcSkipNonParameter

Subcase: Inversion using gcSkipWorld

$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by inversion
$D = \boldsymbol{y} \overset{\triangledown}{\in} (\boldsymbol{\Pi\Gamma_c}.\ \boldsymbol{B_c})^{\#}$	by inversion
$\mathcal{W}_2$ world	by inversion
$\mathcal{W}_2 \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by inversion
$(\Omega, (\boldsymbol{\Pi\Gamma_c}.\ \boldsymbol{B_c})) \in \mathcal{W}_2$	by inversion
$\Omega \leq_* \Omega'$	by Lemma B.7.7 on $\mathcal{E}_1$
$(\Omega', (\boldsymbol{\Pi\Gamma_c}.\ \boldsymbol{B_c})) \in \mathcal{W}_2$	by Lemma B.10.1
Ω' ctx	by inversion using ctxAddNew
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega', D \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$	by gcSkipWorld

Subcase: Inversion using gcCheckWorld

$\Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$	by inversion
$D = u{\overset{\triangledown}{\in}}\mathcal{W}_2$	by inversion
$\mathcal{W}_2 \leq \mathcal{W}_3$	by inversion
$\mathcal{W}_3$ world	by inversion
$\mathcal{W}_3 \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$	by inversion
Ω' ctx	by inversion using ctxAddWorld
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega', D \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$	by gcCheckWorld

□

Lemma B.14.2 (Context Weakening over Coverage without Worlds)**.**
If $\Omega \vdash \overline{c}$ covers τ *and* $\Omega \leq \Omega'$*, and* Ω' ctx*,*
then $\Omega' \vdash \overline{c}$ covers τ*.*

Proof. By induction on $\mathcal{E} : \Omega \vdash \overline{c}$ covers τ. *Note:* Most of these cases are trivial. The interesting ones are only coverPop, coverLF, and coverEmpty.

Case: $\mathcal{E} = \dfrac{c = \epsilon\alpha{\in}\delta.\ \alpha \mapsto f}{\Omega \vdash c \text{ covers } \forall\alpha{\in}\delta.\ \tau}\ \mathsf{coverSimple}$

$c = \epsilon\alpha{\in}\delta.\ \alpha \mapsto f$	by assumption (premise)
$\Omega' \vdash c$ covers $\forall\alpha{\in}\delta.\ \tau$	by coverSimple

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau\ \mathsf{wff} \\ c = \epsilon\boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y\ \Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y\ \Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}$

$\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau$ wff	by assumption (premise)
$c = \epsilon\boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y\ \Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y\ \Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f$	by assumption (premise)
$\Omega' \vdash c$ covers $(\nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma$	by coverPairLF

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \exists\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff} \\ c = \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\exists\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}^{\#}$

$\cdot \vdash \exists\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$ wff	by assumption (premise)
$c = \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto f$	by assumption (premise)
$\Omega' \vdash c$ covers $(\exists\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma$	by coverPairLF$^{\#}$

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)\ \mathsf{wff} \\ c = \epsilon u_1{\in}(\nabla\boldsymbol{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla\boldsymbol{\Gamma}.\ \tau_2).\ (\nu\boldsymbol{\Gamma}.\ (u_1\backslash\boldsymbol{\Gamma},\ u_2\backslash\boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\nabla\boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma}\ \mathsf{coverPairMeta}$

$\cdot \vdash \nabla\boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)\ \mathsf{wff}$ by assumption (premise)
$c = \epsilon u_1{\in}(\nabla\boldsymbol{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla\boldsymbol{\Gamma}.\ \tau_2).\ (\nu\boldsymbol{\Gamma}.\ (u_1\backslash\boldsymbol{\Gamma},\ u_2\backslash\boldsymbol{\Gamma})) \mapsto f$ by assumption (premise)
$\Omega' \vdash c \text{ covers } (\nabla\boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma$ by coverPairMeta

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{E}_1 \\ \Omega_1 \vdash \overline{c} \text{ covers } \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \qquad (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c}\backslash\boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$

$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2) \leq \Omega'$ by assumption
$\Omega'\ \mathsf{ctx}$ by assumption
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \leq \Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2$ by assumption (premise)
$\cdot\ \mathsf{world}$ by worldEmpty
$\Omega' = \Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega'_2$
and $\Omega_1 \leq \Omega'_1$
and $\Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \leq \Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega'_2$ by Lemma B.13.11
$\Omega'_1\ \mathsf{ctx}$ by inversion
$\Omega'_1 \vdash \overline{c} \text{ covers } \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau$ By i.h. on $\mathcal{E}_1$
$\Omega' \vdash \overline{c}\backslash\boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$ by coverPop

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}\mathcal{F}_1 :: (\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})\ \mathsf{ctx}\\ \mathcal{F}_2 :: \Omega_A \text{ only contains declarations of type } \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#}\\ \mathcal{F}_3 :: \boldsymbol{\Sigma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau\\ \mathcal{F}_4 :: \boldsymbol{\Gamma_x} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau\\ \mathcal{F}_5 :: \Omega \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau\end{array}}{\Omega \vdash \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau}\ \mathsf{coverLF}$$

$\Omega \leq \Omega'$ by assumption
$\Omega'\ \mathsf{ctx}$ by assumption
$\cdot \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by $\mathsf{wcEmpty}$
$\cdot\ \mathsf{world}$ by $\mathsf{worldEmpty}$
$\mathcal{F}_5' :: \Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by Lemma B.14.1 on $\mathcal{F}_5$
$\Omega' \vdash \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by $\mathsf{coverLF}$ using $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}_5'$

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}\mathcal{F}_1 :: (\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}\\ \mathcal{F}_2 :: \text{All elements of } \Omega_A \text{ occur free in } \boldsymbol{A}\\ \mathcal{F}_3 :: g = \epsilon\Omega_A.\ \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \nu\boldsymbol{\Gamma_g}.\ (\Omega_A; \boldsymbol{x} \mapsto f)\\ \mathcal{F}_4 :: g \text{ in } \overline{c}\\ \mathcal{F}_5 :: \boldsymbol{\Gamma} \gg^{\nabla} \overline{c} \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)\end{array}}{\Omega \vdash \overline{c} \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverNewLF}^{\#}$$

$\Omega' \vdash \overline{c}$ covers $\nabla\boldsymbol{\Gamma}.\ (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)$ by $\mathsf{coverNewLF}^{\#}$ with the same premises ($\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}_5$)

$$\textbf{Case: } \mathcal{E} = \frac{\cdot \leq_{\cdot} \Omega}{\Omega \vdash \mathit{nil} \text{ covers } (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverEmpty}^{\#}$$

$\Omega \leq \Omega'$ by assumption
$\Omega \leq_{\cdot} \Omega'$ by Lemma B.7.2
$\Omega'\ \mathsf{ctx}$ by assumption
$\cdot \leq_{\cdot} \Omega$ by assumption (premise)
$\cdot \leq_{\cdot} \Omega'$ by Lemma B.10.2
$\Omega' \vdash \mathit{nil}$ covers $(\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)$ by $\mathsf{coverEmpty}$

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\boldsymbol{\Gamma} \Vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type} \\ c = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi \Gamma .}\ \boldsymbol{A}).\ \nu \boldsymbol{\Gamma}.\ ((\boldsymbol{x}\ \boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla \boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)}\ \mathsf{coverNewLF}$

$\boldsymbol{\Gamma} \Vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$	by assumption (premise)
$c = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi \Gamma .}\ \boldsymbol{A}).\ \nu \boldsymbol{\Gamma}.\ ((\boldsymbol{x}\ \boldsymbol{\Gamma}) \mapsto f)$	by assumption (premise)
$\Omega' \vdash c \text{ covers } \nabla \boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)$	by coverNewLF

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla \boldsymbol{\Gamma}.\ \sigma\ \mathsf{wff} \\ c = \epsilon u \in (\nabla \boldsymbol{\Gamma}.\ \sigma).\ \nu \boldsymbol{\Gamma}.\ ((u \backslash \boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla \boldsymbol{\Gamma}.\ (\sigma \supset \tau)}\ \mathsf{coverNewMeta}$

$\cdot \vdash \nabla \boldsymbol{\Gamma}.\ \sigma\ \mathsf{wff}$	by assumption (premise)
$c = \epsilon u \in (\nabla \boldsymbol{\Gamma}.\ \sigma).\ \nu \boldsymbol{\Gamma}.\ ((u \backslash \boldsymbol{\Gamma}) \mapsto f)$	by assumption (premise)
$\Omega' \vdash c \text{ covers } \nabla \boldsymbol{\Gamma}.\ (\sigma \supset \tau)$	by coverNewMeta

□

Lemma B.14.3 (Context Weakening over Coverage with Worlds)**.**
If $\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C \vdash \overline{c}$ covers τ
and $(\Omega_A, u{:}\nabla\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B)$
and $(\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C)$ ctx*,*
then $\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C \vdash \overline{c}$ covers τ*.*

Proof. By induction on $\mathcal{E} : \Omega \vdash \overline{c}$ covers τ. *Note:* Most of these cases are trivial. We only show the interesting cases, which are $\mathsf{coverPop}$, $\mathsf{coverLF}$, and $\mathsf{coverEmpty}$.

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_1 \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x'}{:}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_A, u{:}\nabla\mathcal{W}, \Omega_C)\end{array}}{\Omega_1, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_A, u{:}\nabla\mathcal{W}, \Omega_C \vdash \overline{c}\backslash\boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$$

$(\Omega_1, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_A, u{:}\nabla\mathcal{W}, \Omega_C)$ by assumption
This case is impossible by inversion.

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u{:}\nabla\mathcal{W}, \Omega_C \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x'}{:}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}) \leq (\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c}\backslash\boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$$

$(\Omega_A, u{:}\nabla\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B)$ by assumption
$(\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_2)$ ctx by assumption
$(\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C)$ ctx by inversion
$\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C \vdash \overline{c}$ covers $\nabla \boldsymbol{x'}{:}\boldsymbol{A}^{\#}.\ \tau$ by i.h. on $\mathcal{E}_1$
$(\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}) \leq (\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_2)$ by assumption
$(\Omega_A, u{:}\nabla\mathcal{W}) \leq_* (\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B)$ by Lemma B.7.7
$(\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}) \leq (\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_2)$
by Lemma B.13.15
$\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C \leq_{\mathcal{W}} \Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C$ by Lemma B.13.8
$\uparrow_{\boldsymbol{x}}(\mathrm{id}_{\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C}), \boldsymbol{x}/\boldsymbol{x'} \leq \uparrow_{\boldsymbol{x}}(\mathrm{id}_{\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C}), \boldsymbol{x}/\boldsymbol{x'}$
by Lemma B.4.5 and rules
$\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}] = \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]$ by Lemma B.7.4
$\Omega_A, u{:}\nabla\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}{:}\nabla\boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c}\backslash\boldsymbol{x}$ covers $\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u{:}\nabla\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]$
by $\mathsf{coverPop}$ and above

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}\mathcal{F}_1 :: (\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}})\ \mathsf{ctx}\\ \mathcal{F}_2 :: \Omega_X \text{ only contains declarations of type } \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#}\\ \mathcal{F}_3 :: \boldsymbol{\Sigma} \gg \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau\\ \mathcal{F}_4 :: \boldsymbol{\Gamma}_{\boldsymbol{x}} \gg \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau\\ \mathcal{F}_5 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau}\ \mathsf{coverLF}$$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by assumption

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$ by assumption

$\mathcal{W}\ \mathsf{world}$ by inversion as a context containing $u\overset{\triangledown}{\in}\mathcal{W}$ is well-formed

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ by Lemma B.13.8

There exists a $\mathcal{W}_2$ such that
- $\mathcal{W} \leq \mathcal{W}_2$
- and $\mathcal{W}_2\ \mathsf{world}$
- and $\mathcal{W}_2 \gg \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by inversion on $\mathcal{F}_5$ ($\mathsf{gcCheckWorld}$ on $(u\overset{\triangledown}{\in}\mathcal{W})$)

$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ by Lemma B.13.7

$\mathcal{F}_5' :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by Lemma B.14.1 on $\mathcal{F}_5$

$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by $\mathsf{coverLF}$ using $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}_5'$

Case: $\mathcal{E} = \dfrac{\cdot \leq_{\cdot} (\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_C)}{\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_C \vdash \mathit{nil} \text{ covers } (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverEmpty}^{\#}$

$(\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$\cdot \leq_{\cdot} (\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B)$	by assumption (premise)
$\mathcal{W} \leq \cdot$	by inversion
$\mathcal{W} = \cdot$	by Case analysis on the rules noting that it is impossible for $(\Omega_*, A_*) \in \cdot$
$\cdot\ \mathsf{world}$	by worldEmpty
$\cdot \leq_{\cdot} \Omega'$	by Lemma B.10.2
$\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B, \Omega_C \vdash \mathit{nil} \text{ covers } (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)$	by coverEmpty

□

Lemma B.14.4 (Context Weakening over Typing without Worlds)**.**

- *If* $\Omega \vdash e \in \delta$ *and* $\Omega \leq \Omega'$ *and* Ω' ctx*, then* $\Omega' \vdash e \in \delta$*.*
- *If* $\Omega \vdash c \in \tau$ *and* $\Omega \leq \Omega'$ *and* Ω' ctx*, then* $\Omega' \vdash c \in \tau$*.*
- *If* $\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$ *and* $\Omega \leq \Omega'$ *and* Ω' ctx*, then* $\Omega' \vdash \mathrm{id}_{\Omega'}, \overline{f}/\overline{\alpha} : \Omega', \overline{\alpha \in \delta}$*.*

Proof. By induction on e and c and $\overline{f}$.

Case: $()$ and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx}}{\Omega \vdash () \in \mathrm{unit}}\ \mathsf{top}$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$\Omega' \vdash () \in \mathrm{unit}$	by top

Case: u and $\mathcal{E} = \dfrac{(\Omega_1, u{\in}\tau, \Omega_2)\ \mathsf{ctx} \quad (\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)}{\Omega_1, u{\in}\tau, \Omega_2 \vdash u \in \tau}\ \tau\mathsf{var}$

$(\Omega_1, u{\in}\tau, \Omega_2) \leq \Omega'$	by assumption
Ω' ctx	by assumption
$(\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)$	by assumption
$\Omega' = \Omega'_1, u{\in}\tau, \Omega'_2$ and $\Omega_1 \leq \Omega'_1$ and $\Omega'_1, u{\in}\tau \leq \Omega'_1, u{\in}\tau, \Omega'_2$	by Lemma B.13.11
$\Omega'_1, u{\in}\tau, \Omega'_2 \vdash u \in \tau$	by $\tau\mathsf{var}$

Case: $\boldsymbol{x}$ and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad ((\boldsymbol{x}{\in}\boldsymbol{A}^\#) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#)) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^\#}\ \mathsf{var}^\#$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$((\boldsymbol{x}{\in}\boldsymbol{A}^\#) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#)) \text{ in } \Omega$	by assumption
$((\boldsymbol{x}{\in}\boldsymbol{A}^\#) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#)) \text{ in } \Omega'$	by inspection of weakening rules
$\Omega' \vdash \boldsymbol{x} \in \boldsymbol{A}^\#$	by $\mathsf{var}^\#$

Case: $\boldsymbol{M}$ and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad \|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$\Omega \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by assumption
$\|\Omega\| \leq_{\mathsf{lf}} \|\Omega'\|$	by Lemma B.4.4
$\|\Omega'\|\ \mathbf{ctx}_{\mathsf{lf}}$	by Lemma B.4.1
$\|\Omega'\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by Lemma B.1.9
$\Omega' \vdash \boldsymbol{M} \in \boldsymbol{A}$	by isLF

Case: $(\mathrm{fn}\ \overline{c})$ and

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1 \\ \Omega\ \mathsf{ctx} \quad \Omega \vdash \tau\ \mathsf{wff} \quad \text{for all}_{\,c_i \in \overline{c}}(\Omega \vdash c_i \in \tau) \quad \Omega \vdash \overline{c}\ \mathrm{covers}\ \tau\end{array}}{\Omega \vdash \mathrm{fn}\ \overline{c} \in \tau}\ \mathsf{impI}$$

$\Omega \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\Omega \vdash \tau\ \mathsf{wff}$	by assumption
$\Omega' \vdash \tau\ \mathsf{wff}$	by Lemma B.7.3
$\Omega \vdash \overline{c}\ \mathrm{covers}\ \tau$	by assumption
$\Omega' \vdash \overline{c}\ \mathrm{covers}\ \tau$	by Lemma B.14.2
$\text{for all}_{\,c_i \in \overline{c}}(\Omega' \vdash c_i \in \tau)$	by i.h. on c_i with $\mathcal{E}_1$ (for all $c_i \in \overline{c}$)
$\Omega' \vdash \mathrm{fn}\ \overline{c} \in \tau$	by impI

Case: $(e\ \overline{f})$ and $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash e \in \forall\overline{\alpha \in \delta}.\ \tau & \Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}\end{array}}{\Omega \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$

$\Omega \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\Omega \vdash e \in \forall\overline{\alpha \in \delta}.\ \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash \mathrm{id}_{\Omega'}, \overline{f}/\overline{\alpha} : \Omega', \overline{\alpha \in \delta}$	by i.h. on $\overline{f}$ with $\mathcal{E}_2$
$\Omega' \vdash e\ \overline{f} \in \tau[\mathrm{id}_{\Omega'}, \overline{f}/\overline{\alpha}]$	by impE
$\Omega \vdash \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]\ \mathsf{wff}$	by Lemma B.8.1 on $\mathcal{E}$
$\mathrm{id}_\Omega \leq \mathrm{id}_{\Omega'}$	by Lemma B.4.5
$\tau[\mathrm{id}_{\Omega'}, \overline{f}/\overline{\alpha}] = \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]$	by Lemma B.7.4
$\Omega' \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]$	by above

Case: $(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau\end{array}}{\Omega \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{new}$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$(\Omega, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#})$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \boldsymbol{A}^{\#}$ wff	by inversion using ctxAddNew
$\Omega' \vdash \boldsymbol{A}^{\#}$ wff	by Lemma B.7.3
$(\Omega', \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#})$ ctx	by ctxAddNew
$\Omega, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \leq \Omega', \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}$	by leMiddle
$\Omega', \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$	by new

Case: $(\nu u{\in}\mathcal{W}.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, u\overset{\nabla}{\in}\mathcal{W} \vdash e \in \tau\end{array}}{\Omega \vdash \nu u{\in}\mathcal{W}.\ e \in \nabla \mathcal{W}.\ \tau}\ \mathsf{newW}$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$(\Omega, u\overset{\nabla}{\in}\mathcal{W})$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\mathcal{W}$ world	by inversion using ctxAddWorld
$(\Omega', u\overset{\nabla}{\in}\mathcal{W})$ ctx	by ctxAddWorld
$\Omega, u\overset{\nabla}{\in}\mathcal{W} \leq \Omega', u\overset{\nabla}{\in}\mathcal{W}$	by leMiddle
$\Omega', u\overset{\nabla}{\in}\mathcal{W} \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash \nu u{\in}\mathcal{W}.\ e \in \nabla \mathcal{W}.\ \tau$	by newW

Case: $(e\backslash \boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2) \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$	by assumption
$\cdot\ \mathsf{world}$	by worldEmpty
$\Omega' = \Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega'_2$ and $\Omega_1 \leq \Omega'_1$ and $\Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \leq \Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega'_2$	by Lemma B.13.11
$\Omega'_1\ \mathsf{ctx}$	by inversion
$\Omega'_1 \vdash e \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega'_2 \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega'_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by pop
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]\ \mathsf{wff}$	by Lemma B.8.1 on $\mathcal{E}$
$\mathrm{id}_{\Omega_1} \leq \mathrm{id}_{\Omega'_1}$	by Lemma B.4.5
$\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'} \leq \uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega'_1}, \boldsymbol{x}/\boldsymbol{x'}$	by rules
$\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega'_1}, \boldsymbol{x}/\boldsymbol{x'}] = \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by Lemma B.7.4
$\Omega'_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega'_2 \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by above

Case: $(e\backslash u)$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ (\Omega_1, u{:}\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega_2)\end{array}}{\Omega_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \in \tau}\ \mathsf{popW}$

$(\Omega_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega_2) \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\mathcal{W} \leq \mathcal{W}_2$	by assumption
$(\Omega_1, u{:}\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega_2)$	by assumption
$\Omega_1 \vdash \nabla\mathcal{W}_2.\ \tau\ \mathsf{wff}$	by Lemma B.8.1
$\mathcal{W}_2\ \mathsf{world}$	by inversion using $\nabla_{\text{world}}\mathsf{wff}$
$\Omega' = \Omega'_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega'_2$ and $\Omega_1 \leq \Omega'_1$ and $\Omega'_1, u{:}\overset{\nabla}{\in}\mathcal{W} \leq_{\mathcal{W}_2} \Omega'_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega'_2$	by Lemma B.13.10
$\Omega'_1\ \mathsf{ctx}$	by inversion
$\Omega'_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega'_1, u{:}\overset{\nabla}{\in}\mathcal{W}, \Omega'_2 \vdash e\backslash u \in \tau$	by popW

Case: $(e,\ f)$ and $\mathcal{E} = \dfrac{\Omega \vdash (\exists\alpha{\in}\delta.\ \tau)\ \mathsf{wff} \qquad \overset{\mathcal{E}_1}{\Omega \vdash e \in \delta} \qquad \overset{\mathcal{E}_2}{\Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]}}{\Omega \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\ \mathsf{pairI}$

$\Omega \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\Omega' \vdash e \in \delta$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]$	by i.h. on f with $\mathcal{E}_2$
$\Omega' \vdash \tau[\mathrm{id}_\Omega, e/\alpha]\ \mathsf{wff}$	by Lemma B.8.1
$\mathrm{id}_\Omega \leq \mathrm{id}_{\Omega'}$	by Lemma B.4.5
$\Omega' \vdash f \in \tau[\mathrm{id}_{\Omega'}, e/\alpha]$	by Lemma B.7.4
$\Omega \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by assumption
$\Omega' \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by Lemma B.7.3
$\Omega' \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau$	by pairI

Case: $(\mu u{\in}\tau.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega, u{\in}\tau \vdash e \in \tau\end{array}}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\ \mathsf{fix}$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$(\Omega, u{\in}\tau)$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \tau$ wff	by inversion using ctxAdd
$\Omega' \vdash \tau$ wff	by Lemma B.7.3
$(\Omega', u{\in}\tau)$ ctx	by ctxAdd
$\Omega, u{\in}\tau \leq \Omega', u{\in}\tau$	by leMiddle
$\Omega', u{\in}\tau \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash \mu u{\in}\tau.\ e \in \tau$	by fix

. . . . . .

Case: $(\epsilon\alpha{\in}\delta.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega \vdash \tau \text{ wff} \qquad \Omega, \alpha{\in}\delta \vdash c \in \tau\end{array}}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\ \mathsf{cEps}$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$(\Omega, \alpha{\in}\delta)$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \delta$ wff	by inversion using ctxAdd
$\Omega' \vdash \delta$ wff	by Lemma B.7.3
$(\Omega', \alpha{\in}\delta)$ ctx	by ctxAdd
$\Omega, \alpha{\in}\delta \leq \Omega', \alpha{\in}\delta$	by leMiddle
$\Omega', \alpha{\in}\delta \vdash c \in \tau$	by i.h. on c with $\mathcal{E}_1$
$\Omega \vdash \tau$ wff	by assumption
$\Omega' \vdash \tau$ wff	by Lemma B.7.3
$\Omega' \vdash \epsilon\alpha{\in}\delta.\ c \in \tau$	by cEps

Case: $(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau\end{array}}{\Omega \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{cNew}$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \boldsymbol{A}^{\#}$ wff	by inversion using ctxAdd
$\Omega' \vdash \boldsymbol{A}^{\#}$ wff	by Lemma B.7.3
$(\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ ctx	by ctxAdd
$\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \leq \Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by leMiddle
$\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau$	by i.h. on c with $\mathcal{E}_1$
$\Omega' \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$	by cNew

Case: $(\overline{e} \mapsto f)$ and

$$\mathcal{E} = \frac{\begin{array}{ccc}\mathcal{E}_1 & \mathcal{E}_2 & \\ \Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}] & \Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta} & \Omega \vdash \forall \overline{\alpha{\in}\delta}.\ \tau \text{ wff}\end{array}}{\Omega \vdash \overline{e} \mapsto f \in \forall \overline{\alpha{\in}\delta}.\ \tau}\ \mathsf{cMatch}$$

$\Omega \leq \Omega'$	by assumption
Ω' ctx	by assumption
$\Omega' \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}]$	by i.h. on f with $\mathcal{E}_1$
$\Omega' \vdash \mathrm{id}_{\Omega'}, \overline{e}/\overline{\alpha} : \Omega', \overline{\alpha{\in}\delta}$	by i.h. on $\overline{e}$ with $\mathcal{E}_2$
$\Omega' \vdash \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}]$ wff	by Lemma B.8.1
$\mathrm{id}_\Omega \leq \mathrm{id}_{\Omega'}$	by Lemma B.4.5
$\tau[\mathrm{id}_{\Omega'}, \overline{e}/\overline{\alpha}] = \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}]$	by Lemma B.7.4
$\Omega' \vdash f \in \tau[\mathrm{id}_{\Omega'}, \overline{e}/\overline{\alpha}]$	by above
$\Omega \vdash \forall \alpha{\in}\delta.\ \tau$ wff	by assumption
$\Omega' \vdash \forall \alpha{\in}\delta.\ \tau$ wff	by Lemma B.7.3
$\Omega' \vdash \overline{e} \mapsto f \in \forall \overline{\alpha{\in}\delta}.\ \tau$	by cMatch

Case: $(c\backslash\boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$

See Case for pop (above)

. . . . . .

Case: *nil* and $\mathcal{E} = \Omega \vdash \mathrm{id}_\Omega : \Omega$

$\Omega \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\Omega' \vdash \mathrm{id}_{\Omega'} : \Omega'$	by Lemma B.5.5

Case: $(\overline{f'}; f)$ and $\mathcal{E} = \dfrac{\mathcal{E}_1 \qquad\qquad \mathcal{E}_2}{\Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'} : \Omega, \overline{\alpha'{\in}\delta'} \qquad \Omega \vdash f \in \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}]}{\Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega, \overline{\alpha'{\in}\delta'}, \alpha{\in}\delta}\ \mathsf{tpSubInd}$

$\Omega \leq \Omega'$	by assumption
$\Omega'\ \mathsf{ctx}$	by assumption
$\Omega' \vdash \mathrm{id}_{\Omega'}, \overline{f'}/\overline{\alpha'} : \Omega', \overline{\alpha'{\in}\delta'}$	by i.h. on $\overline{f'}$ with $\mathcal{E}_1$
$\Omega' \vdash f \in \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}]$	by i.h. on f with $\mathcal{E}_2$
$\Omega' \vdash \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}]\ \mathsf{wff}$	by Lemma B.8.1
$\mathrm{id}_\Omega \leq \mathrm{id}_{\Omega'}$	by Lemma B.4.5
$\delta[\mathrm{id}_{\Omega'}, \overline{f'}/\overline{\alpha'}] = \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}]$	by Lemma B.7.4
$\Omega' \vdash f \in \delta[\mathrm{id}_{\Omega'}, \overline{f'}/\overline{\alpha'}]$	by above
$\Omega' \vdash \mathrm{id}_{\Omega'}, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega', \overline{\alpha'{\in}\delta'}, \alpha{\in}\delta$	by $\mathsf{tpSubInd}$

□

Lemma B.14.5 (Context Weakening over Typing with Worlds)**.**

- *If* $\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash e \in \delta$ *and* $(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ *and* $(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ ctx*, then* $\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e \in \delta$*.*
- *If* $\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash c \in \tau$ *and* $(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ *and* $(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ ctx*, then* $\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash c \in \tau$*.*
- *If* $\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f}/\overline{\alpha} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \overline{\alpha\in\delta}$ *and* $(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ *and* $(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ ctx*, then* $\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f}/\overline{\alpha} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \overline{\alpha\in\delta}$*.*

Proof. By induction on e and c and $\overline{f}$.
Notice that there are two cases each for $e\backslash\boldsymbol{x}$, $c\backslash\boldsymbol{x}$, and u'. There are three cases for $e\backslash u'$.

Case: () and $\mathcal{E} = \dfrac{(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash () \in \mathrm{unit}}\ \mathsf{top}$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ ctx	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash () \in \mathrm{unit}$	by top

Case: u' and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, u'{\in}\tau, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx} \\ (\Omega_1, u'{\in}\tau) \leq (\Omega_1, u'{\in}\tau, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\end{array}}{\Omega_1, u'{\in}\tau, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash u' \in \tau}\ \tau\mathsf{var}$

$(\Omega_1, u'{\in}\tau) \leq (\Omega_1, u'{\in}\tau, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)$ by assumption
This case is impossible by inversion

Case: u' and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u'{\in}\tau, \Omega_2)\ \mathsf{ctx} \\ (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u'{\in}\tau) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u'{\in}\tau, \Omega_2)\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u'{\in}\tau, \Omega_2 \vdash u' \in \tau}\ \tau\mathsf{var}$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'{\in}\tau, \Omega_2)\ \mathsf{ctx}$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u'{\in}\tau) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u'{\in}\tau, \Omega_2)$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_* (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by Lemma B.7.7
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'{\in}\tau) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'{\in}\tau, \Omega_2)$
by Lemma B.13.15
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'{\in}\tau, \Omega_2 \vdash u' \in \tau$ by $\tau\mathsf{var}$

Case: $\boldsymbol{x}$ and
$$\mathcal{E} = \frac{(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx} \quad ((\boldsymbol{x}{:}\in\boldsymbol{A}^{\#})\text{ or }(\boldsymbol{x}{:}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}))\text{ in }(\Omega_{\mathrm{A}}, \mathrm{u}{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_{\mathrm{C}})}{\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\ \mathsf{var}^{\#}$$

$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$((\boldsymbol{x}{:}\in\boldsymbol{A}^{\#})\text{ or }(\boldsymbol{x}{:}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}))\text{ in }(\Omega_{\mathrm{A}}, \mathrm{u}{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_{\mathrm{C}})$	by assumption
$((\boldsymbol{x}{:}\in\boldsymbol{A}^{\#})\text{ or }(\boldsymbol{x}{:}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}))\text{ in }(\Omega_{\mathrm{A}}, \mathrm{u}{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_{\mathrm{B}}, \Omega_{\mathrm{C}})$	by above
$\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}$	by $\mathsf{var}^{\#}$

Case: $\boldsymbol{M}$ and $\mathcal{E} = \dfrac{(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx} \quad \|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$\|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$\|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_C\| \leq_{\mathsf{lf}} \|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C\|$	by Lemma B.4.4
$\|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C\|\ \mathsf{ctx}_{\mathsf{lf}}$	by Lemma B.4.1
$\|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by Lemma B.1.9
$\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \boldsymbol{M} \in \boldsymbol{A}$	by isLF

Case: (fn $\overline{c}$) and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx} \\ \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \tau\ \mathsf{wff} \\ \mathcal{E}_1 :: \text{for all }_{c_i\in\overline{c}}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash c_i \in \tau) \\ \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \overline{c}\text{ covers }\tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \text{fn }\overline{c} \in \tau}\ \mathsf{impI}$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \tau\ \mathsf{wff}$	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \tau\ \mathsf{wff}$	by Lemma B.7.1
for all $_{c_i\in\overline{c}}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash c_i \in \tau)$	by i.h. on c_i with $\mathcal{E}_1$ (for all $c_i \in \overline{c}$)
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \overline{c}\text{ covers }\tau$	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \overline{c}\text{ covers }\tau$	by Lemma B.14.3
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \text{fn }\overline{c} \in \tau$	by impI

Case: $(e\ \overline{f})$ and

$$\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash e \in \forall\overline{\alpha\in\delta}.\ \tau \\ \mathcal{E}_2 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f}/\overline{\alpha} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \overline{\alpha\in\delta}\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash e\ \overline{f} \in \tau[\text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e \in \forall\overline{\alpha\in\delta}.\ \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f}/\overline{\alpha} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \overline{\alpha\in\delta}$	by i.h. on $\overline{f}$ with $\mathcal{E}_2$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e\ \overline{f} \in \tau[\text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f}/\overline{\alpha}]$	by impE
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \tau[\text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f}/\overline{\alpha}]\ \mathsf{wff}$	by Lemma B.8.1 on $\mathcal{E}$
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$\text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C} \leq \text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}$	by Lemma B.4.5
$\tau[\text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f}/\overline{\alpha}] = \tau[\text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f}/\overline{\alpha}]$	by Lemma B.7.4
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e\ \overline{f} \in \tau[\text{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f}/\overline{\alpha}]$	by above

Case: $(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{new}$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ ctx	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#})$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \boldsymbol{A}^{\#}$ wff	by inversion using ctxAddNew
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \boldsymbol{A}^{\#}$ wff	by Lemma B.7.1
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#})$ ctx	by ctxAddNew
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$	by new

Case: $(\nu u'{\in}\mathcal{W}'.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}' \vdash e \in \tau\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \nu u'{\in}\mathcal{W}'.\ e \in \nabla \mathcal{W}'.\ \tau}\ \mathsf{newW}$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ ctx	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}')$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\mathcal{W}'$ world	by inversion using ctxAddWorld
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}')$ ctx	by ctxAddWorld
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}' \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \nu u'{\in}\mathcal{W}'.\ e \in \nabla \mathcal{W}'.\ \tau$	by newW

Case: $(e\backslash\boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx}\\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\boldsymbol{x'}\in\boldsymbol{A}^{\#}.\ \tau\\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)$ by assumption
This case is impossible by inversion.

Case: $(e\backslash\boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx}\\ \mathcal{E}_1 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash e \in \nabla\boldsymbol{x'}\in\boldsymbol{A}^{\#}.\ \tau\\ (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx}$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$ by inversion
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e \in \nabla\boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}.\ \tau$ by i.h. on e with $\mathcal{E}_1$
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_* (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by Lemma B.7.7
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by Lemma B.13.15
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]\ \mathsf{wff}$ by Lemma B.8.1 on $\mathcal{E}$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \leq_{\mathcal{W}} \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C$ by Lemma B.13.8
$\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C} \leq \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}$ by Lemma B.4.5
$\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'} \leq \uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}$ by rules
$\tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}] = \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]$ by Lemma B.7.4
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]$ by pop and above

Case: $(e\backslash u')$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W}' \leq \mathcal{W}_2 \\ (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C)\end{array}}{\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash e\backslash u' \in \tau}\ \mathsf{popW}$

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$ by assumption

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$ by assumption

$\Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau$ by assumption $(\mathcal{E}_1)$

$\Omega_1 \vdash \nabla\mathcal{W}_2.\ \tau\ \mathsf{wff}$ by Lemma B.8.1

$\mathcal{W}_2\ \mathsf{world}$ by inversion using $\nabla_{\mathrm{world}}\mathsf{wff}$

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C)$ by assumption

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W})$ by Lemma B.13.12

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C)$ by Lemma B.13.12

$\mathcal{W} \leq \mathcal{W}_2$ by inversion using $\mathsf{leWorldAddMarker}$

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$ by Lemma B.13.7

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ by Lemma B.13.16

$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ by Lemma B.10.2

$\mathcal{W}' \leq \mathcal{W}_2$ by assumption

$\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e\backslash u' \in \tau$ by popW

Case: $(e\backslash u')$ and

$$\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)\ \mathsf{ctx}\\ \mathcal{E}_1 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash e \in \nabla\mathcal{W}_2.\ \tau\\ \mathcal{W}' \leq \mathcal{W}_2\\ (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2 \vdash e\backslash u' \in \tau}\ \mathsf{popW}$$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$ by assumption

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)\ \mathsf{ctx}$ by assumption

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$ by inversion

$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e \in \nabla\mathcal{W}_2.\ \tau$ by i.h. on e with $\mathcal{E}_1$

$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \nabla\mathcal{W}_2.\ \tau\ \mathsf{wff}$ by Lemma B.8.1

$\mathcal{W}_2\ \mathsf{world}$ by inversion using $\nabla_{\text{world}}\mathsf{wff}$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)$ by assumption

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_* (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$ by Lemma B.7.7

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)$ by Lemma B.13.14

$\mathcal{W}' \leq \mathcal{W}_2$ by assumption

$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2 \vdash e\backslash u' \in \tau$ by popW

Case: $(e\backslash u)$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C)\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash e\backslash u \in \tau}\ \mathsf{popW}$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$\mathcal{E}_1 :: \Omega_A \vdash e \in \nabla\mathcal{W}_2.\ \tau$	by assumption
$\mathcal{W} \leq \mathcal{W}_2$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C)$	by assumption
$\Omega_A \vdash \nabla\mathcal{W}_2.\ \tau\ \mathsf{wff}$	by Lemma B.8.1
$\mathcal{W}_2\ \mathsf{world}$	by inversion using $\nabla_{\text{world}}\mathsf{wff}$
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by Lemma B.13.7
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.16
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e\backslash u \in \tau$	by popW

Case: $(e,\ f)$ and $\mathcal{E} = \dfrac{\begin{array}{l}\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash (\exists\alpha{\in}\delta.\ \tau)\ \mathsf{wff} \\ \mathcal{E}_1 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash e \in \delta \\ \mathcal{E}_2 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, e/\alpha]\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\ \mathsf{pairI}$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash e \in \delta$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, e/\alpha]$	by i.h. on f with $\mathcal{E}_2$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, e/\alpha]\ \mathsf{wff}$	by Lemma B.8.1
$\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C} \leq \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C}$	by Lemma B.4.5
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C}, e/\alpha]$	by Lemma B.7.4
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \exists\alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by Lemma B.7.1
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau$	by pairI

Case: $(\mu u{\in}\tau.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u{\in}\tau \vdash e \in \tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \mu u{\in}\tau.\ e \in \tau}\ \mathsf{fix}$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, u{\in}\tau)\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \tau\ \mathsf{wff}$	by inversion using ctxAdd
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \tau\ \mathsf{wff}$	by Lemma B.7.1
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, u{\in}\tau)\ \mathsf{ctx}$	by ctxAdd
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, u{\in}\tau \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \mu u{\in}\tau.\ e \in \tau$	by fix

. . . .

Case: $(\epsilon\alpha{\in}\delta.\ c)$ and

$$\mathcal{E} = \dfrac{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \tau\ \mathsf{wff} \qquad \begin{array}{c}\mathcal{E}_1\\ \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \alpha{\in}\delta \vdash c \in \tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\ \mathsf{cEps}$$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \alpha{\in}\delta)\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \delta\ \mathsf{wff}$	by inversion using ctxAdd
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \delta\ \mathsf{wff}$	by Lemma B.7.1
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \alpha{\in}\delta)\ \mathsf{ctx}$	by ctxAdd
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \alpha{\in}\delta \vdash c \in \tau$	by i.h. on c with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \tau\ \mathsf{wff}$	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \tau\ \mathsf{wff}$	by Lemma B.7.1
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \epsilon\alpha{\in}\delta.\ c \in \tau$	by cEps

Case: $(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{cNew}$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$	by inversion using ctxAddNew
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$	by Lemma B.7.1
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$	by ctxAddNew
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau$	by i.h. on c with $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$	by cNew

Case: $(\overline{e} \mapsto f)$ and

$$\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, \overline{e}/\overline{\alpha}] \\ \mathcal{E}_2 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, \overline{e}/\overline{\alpha} : \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C, \overline{\alpha{\in}\delta} \\ \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \forall \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \overline{e} \mapsto f \in \forall \overline{\alpha{\in}\delta}.\ \tau}\ \mathsf{cMatch}$$

$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C)$	by Lemma B.13.8
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, \overline{e}/\overline{\alpha}]$	by i.h. on f with $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{e}/\overline{\alpha} : \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C, \overline{\alpha{\in}\delta}$	by i.h. on $\overline{e}$ with $\mathcal{E}_2$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, \overline{e}/\overline{\alpha}]\ \mathsf{wff}$	by Lemma B.8.1
$\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C} \leq \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C}$	by Lemma B.4.5
$\tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{e}/\overline{\alpha}] = \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C}, \overline{e}/\overline{\alpha}]$	by Lemma B.7.4
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{e}/\overline{\alpha}]$	by above
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_C \vdash \forall \alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \forall \alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by Lemma B.7.1
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \overline{e} \mapsto f \in \forall \overline{\alpha{\in}\delta}.\ \tau$	by cMatch

Case: $(c\backslash\boldsymbol{x})$ and

$$\mathcal{E} = \frac{\begin{array}{l}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$$

or

$$\mathcal{E} = \frac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$$

See Cases for pop (above)

. . . . . .

Case: *nil* and $\mathcal{E} = \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$ by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C$
by Lemma B.5.5

Case: $(\overline{f'}; f)$ and $\mathcal{E} =$

$$\frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f'}/\overline{\alpha'} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \overline{\alpha'\in\delta'} \\ \mathcal{E}_2 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash f \in \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f'}/\overline{\alpha'}]\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C, \overline{\alpha'\in\delta'}, \alpha\in\delta}\ \mathsf{tpSubInd}$$

$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)\ \mathsf{ctx}$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C) \leq_{\mathcal{W}} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C)$ by Lemma B.13.8
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f'}/\overline{\alpha'} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \overline{\alpha'\in\delta'}$
by i.h. on $\overline{f'}$ with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash f \in \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f'}/\overline{\alpha'}]$ by i.h. on f with $\mathcal{E}_2$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f'}/\overline{\alpha'}]\ \mathsf{wff}$ by Lemma B.8.1
$\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C} \leq \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}$ by Lemma B.4.5
$\delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f'}/\overline{\alpha'}] = \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_C}, \overline{f'}/\overline{\alpha'}]$ by Lemma B.7.4
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C \vdash f \in \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f'}/\overline{\alpha'}]$ by above
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C$
$\quad\vdash\quad \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C}, \overline{f'}/\overline{\alpha'}, f/\alpha$
$\quad:\quad \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \Omega_C, \overline{\alpha'\in\delta'}, \alpha\in\delta$ by $\mathsf{tpSubInd}$

□

B.15 Meta-Theory: Strengthening

Lemma B.15.1 (Swap World in Well-Formedness)**.**
If $\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash \delta$ wff*, then* $\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2 \vdash \delta$ wff*.*

Proof. By induction on $\mathcal{E} :: \Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash \delta$ wff. This proof is straightforward. In the case for LF_wff we exploit that $\|\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2\| = \|\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_2\|$ since Casting (Def. A.2.1) throws out the u. □

Lemma B.15.2 (Well-Formedness of Context over Strengthening)**.**
If $(\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ ctx *and* $\mathcal{W}'$ world*, then* $(\Omega, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2)$ ctx*.*

Proof. By induction on Ω_2 using rules for $(\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ ctx as well as Lemma B.15.1 to verify that the types are still well-formed. □

Lemma B.15.3 (Swapping in World Inclusion (Alt))**.**
If $((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2), \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$*, then* $((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2), \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$*. Notice that there is no relationship between* $\mathcal{W}_2$ *and* $\mathcal{W}_3$*.*

Proof. By induction on $((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2), \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$. In the case for includesAltYes we exploit that $\|(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2)\|^{\triangledown} = \|(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2)\|^{\triangledown}$ since Casting (Def. B.2.1) throws out the u. □

Lemma B.15.4 (Swapping in World Inclusion)**.**
If $((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2), \boldsymbol{A}) \in \mathcal{W}$ *and* $\mathcal{W}$ world *and* $(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2)$ ctx*,*
then $((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2), \boldsymbol{A}) \in \mathcal{W}$*.*

Proof.

$((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2), \boldsymbol{A}) \in \mathcal{W}$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2)$ ctx	by assumption
$((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2), \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$	by Lemma B.9.2
$((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2), \boldsymbol{A}) \in_{\text{alt}} \mathcal{W}$	by Lemma B.15.3
$((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2), \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.9.1

□

Lemma B.15.5 (Strengthening World in $\leq_{\mathcal{W}}$)**.**
If $\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3$ *and* $\mathcal{W}_3$ world *and* $\mathcal{W}_1 \leq \mathcal{W}_2$ *and* $(\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3)$ ctx*, then* $\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3$.

Proof. By induction on Ω_3.

Case: $\cdot$

$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2$	by assumption
$\mathcal{W}_3$ world	by assumption
$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2$	by inversion using leWorldAddMarker
$\mathcal{W}_2 \leq \mathcal{W}_3$	by inversion using leWorldAddMarker
$\mathcal{W}_1 \leq \mathcal{W}_3$	by Lemma B.13.3
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1$	by leWorldAddMarker

Case: $\Omega_3, \alpha{\in}\delta$

$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3, \alpha{\in}\delta$	by assumption
$\mathcal{W}_3$ world	by assumption
$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$(\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3, \alpha{\in}\delta)$ ctx	by assumption
$(\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3)$ ctx	by inversion using ctxAdd
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3$	by inversion using leWorldAdd
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3$	by i.h. on Ω_3
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3, \alpha{\in}\delta$	by leWorldAdd

Case: $\Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$

$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by assumption
$\mathcal{W}_3$ world	by assumption
$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$(\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ ctx	by assumption
$(\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3)$ ctx	by inversion using ctxAddNew
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3$	by inversion using leWorldAddNew
$((\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3), \boldsymbol{A}) \in \mathcal{W}_3$	by inversion using leWorldAddNew
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3$	by i.h. on Ω_3
$((\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3), \boldsymbol{A}) \in \mathcal{W}_3$	by Lemma B.15.4
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by leWorldAddNew

Case: $\Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}'$

$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}'$	by assumption
$\mathcal{W}_3$ world	by assumption
$\mathcal{W}_1 \leq \mathcal{W}_2$	by assumption
$(\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}')$ ctx	by assumption
$(\Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3)$ ctx	by inversion using ctxAddWorld
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_3$	by inversion using leWorldAddMarker
$\mathcal{W}' \leq \mathcal{W}_3$	by inversion using leWorldAddMarker
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3$	by i.h. on Ω_3
$\Omega \leq_{\mathcal{W}_3} \Omega, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W}_1, \Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}'$	by leWorldAddMarker

□

Lemma B.15.6 (Swap Element in $\leq_{\mathcal{W}}$)**.**
If $(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2, \Omega_3)$ *and* $\mathcal{W}$ world *and* $(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3)$ ctx*, then* $(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3)$.

Proof. By induction on Ω_3.

Case: $\cdot$

$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2)$	by assumption
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2)$	by leWorldEq

Case: $\Omega_3, \alpha{\in}\delta$

$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2, \Omega_3, \alpha{\in}\delta)$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3, \alpha{\in}\delta)$ ctx	by assumption
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3)$ ctx	by inversion using ctxAdd
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2, \Omega_3)$	by inversion using leWorldAdd
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3)$	by i.h. on Ω_3
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2\Omega_3, \alpha{\in}\delta)$	by leWorldAdd

Case: $\Omega_3, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}$

$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2, \Omega_3, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#})$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#})$ ctx	by assumption
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3)$ ctx	by inversion using ctxAddNew
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2, \Omega_3)$	by inversion using leWorldAddNew
$((\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2, \Omega_3), \boldsymbol{A}) \in \mathcal{W}$	by inversion using leWorldAddNew
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3)$	by i.h. on Ω_3
$((\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_3, \Omega_2, \Omega_3), \boldsymbol{A}) \in \mathcal{W}$	by Lemma B.15.4
$(\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u{\overset{\triangledown}{\in}}\mathcal{W}_2, \Omega_2, \Omega_3, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#})$	by leWorldAddNew

Case: $\Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}'$

$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2, \Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}')$	by assumption
$\mathcal{W}$ world	by assumption
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2, \Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}')$ ctx	by assumption
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2, \Omega_3)$ ctx	by inversion using ctxAddWorld
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2, \Omega_3)$	by inversion using leWorldAddMarker
$\mathcal{W}' \leq \mathcal{W}$	by inversion using leWorldAddMarker
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_3, \Omega_2, \Omega_3)$	by i.h. on Ω_3
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2) \leq_{\mathcal{W}} (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}_2, \Omega_2, \Omega_3, u'\overset{\triangledown}{\in}\mathcal{W}')$	by leWorldAddMarker

□

Lemma B.15.7 (Swap Element in $\leq$)**.**
If $(\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}_2, \Omega_2) \leq (\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}_2, \Omega_2, \Omega_3)$ *and* $(\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}_3, \Omega_2, \Omega_3)$ ctx*,*
then $(\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}_3, \Omega_2) \leq (\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}_3, \Omega_2, \Omega_3)$.

Proof. By induction on Ω_3. Very similar to Lemma B.15.6, but the case with $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$ does not come up. □

Lemma B.15.8 (Strengthening of World in Global Parameter Coverage)**.**
If $\mathcal{W} \leq \mathcal{W}'$ *and* $\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2 \gg^{world} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
and $(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ ctx*,*
then $\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \gg^{world} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$*.*

Proof. By induction on Ω_2.

Case: $\cdot$

$\mathcal{W} \leq \mathcal{W}'$ by assumption
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by assumption
$\Omega_1 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
by inversion using gcCheckWorld
$\mathcal{W}' \leq \mathcal{W}_2$ by inversion using gcCheckWorld
$\mathcal{W}_2$ world by inversion using gcCheckWorld
$\mathcal{W}_2 \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by inversion using gcCheckWorld
$\mathcal{W} \leq \mathcal{W}_2$ by Lemma B.13.3
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W} \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by gcCheckWorld

Case: $\Omega_2, \alpha{\in}\delta$

$\mathcal{W} \leq \mathcal{W}'$ by assumption
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2, \alpha{\in}\delta \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
by assumption
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \alpha{\in}\delta)$ ctx by assumption
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ ctx by inversion using ctxAdd
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
by inversion using gcSkipNonParameter
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
by i.h. on Ω_2
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \alpha{\in}\delta \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
by gcSkipNonParameter

Case: $\Omega_2, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}$

$\mathcal{W} \leq \mathcal{W}'$ by assumption

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by assumption

$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#})$ ctx by assumption

$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ ctx by inversion using ctxAddNew

Subcase: gcSkipMismatch applied

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by inversion

$(\boldsymbol{B_x} \approx \boldsymbol{B_c})$ do not unify by inversion

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by i.h. on Ω_2

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by gcSkipMismatch

Subcase: gcSkipWorld applied

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by inversion

$\mathcal{W}_2$ world by inversion

$\mathcal{W}_2 \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by inversion

$((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}', \Omega_2), (\boldsymbol{\Pi\Gamma_c.\ B_c})) \in \mathcal{W}_2$ by inversion

$((\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2), (\boldsymbol{\Pi\Gamma_c.\ B_c})) \in \mathcal{W}_2$ by Lemma B.15.4

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by i.h. on Ω_2

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by gcSkipWorld

Case: $\Omega_2, u_3 \overset{\triangledown}{\in} \mathcal{W}_3$

$\mathcal{W} \leq \mathcal{W}'$ by assumption

$\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}', \Omega_2, u_3 \overset{\triangledown}{\in} \mathcal{W}_3 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by assumption

$(\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2, u_3 \overset{\triangledown}{\in} \mathcal{W}_3)$ ctx by assumption

$(\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2)$ ctx by inversion using ctxAddWorld

$\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}', \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by inversion using gcCheckWorld

$\mathcal{W}_3 \leq \mathcal{W}_2$ by inversion using gcCheckWorld

$\mathcal{W}_2$ world by inversion using gcCheckWorld

$\mathcal{W}_2 \gg \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by inversion using gcCheckWorld

$\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by i.h. on Ω_2

$\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2, u_3 \overset{\triangledown}{\in} \mathcal{W}_3 \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$ by gcCheckWorld

□

Lemma B.15.9 (Strengthening of World in Coverage)**.**
If $\mathcal{W}_0 \leq \mathcal{W}$ *and* $\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B \vdash \overline{c}$ covers τ *and* $(\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B)$ ctx*,*
then $\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B \vdash \overline{c}$ covers τ*.*

Proof. By induction on $\mathcal{E} : \Omega_A, u{:}{\in}\mathcal{W}, \Omega_B \vdash \overline{c}$ covers τ.
Note: We omit the completely trivial cases, which are $\mathsf{coverSimple}$, $\mathsf{coverPairLF}$, $\mathsf{coverPairLF}^{\#}$, $\mathsf{coverPairMeta}$, $\mathsf{coverNewLF}^{\#}$, $\mathsf{coverNewLF}$, and $\mathsf{coverNewMeta}$. The only interesting cases are $\mathsf{coverPop}$ (2 cases), $\mathsf{coverLF}$, and $\mathsf{coverEmpty}$.

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_1 \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_A, u{:}{\in}\mathcal{W}, \Omega_B)\end{array}}{\Omega_1, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_A, u{:}{\in}\mathcal{W}, \Omega_B \vdash \overline{c}\backslash\boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$$

$(\Omega_1, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_A, u{:}{\in}\mathcal{W}, \Omega_B)$ by assumption
This is impossible by inspection of the rules, so this case is vacuously true.

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u{:}{\in}\mathcal{W}, \Omega_B \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c}\backslash\boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$$

$\mathcal{W}_0 \leq \mathcal{W}$ by assumption
$(\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ ctx by assumption
$(\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B)$ ctx by inversion
$\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B \vdash \overline{c}$ covers $\nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau$ by i.h. on $\mathcal{E}_1$
$(\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by assumption
$(\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by Lemma B.15.7
$\mathrm{id}_{\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B} = \mathrm{id}_{\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B}$ by Def. of id
$\Omega_A, u{:}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}{:}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u{:}{\in}\mathcal{W}, \Omega_B}, \boldsymbol{x}/\boldsymbol{x'}]$
by $\mathsf{coverPop}$ and above

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{F}_1 :: (\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}})\ \mathsf{ctx} \\ \mathcal{F}_2 :: \Omega_X \text{ only contains declarations of type } \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#} \\ \mathcal{F}_3 :: \boldsymbol{\Sigma} \gg \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau \\ \mathcal{F}_4 :: \boldsymbol{\Gamma}_{\boldsymbol{x}} \gg \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau \\ \mathcal{F}_5 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau}\ \mathsf{coverLF}$

$\mathcal{W}_0 \leq \mathcal{W}$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B)\ \mathsf{ctx}$ by assumption
$\mathcal{F}'_5 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
by Lemma B.15.8 on $\mathcal{F}_5$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \overline{c} \text{ covers } \forall(\Omega_X, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma}_{\boldsymbol{x}}.\ \boldsymbol{B}_{\boldsymbol{x}}).\ \tau$
by $\mathsf{coverLF}$ using $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$, $\mathcal{F}'_5$

Case: $\mathcal{E} = \dfrac{\cdot \leq_{\cdot} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \mathit{nil} \text{ covers } (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverEmpty}^{\#}$

$\mathcal{W}_0 \leq \mathcal{W}$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B)\ \mathsf{ctx}$ by assumption
$\cdot \leq_{\cdot} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by assumption
$\mathcal{W} \leq \cdot$ by inversion
$\mathcal{W} = \cdot$ by Case analysis on the rules noting that it is impossible for $(\Omega_*, \boldsymbol{A}_*) \in \cdot$
$\mathcal{W}_0 \leq \cdot$ by above
$\mathcal{W}_0 = \mathcal{W} = \cdot$ by above
$\cdot \leq_{\cdot} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B)$ by above
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \mathit{nil} \text{ covers } (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)$ by $\mathsf{coverEmpty}$

□

Lemma B.15.10 (Strengthening of World in Typing)**.**

- *If* $\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B \vdash e \in \delta$ *and* $\mathcal{W}_0 \leq \mathcal{W}$ *and* $\mathcal{W}_0$ world*,*
  then $\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}_0, \Omega_B \vdash e \in \delta$*.*
- *If* $\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B \vdash c \in \tau$ *and* $\mathcal{W}_0 \leq \mathcal{W}$ *and* $\mathcal{W}_0$ world*,*
  then $\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}_0, \Omega_B \vdash c \in \tau$*.*
- *If* $\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B \vdash \mathrm{id}_{\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B}, \overline{f}/\overline{\alpha} : \Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B, \overline{\alpha{\in}\delta}$
  and $\mathcal{W}_0 \leq \mathcal{W}$ *and* $\mathcal{W}_0$ world*,*
  then $\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}_0, \Omega_B \vdash \mathrm{id}_{\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}_0, \Omega_B}, \overline{f}/\overline{\alpha} : \Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}_0, \Omega_B, \overline{\alpha{\in}\delta}$*.*

Proof. By induction on e and c and $\overline{f}$.
Notice that there are two cases each for $e\backslash\boldsymbol{x}$, $c\backslash\boldsymbol{x}$, and u'. There are three cases for $e\backslash u'$.

Case: () and $\mathcal{E} = \dfrac{(\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B)\ \mathsf{ctx}}{\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B \vdash () \in \mathrm{unit}}\ \mathsf{top}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B)$ ctx	by assumption
$(\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}_0, \Omega_B)$ ctx	by Lemma B.15.2
$\Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}_0, \Omega_B \vdash () \in \mathrm{unit}$	by top

Case: u' and $\mathcal{E} = \dfrac{\begin{array}{c}(\Omega_1, u'{\in}\tau, \Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \\ (\Omega_1, u'{\in}\tau) \leq (\Omega_1, u'{\in}\tau, \Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B)\end{array}}{\Omega_1, u'{\in}\tau, \Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B \vdash u' \in \tau}\ \tau\mathsf{var}$

$(\Omega_1, u'{\in}\tau) \leq (\Omega_1, u'{\in}\tau, \Omega_A, u{\overset{\triangledown}{\in}}\mathcal{W}, \Omega_B)$	by assumption
This is impossible by inversion	

Case: u' and $\mathcal{E} = \dfrac{(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'{:}\in\tau, \Omega_2)\ \mathsf{ctx} \quad (\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'{:}\in\tau) \leq (\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'{:}\in\tau, \Omega_2)}{\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'{:}\in\tau, \Omega_2 \vdash u' \in \tau}\ \tau\mathsf{var}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'{:}\in\tau, \Omega_2)$ ctx	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, u'{:}\in\tau, \Omega_2)$ ctx	by Lemma B.15.2
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'{:}\in\tau) \leq (\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'{:}\in\tau, \Omega_2)$	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, u'{:}\in\tau) \leq (\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, u'{:}\in\tau, \Omega_2)$	by Lemma B.15.7
$\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, u'{:}\in\tau, \Omega_2 \vdash u' \in \tau$	by $\tau\mathsf{var}$

Case: $\boldsymbol{x}$ and $\mathcal{E} = \dfrac{(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \quad ((\boldsymbol{x}{:}\in\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}{:}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } (\Omega_{\mathrm{A}}, \mathrm{u}{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_{\mathrm{B}})}{\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\ \mathsf{var}^{\#}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ ctx	by assumption
$((\boldsymbol{x}{:}\in\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}{:}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } (\Omega_{\mathrm{A}}, \mathrm{u}{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_{\mathrm{B}})$	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B)$ ctx	by Lemma B.15.2
$((\boldsymbol{x}{:}\in\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}{:}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } (\Omega_{\mathrm{A}}, \mathrm{u}{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_{\mathrm{B}})$	by above (since $u \neq \boldsymbol{x}$)
$\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}$	by $\mathsf{var}^{\#}$

Case: $\boldsymbol{M}$ and $\mathcal{E} = \dfrac{(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \quad \|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ ctx	by assumption
$(\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B)$ ctx	by Lemma B.15.2
$\|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by assumption
$\|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}, \Omega_B\| = \|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B\|$	by Def. of Casting (Def. A.2.1)
$\|\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B\| \vdash^{\mathsf{lf}} \boldsymbol{M} : \boldsymbol{A}$	by above
$\Omega_A, u{:}\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \boldsymbol{M} \in \boldsymbol{A}$	by isLF

Case: $(\text{fn } \overline{c})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \\ \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \tau\ \mathsf{wff} \\ \mathcal{E}_1 :: \text{for all } {}_{c_i \in \overline{c}}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash c_i \in \tau) \\ \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \overline{c} \text{ covers } \tau\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \text{fn } \overline{c} \in \tau}\ \mathsf{impl}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$ ctx	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B)$ ctx	by Lemma B.15.2
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \tau$ wff	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash \tau$ wff	by Lemma B.15.1
for all ${}_{c_i \in \overline{c}}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash c_i \in \tau)$	by i.h. on c_i with $\mathcal{E}_1$ (for all $c_i \in \overline{c}$)
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \overline{c}$ covers τ	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash \overline{c}$ covers τ	by Lemma B.15.9
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash \text{fn } \overline{c} \in \tau$	by impl

Case: $(e\ \overline{f})$ and

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash e \in \forall\overline{\alpha\in\delta}.\ \tau \\ \mathcal{E}_2 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \overline{f}/\overline{\alpha} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \overline{\alpha\in\delta}\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash e\ \overline{f} \in \tau[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash e \in \forall\overline{\alpha\in\delta}.\ \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B}, \overline{f}/\overline{\alpha} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \overline{\alpha\in\delta}$	by i.h. on $\overline{f}$ with $\mathcal{E}_2$
$\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B} = \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B}$	by Def. of id
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash e\ \overline{f} \in \tau[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \overline{f}/\overline{\alpha}]$	by impE and above

Case: $(\nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e)$ and $\mathcal{E} = \dfrac{\mathcal{E}_1}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau}\ \mathsf{new}$

$$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ \tau$$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ \tau$	by new

Case: $(\nu u'\in\mathcal{W}'.\ e)$ and $\mathcal{E} = \dfrac{\mathcal{E}_1}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u'\overset{\triangledown}{\in}\mathcal{W}' \vdash e \in \tau}\ \mathsf{newW}$

$$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \nu u'\in\mathcal{W}'.\ e \in \nabla\mathcal{W}'.\ \tau$$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, u'\overset{\triangledown}{\in}\mathcal{W}' \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \nu u'\in\mathcal{W}'.\ e \in \nabla\mathcal{W}'.\ \tau$	by newW

Case: $(e\backslash \boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ by assumption
This case is impossible by inversion.

Case: $(e\backslash \boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash e \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$\mathcal{W}_0 \leq \mathcal{W}$ by assumption
$\mathcal{W}_0\ \mathsf{world}$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx}$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx}$ by Lemma B.15.2
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash e \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau$ by i.h. on e with $\mathcal{E}_1$
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by Lemma B.15.7
$\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B} = \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B}$ by Def. of id
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \boldsymbol{x}/\boldsymbol{x'}]$
by pop and above

Case: $(e\backslash u')$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W}' \leq \mathcal{W}_2 \\ (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)\end{array}}{\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash e\backslash u' \in \tau}\ \mathsf{popW}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$ ctx	by assumption
$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B)$ ctx	by Lemma B.15.2
$\Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau$	by assumption ($\mathcal{E}_1$)
$\mathcal{W}' \leq \mathcal{W}_2$	by assumption
$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$\Omega_1 \vdash \nabla\mathcal{W}_2.\ \tau$ wff	by Lemma B.8.1
$\mathcal{W}_2$ world	by inversion using ∇_{world}wff
$(\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B)$	by Lemma B.15.5
$\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash e\backslash u' \in \tau$	by popW

Case: $(e\backslash u')$ and

$$\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W}' \leq \mathcal{W}_2 \\ (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2 \vdash e\backslash u' \in \tau}\ \mathsf{popW}$$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0\ \mathsf{world}$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)\ \mathsf{ctx}$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)\ \mathsf{ctx}$	by Lemma B.15.2
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash e \in \nabla\mathcal{W}_2.\ \tau$	by i.h. on e with $\mathcal{E}_1$
$\mathcal{W}' \leq \mathcal{W}_2$	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)$	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \nabla\mathcal{W}_2.\ \tau\ \mathsf{wff}$	by Lemma B.8.1 on $\mathcal{E}_1$
$\mathcal{W}_2\ \mathsf{world}$	by inversion using $\nabla_{\text{world}}\mathsf{wff}$
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}') \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2)$	by Lemma B.15.6
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B, u'\overset{\nabla}{\in}\mathcal{W}', \Omega_2 \vdash e\backslash u' \in \tau$	by popW

Case: $(e\backslash u)$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash e\backslash u \in \tau}\ \mathsf{popW}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$ ctx	by assumption
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B)$ ctx	by Lemma B.15.2
$\Omega_A \vdash e \in \nabla\mathcal{W}_2.\ \tau$	by assumption ($\mathcal{E}_1$)
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)$	by assumption
$\Omega_A \vdash \nabla\mathcal{W}_2.\ \tau$ wff	by Lemma B.8.1 on $\mathcal{E}_1$
$\mathcal{W}_2$ world	by inversion using ∇_{world}wff
$(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B)$	by Lemma B.15.6
$\mathcal{W} \leq \mathcal{W}_2$	by assumption
$\Omega_A \vdash \nabla\mathcal{W}_2.\ \tau$ wff	by Lemma B.8.1
$\mathcal{W}_2$ world	by inversion using ∇_{world}wff
$\mathcal{W}_0 \leq \mathcal{W}_2$	by Lemma B.13.3
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash e\backslash u \in \tau$	by popW

Case: $(e,\ f)$ and $\mathcal{E} = \dfrac{\begin{array}{l}\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash (\exists\alpha{\in}\delta.\ \tau)\ \mathsf{wff} \\ \mathcal{E}_1 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash e \in \delta \\ \mathcal{E}_2 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B}, e/\alpha]\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\ \mathsf{pairI}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash (\exists\alpha{\in}\delta.\ \tau)$ wff	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash (\exists\alpha{\in}\delta.\ \tau)$ wff	by Lemma B.15.1
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash e \in \delta$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B}, e/\alpha]$	by i.h. on f with $\mathcal{E}_2$
$\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B} = \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B}$	by Def. of id
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B}, e/\alpha]$	by above
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau$	by pairI

Case: $(\mu u{\in}\tau.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, u{\in}\tau \vdash e \in \tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \mu u{\in}\tau.\ e \in \tau}\ \mathsf{fix}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, u{\in}\tau \vdash e \in \tau$	by i.h. on e with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \mu u{\in}\tau.\ e \in \tau$	by fix

. . . . . .

Case: $(\epsilon\alpha{\in}\delta.\ c)$ and

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1\\ \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \tau\ \mathsf{wff} \qquad \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \alpha{\in}\delta \vdash c \in \tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\ \mathsf{cEps}$$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \tau\ \mathsf{wff}$	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \tau\ \mathsf{wff}$	by Lemma B.15.1
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \alpha{\in}\delta \vdash c \in \tau$	by i.h. on c with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \epsilon\alpha{\in}\delta.\ c \in \tau$	by cEps

Case: $(\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{cNew}$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau$	by i.h. on c with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$	by cNew

Case: $(\overline{e} \mapsto f)$ and

$$\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B}, \overline{e}/\overline{\alpha}] \\ \mathcal{E}_2 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B}, \overline{e}/\overline{\alpha} : \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \overline{\alpha\in\delta} \\ \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \forall\overline{\alpha\in\delta}.\ \tau\ \mathsf{wff}\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \overline{e} \mapsto f \in \forall\overline{\alpha\in\delta}.\ \tau}\ \mathsf{cMatch}$$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B}, \overline{e}/\overline{\alpha}]$	by i.h. on f with $\mathcal{E}_1$
$\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B} = \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B}$	by Def. of id
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash f \in \tau[\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B}, \overline{e}/\overline{\alpha}]$	by above
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B}, \overline{e}/\overline{\alpha} : \Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B, \overline{\alpha\in\delta}$	by i.h. on $\overline{e}$ with $\mathcal{E}_2$
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash \forall\overline{\alpha\in\delta}.\ \tau\ \mathsf{wff}$	by assumption
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash \forall\overline{\alpha\in\delta}.\ \tau\ \mathsf{wff}$	by Lemma B.15.1
$\Omega_A, u\overset{\nabla}{\in}\mathcal{W}_0, \Omega_B \vdash \overline{e} \mapsto f \in \forall\overline{\alpha\in\delta}.\ \tau$	by cMatch

Case: $(c\backslash\boldsymbol{x})$ and

$$\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$$

or

$$\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash c \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_A, u\overset{\nabla}{\in}\mathcal{W}, \Omega_B \vdash c\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$$

See Cases for pop (above)

. . . . . .

Case: *nil* and $\mathcal{E} = \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)$ ctx	by Lemma B.3.2 on $\mathcal{E}$
$(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B)$ ctx	by Lemma B.15.2
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B$	by Lemma B.5.5

Case: $(\overline{f'}; f)$ and $\mathcal{E} =$

$$\frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \overline{f'}/\overline{\alpha'} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \overline{\alpha'\in\delta'} \\ \mathcal{E}_2 :: \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash f \in \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \overline{f'}/\overline{\alpha'}]\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B, \overline{\alpha'\in\delta'}, \alpha\in\delta}\ \mathsf{tpSubInd}$$

$\mathcal{W}_0 \leq \mathcal{W}$	by assumption
$\mathcal{W}_0$ world	by assumption
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B}, \overline{f'}/\overline{\alpha'} : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \overline{\alpha'\in\delta'}$	by i.h. on $\overline{f'}$ with $\mathcal{E}_1$
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash f \in \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B}, \overline{f'}/\overline{\alpha'}]$	by i.h. on f with $\mathcal{E}_2$
$\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B} = \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B}$	by Def. of id
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash f \in \delta[\mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B}, \overline{f'}/\overline{\alpha'}]$	by above
$\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B \vdash \mathrm{id}_{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B}, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}_0, \Omega_B, \overline{\alpha'\in\delta'}, \alpha\in\delta$	by tpSubInd

□

B.16 Meta-Theory: Substitution Properties 2

Lemma B.16.1 (Substitution Inversion for pop)**.**
If $\Omega' \vdash \omega : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2$ *and* $(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$*, then there exists an* Ω_1'*,* Ω_2'*,* $\boldsymbol{x_0}$*, and* ω_1 *such that*

- $\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$
- *and* $\Omega' = \Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2'$
- *and* $\omega_1 \leq \omega$ *without using* leSubMiddle *or* leSubMiddleShift[2]
- *and* $(\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq \omega$
- *and* $(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \leq (\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2')$.

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2$.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2) \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2, \alpha{\in}\delta)$ by assumption
$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by inversion using leAdd
There exists an Ω_1', Ω_2', $\boldsymbol{x_0}$, and ω_1 such that
$\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$
and $\Omega' = \Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2'$
and $\omega_1 \leq \omega$ without using leSubMiddle or leSubMiddleShift
and $(\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq \omega$
and $(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \leq (\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2')$. by i.h. on $\mathcal{E}_1$
$\omega_1 \leq (\omega, e/\alpha)$ by leSubAdd
$(\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq (\omega, e/\alpha)$ by leSubAdd

[2]This condition on the derivation was added for use in Lemma B.18.5; it can also be viewed as stating that ω_1 is a subterm of ω.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2) \qquad \Omega' \vdash \boldsymbol{A''}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A''}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}}\omega, \boldsymbol{x'}/\boldsymbol{x''}) : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2, \boldsymbol{x''}\overset{\triangledown}{\in}\boldsymbol{A''}^{\#})}\ \mathsf{tpSubIndNew}$

$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2, \boldsymbol{x''}\overset{\triangledown}{\in}\boldsymbol{A''}^{\#})$ by assumption
This is impossible by inversion, so this case is vacuously true.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2) \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'}\omega, u'/u) : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2, u\overset{\triangledown}{\in}\mathcal{W})$ by assumption
This is impossible by inversion, so this case is vacuously true.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1' \vdash \omega_1 : \Omega \qquad \Omega_1' \vdash \boldsymbol{A}^{\#}[\omega_1]\ \mathsf{wff}\end{array}}{(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \vdash (\uparrow_{\boldsymbol{x_0}}\omega_1, \boldsymbol{x_0}/\boldsymbol{x}) : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

$\omega_1 \leq (\uparrow_{\boldsymbol{x_0}}\omega_1, \boldsymbol{x_0}/\boldsymbol{x})$ by $\mathsf{leSubShift}$ and $\mathsf{leSubAdd}$
$(\uparrow_{\boldsymbol{x_0}}\omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq (\uparrow_{\boldsymbol{x_0}}\omega_1, \boldsymbol{x_0}/\boldsymbol{x})$ by $\mathsf{leSubEq}$
$(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \leq (\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1])$ by leEq

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2) \qquad \Omega' \vdash \delta'' \text{ wff}\end{array}}{\Omega', \alpha''{\in}\delta'' \vdash \uparrow_{\alpha''}\omega : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)}\,\mathsf{tpSubShift}$

$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by assumption

There exists an Ω_1', Ω_2', $\boldsymbol{x_0}$, and ω_1 such that

$$\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x_0}}\omega_1, \boldsymbol{x_0}/\boldsymbol{x}) : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$$

and $\Omega' = \Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2'$

and $\omega_1 \leq \omega$ without using $\mathsf{leSubMiddle}$ or $\mathsf{leSubMiddleShift}$

and $(\uparrow_{\boldsymbol{x_0}}\omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq \omega$

and $(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \leq (\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2')$. by i.h. on $\mathcal{E}_1$

$\Omega', \alpha''{\in}\delta'' = \Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2', \alpha''{\in}\delta''$ by above

$\omega_1 \leq (\uparrow_{\alpha''}\omega)$ by $\mathsf{leSubShift}$

$(\uparrow_{\boldsymbol{x_0}}\omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq (\uparrow_{\alpha''}\omega)$ by $\mathsf{leSubShift}$

$(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \leq (\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2', \alpha''{\in}\delta'')$ by leAdd

□

Lemma B.16.2 (Substitution Inversion for popW)**.**
If $\Omega' \vdash \omega : \Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2$ *and* $(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ *and* $\mathcal{W}_2$ world*, then there exists an* Ω_1'*,* Ω_2'*,* u_0*, and* ω_1 *such that*

- $\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u_0} \omega_1, u_0/u) : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$
- *and* $\Omega' = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2'$
- *and* $\omega_1 \leq \omega$ *without using* leSubMiddle *or* leSubMiddleShift[3]
- *and* $(\uparrow_{u_0} \omega_1, u_0/u) \leq \omega$
- *and* $(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2')$.

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2$.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2) \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \alpha{\in}\delta)}\ \textsf{tpSubInd}$

$\mathcal{W}_2$ world — by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \alpha{\in}\delta)$ — by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ — by inversion using leWorldAdd
There exists an Ω_1', Ω_2', u_0, and ω_1 such that
$\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u_0} \omega_1, u_0/u) : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$
and $\Omega' = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2'$
and $\omega_1 \leq \omega$ without using leSubMiddle or leSubMiddleShift
and $(\uparrow_{u_0} \omega_1, u_0/u) \leq \omega$
and $(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2')$. — by i.h. on $\mathcal{E}_1$
$\omega_1 \leq (\omega, e/\alpha)$ — by leSubAdd
$(\uparrow_{u_0} \omega_1, u_0/u) \leq (\omega, e/\alpha)$ — by leSubAdd

[3]This condition on the derivation was added for use in Lemma B.18.5; it can also be viewed as stating that ω_1 is a subterm of ω.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2) \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

$\mathcal{W}_2$ world — by assumption

$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ — by assumption

$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ — by inversion using leWorldAddNew

$((\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2), \boldsymbol{A}) \in \mathcal{W}_2$ — by inversion using leWorldAddNew

There exists an Ω_1', Ω_2', u_0, and ω_1 such that

$$\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u_0} \omega_1, u_0/u) : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$$

and $\Omega' = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2'$

and $\omega_1 \leq \omega$ without using leSubMiddle or leSubMiddleShift

and $(\uparrow_{u_0} \omega_1, u_0/u) \leq \omega$

and $(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2')$. — by i.h. on $\mathcal{E}_1$

$\Omega', \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega] = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2', \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega]$ — by above

$\omega_1 \leq (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x})$ — by leSubShift and leSubAdd

$(\uparrow_{u_0} \omega_1, u_0/u) \leq (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x})$ — by leSubShift and leSubAdd

$((\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2'), \boldsymbol{A}[\omega]) \in \mathcal{W}_2$ — by Lemma B.12.6

$(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2', \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega])$ — by leWorldAddNew

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2) \qquad \mathcal{W}'' \text{ world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}'') \vdash (\uparrow_{u'} \omega, u'/u'') : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, u''\overset{\triangledown}{\in}\mathcal{W}'')}\ \mathsf{tpSubWorld}$

$\mathcal{W}_2$ world — by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2, u''\overset{\triangledown}{\in}\mathcal{W}'')$ — by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ — by inversion using leWorldAddMarker
$\mathcal{W}'' \leq \mathcal{W}_2$ — by inversion using leWorldAddMarker
There exists an Ω_1', Ω_2', u_0, and ω_1 such that
$\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u_0} \omega_1, u_0/u) : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$
and $\Omega' = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2'$
and $\omega_1 \leq \omega$ without using leSubMiddle or leSubMiddleShift
and $(\uparrow_{u_0} \omega_1, u_0/u) \leq \omega$
and $(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2')$. — by i.h. on $\mathcal{E}_1$
$\Omega', u'\overset{\triangledown}{\in}\mathcal{W}'' = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2', u'\overset{\triangledown}{\in}\mathcal{W}''$ — by above
$\omega_1 \leq (\uparrow_{u'} \omega, u'/u)$ — by leSubShift and leSubAdd
$(\uparrow_{u_0} \omega_1, u_0/u) \leq (\uparrow_{u'} \omega, u'/u)$ — by leSubShift and leSubAdd
$(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2', u'\overset{\triangledown}{\in}\mathcal{W}'')$ — by leWorldAddMarker

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1' \vdash \omega_1 : \Omega \qquad \mathcal{W} \text{ world}\end{array}}{(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u_0} \omega_1, u_0/u) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

$\omega_1 \leq (\uparrow_{u_0} \omega_1, u_0/u)$ — by leSubShift and leSubAdd
$(\uparrow_{u_0} \omega_1, u_0/u) \leq (\uparrow_{u_0} \omega_1, u_0/u)$ — by leSubEq
$(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W})$ — by leWorldEq

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2) \qquad \Omega' \vdash \delta'' \ \mathsf{wff}\end{array}}{\Omega', \alpha''{\in}\delta'' \vdash \uparrow_{\alpha''}\omega : (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)}\ \mathsf{tpSubShift}$

$\mathcal{W}_2$ world	by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$	by assumption
There exists an Ω_1', Ω_2', u_0, and ω_1 such that $\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u_0}\omega_1, u_0/u) : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$ and $\Omega' = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2'$ and $\omega_1 \leq \omega$ without using leSubMiddle or leSubMiddleShift and $(\uparrow_{u_0}\omega_1, u_0/u) \leq \omega$ and $(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2')$.	by i.h. on $\mathcal{E}_1$
$\Omega', \alpha''{\in}\delta'' = \Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2', \alpha''{\in}\delta''$	by above
$\omega_1 \leq (\uparrow_{\alpha''}\omega)$	by leSubShift
$(\uparrow_{u_0}\omega_1, u_0/u) \leq (\uparrow_{\alpha''}\omega)$	by leSubShift
$(\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega_2', \alpha''{\in}\delta'')$	by leWorldAdd

□

Lemma B.16.3 ($\cdot$ Redundant on Left of Composition)**.**

$$\cdot \circ \omega = \omega$$

Proof. By induction on ω.

Case: $\omega = \uparrow_\alpha \omega'$

$$\begin{aligned}
&\cdot \circ (\uparrow_\alpha \omega') \\
&\quad = \uparrow_\alpha (\cdot \circ \omega') && \text{by Composition (Def. A.2.3)} \\
&\quad = \uparrow_\alpha \omega' && \text{by i.h. on } \omega'
\end{aligned}$$

Case: $\omega \neq \uparrow_\alpha \omega'$

$$\cdot \circ \omega = \omega \qquad \text{by Composition (Def. A.2.3)}$$

□

Lemma B.16.4 (id Redundant on Left of Composition)**.**
If $\Omega' \vdash \omega : \Omega$*, then* $(\mathrm{id}_\Omega \circ \omega) = \omega$*.*

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$.

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

$\mathrm{id}_\cdot = \cdot$ by Definition of id
$\cdot \circ \omega = \omega$ by Lemma B.16.3

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

$\mathrm{id}_{\Omega,\alpha\in\delta} = \uparrow_\alpha \mathrm{id}_\Omega, \alpha/\alpha$ by Definition of id
$(\uparrow_\alpha \mathrm{id}_\Omega, \alpha/\alpha) \circ (\omega, e/\alpha)]$
$= (\mathrm{id}_\Omega \circ \omega), e/\alpha$ by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)
$= (\omega, e/\alpha)$ by i.h. on $\mathcal{E}_1$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^\#[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^\#[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^\#)}\ \mathsf{tpSubIndNew}$

$\mathrm{id}_{\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#} = \uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x}$ by Definition of id
$(\uparrow_{\boldsymbol{x}} \mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x}) \circ ((\uparrow_{\boldsymbol{x'}} \omega), \boldsymbol{x'}/\boldsymbol{x})$
$= (\mathrm{id}_\Omega \circ (\uparrow_{\boldsymbol{x'}} \omega)), \boldsymbol{x'}/\boldsymbol{x}$ by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)
$= (\uparrow_{\boldsymbol{x'}} (\mathrm{id}_\Omega \circ \omega)), \boldsymbol{x'}/\boldsymbol{x}$ by Composition (Def. A.2.3)
$= (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x})$ by i.h. on $\mathcal{E}_1$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'}\omega, u'/u) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

$$
\begin{array}{ll}
\mathrm{id}_{\Omega, u\overset{\triangledown}{\in}\mathcal{W}} = \uparrow_u \mathrm{id}_\Omega, u/u & \text{by Definition of id} \\
(\uparrow_u \mathrm{id}_\Omega, u/u) \circ ((\uparrow_{u'}\omega), u'/u) & \\
\quad = (\mathrm{id}_\Omega \circ (\uparrow_{u'}\omega)), u'/u & \text{by Composition (Def. A.2.3)} \\
 & \text{and Subst. App. (Def. A.2.4)} \\
\quad = (\uparrow_{u'}(\mathrm{id}_\Omega \circ \omega)), u'/u & \text{by Composition (Def. A.2.3)} \\
\quad = (\uparrow_{u'}\omega, u'/u) & \text{by i.h. on } \mathcal{E}_1
\end{array}
$$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta\ \mathsf{wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

$$
\begin{array}{ll}
\mathrm{id}_\Omega \circ (\uparrow_\alpha \omega) & \\
\quad = \uparrow_\alpha (\mathrm{id}_\Omega \circ \omega) & \text{by Composition (Def. A.2.3)} \\
\quad = \uparrow_\alpha \omega & \text{by i.h. on } \mathcal{E}_1
\end{array}
$$

□

Lemma B.16.5 (id Redundant on Right of Composition)**.**
If $\Omega' \vdash \omega : \Omega$*, then* $(\omega \circ \mathrm{id}_{\Omega'}) = \omega$*.*

Proof. By induction on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$.

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

$(\cdot \circ \mathrm{id}_{\cdot}) = \cdot$ by Composition (Def. A.2.3) and Def. of id

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

$\Omega' \vdash e \in \delta[\omega]$ by assumption
$e[\mathrm{id}_{\Omega'}] = e$ by Lemma B.5.3
$((\omega, e/\alpha) \circ \mathrm{id}_{\Omega'})$
$= (\omega \circ \mathrm{id}_{\Omega'}), e/\alpha$ by Composition (Def. A.2.3), Definition of id, and above
$= (\omega, e/\alpha)$ by i.h. on $\mathcal{E}_1$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

$((\uparrow_{\boldsymbol{x'}} \omega), \boldsymbol{x'}/\boldsymbol{x}) \circ \mathrm{id}_{\Omega', \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega]}$
$= ((\uparrow_{\boldsymbol{x'}} \omega), \boldsymbol{x'}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x'}} \mathrm{id}_{\Omega'}, \boldsymbol{x'}/\boldsymbol{x'})$ by Def. of id
$= ((\uparrow_{\boldsymbol{x'}} \omega) \circ (\uparrow_{\boldsymbol{x'}} \mathrm{id}_{\Omega'}, \boldsymbol{x'}/\boldsymbol{x'})), \boldsymbol{x'}/\boldsymbol{x}$ by Composition (Def. A.2.3), and Subst. App. (Def. A.2.4)
$= (\omega \circ (\uparrow_{\boldsymbol{x'}} \mathrm{id}_{\Omega'})), \boldsymbol{x'}/\boldsymbol{x}$ by Composition (Def. A.2.3)
$= (\uparrow_{\boldsymbol{x'}} (\omega \circ \mathrm{id}_{\Omega'})), \boldsymbol{x'}/\boldsymbol{x}$ by Composition (Def. A.2.3)
$= (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x})$ by i.h. on $\mathcal{E}_1$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'}\omega, u'/u) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

$$
\begin{array}{lr}
((\uparrow_{u'}\omega), u'/u) \circ \mathrm{id}_{\Omega', u'\overset{\triangledown}{\in}\mathcal{W}} & \\
\quad = ((\uparrow_{u'}\omega), u'/u) \circ (\uparrow_{u'}\mathrm{id}_{\Omega'}, u'/u') & \text{by Def. of id} \\
\quad = ((\uparrow_{u'}\omega) \circ (\uparrow_{u'}\mathrm{id}_{\Omega'}, u'/u')), u'/u & \text{by Composition (Def. A.2.3),} \\
 & \text{and Subst. App. (Def. A.2.4)} \\
\quad = (\omega \circ (\uparrow_{u'}\mathrm{id}_{\Omega'})), u'/u & \text{by Composition (Def. A.2.3)} \\
\quad = (\uparrow_{u'}(\omega \circ \mathrm{id}_{\Omega'})), u'/u & \text{by Composition (Def. A.2.3)} \\
\quad = (\uparrow_{u'}\omega, u'/u) & \text{by i.h. on } \mathcal{E}_1
\end{array}
$$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta\ \mathsf{wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

$$
\begin{array}{lr}
(\uparrow_\alpha \omega) \circ \mathrm{id}_{\Omega', \alpha\in\delta} & \\
\quad = (\uparrow_\alpha \omega) \circ (\uparrow_\alpha \mathrm{id}_{\Omega'}, \alpha/\alpha) & \text{by Definition of id} \\
\quad = \omega\circ\ \uparrow_\alpha \mathrm{id}_{\Omega'} & \text{by Composition (Def. A.2.3)} \\
\quad = \uparrow_\alpha (\omega \circ \mathrm{id}_{\Omega'}) & \text{by Composition (Def. A.2.3)} \\
\quad = \uparrow_\alpha \omega & \text{by i.h. on } \mathcal{E}_1
\end{array}
$$

□

Lemma B.16.6 (id Swap in Composition)**.**
If $\Omega' \vdash \omega : \Omega$*, then* $(\mathrm{id}_\Omega \circ \omega) = (\omega \circ \mathrm{id}_{\Omega'})$.

Proof. This follows directly from Lemmas B.16.4 and B.16.5. □

Lemma B.16.7 (Existence of Certain Substitutions)**.**
If $\Omega' \vdash \omega : \Omega$*, then*

- $(\uparrow_{\alpha'} \omega, \alpha'/\alpha') \circ (\uparrow_{\alpha_0} \mathrm{id}_{\Omega'}, \alpha_0/\alpha')$ *exists.*
- $(\omega + \overline{\alpha \in \delta}) \circ (\mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha})$ *exists.*
- $(\mathrm{id}_\Omega, e/\alpha) \circ \omega$ *exists.*
- $(\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}) \circ \omega$ *exists.*

Proof. Direct (each part shown below as separate cases).

Case: $(\uparrow_{\alpha'} \omega, \alpha'/\alpha') \circ (\uparrow_{\alpha_0} \mathrm{id}_{\Omega'}, \alpha_0/\alpha')$

$$
\begin{aligned}
&(\uparrow_{\alpha'} \omega, \alpha'/\alpha') \circ (\uparrow_{\alpha_0} \mathrm{id}_{\Omega'}, \alpha_0/\alpha') \\
&\quad = \uparrow_{\alpha_0} (\omega \circ \mathrm{id}_{\Omega'}), \alpha_0/\alpha' \\
&\qquad\qquad \text{by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)} \\
&\quad = \uparrow_{\alpha_0} \omega, \alpha_0/\alpha' \qquad \text{by Lemma B.16.5}
\end{aligned}
$$

Case: $(\omega + \overline{\alpha \in \delta}) \circ (\mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha})$

$$
\begin{aligned}
&(\omega + \overline{\alpha \in \delta}) \circ (\mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha}) \\
&\quad = (\omega \circ \mathrm{id}_{\Omega'}), (\overline{f}[\omega])/\overline{\alpha} \\
&\qquad\qquad \text{by induction on } \overline{\alpha} \text{ using Composition (Def. A.2.3)} \\
&\qquad\qquad \text{and Subst. App. (Def. A.2.4)} \\
&\quad = \omega, (\overline{f}[\omega])/\overline{\alpha} \qquad \text{by Lemma B.16.5}
\end{aligned}
$$

Case: $(\mathrm{id}_\Omega, e/\alpha) \circ \omega$

Without loss of generality, assume $\omega = \uparrow_{\overline{\alpha'}} \omega'$ where $\omega' \neq \uparrow_{\alpha''} \omega''$

$$
\begin{aligned}
&(\mathrm{id}_\Omega, e/\alpha) \circ \omega \\
&\quad = \uparrow_{\overline{\alpha'}} ((\mathrm{id}_\Omega \circ \omega), e[\omega]/\alpha) \\
&\qquad\qquad \text{by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)} \\
&\quad = \uparrow_{\overline{\alpha'}} (\omega, e[\omega]/\alpha) \qquad \text{by Lemma B.16.4}
\end{aligned}
$$

Case: $(\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}) \circ \omega$

Without loss of generality, assume $\omega = \uparrow_{\overline{\alpha'}} \omega'$ where $\omega' \neq \uparrow_{\alpha''} \omega''$

$(\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}) \circ \omega$

$= \uparrow_{\overline{\alpha'}} ((\mathrm{id}_\Omega \circ \omega), \overline{e}[\omega]/\overline{\alpha})$

by Composition (Def. A.2.3) and Subst. App. (Def. A.2.4)

$= \uparrow_{\overline{\alpha'}} (\omega, e[\omega]/\alpha)$ by Lemma B.16.4

□

Lemma B.16.8 (Substitution Applied to Declaration List)**.**
$(\overline{\alpha'\in\delta'};\alpha\in\delta)[\omega] = (\overline{\alpha'\in\delta'})[\omega];\alpha\in(\delta[\omega + \overline{\alpha'\in\delta'}])$.

Proof. By induction on $\overline{\alpha'\in\delta'}$.

Case: *nil*

$$
\begin{array}{ll}
(\alpha\in\delta)[\omega] & \\
\quad = \alpha\in\delta[\omega] & \text{by Subst. App. (Def. A.2.4)} \\
\quad = \alpha\in\delta[\omega + \mathit{nil}] & \text{by Subst. App. (Def. A.2.4)}
\end{array}
$$

Case: $\alpha_2\in\delta_2;\overline{\alpha'\in\delta'}$

$$
\begin{array}{ll}
(\alpha_2\in\delta_2;\overline{\alpha'\in\delta'};\alpha\in\delta)[\omega] & \\
\quad = \alpha_2\in\delta_2[\omega];(\overline{\alpha'\in\delta'};\alpha\in\delta)[\uparrow_{\alpha_2}\omega,\alpha_2/\alpha_2] & \\
\qquad\qquad \text{by Subst. App. (Def. A.2.4)} & \\
\quad = \alpha_2\in\delta_2[\omega];\overline{\alpha'\in\delta'}[\uparrow_{\alpha_2}\omega,\alpha_2/\alpha_2];\alpha\in\delta[(\uparrow_{\alpha_2}\omega,\alpha_2/\alpha_2) + \overline{\alpha'\in\delta'}] & \\
\qquad\qquad \text{by i.h. on } \overline{\alpha'\in\delta'} & \\
\quad = \alpha_2\in\delta_2[\omega];\overline{\alpha'\in\delta'}[\uparrow_{\alpha_2}\omega,\alpha_2/\alpha_2];\alpha\in\delta[\omega + \alpha_2\in\delta_2;\overline{\alpha'\in\delta'}] & \\
\qquad\qquad \text{by Subst. App. (Def. A.2.4)} & \\
\quad = (\alpha_2\in\delta_2;\overline{\alpha'\in\delta'})[\omega];\alpha\in\delta[\omega + \alpha_2\in\delta_2;\overline{\alpha'\in\delta'}] & \\
\qquad\qquad \text{by Subst. App. (Def. A.2.4)} &
\end{array}
$$

□

Lemma B.16.9 (Substitution Equivalence w.r.t. +)**.**
$(\uparrow_{\alpha_2}(\omega + \overline{\alpha{\in}\delta}), \alpha_2/\alpha_2) = (\omega + (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2))$.

Proof. By induction on $\overline{\alpha{\in}\delta}$.

Case: $\cdot$

$$
\begin{array}{ll}
(\uparrow_{\alpha_2}(\omega + \mathit{nil}), \alpha_2/\alpha_2) & \\
\quad = (\uparrow_{\alpha_2}\omega, \alpha_2/\alpha_2) & \text{by Subst. App. (Def. A.2.4)} \\
\quad = (\omega + (\alpha_2{\in}\delta_2)) & \text{by Subst. App. (Def. A.2.4)}
\end{array}
$$

Case: $\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}$

$$
\begin{array}{ll}
(\uparrow_{\alpha_2}(\omega + \alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}), \alpha_2/\alpha_2) & \\
\quad = (\uparrow_{\alpha_2}((\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1) + \overline{\alpha{\in}\delta}), \alpha_2/\alpha_2) & \text{by Subst. App. (Def. A.2.4)} \\
\quad = (\uparrow_{\alpha_1}\omega, \alpha_1/\alpha_1) + (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2) & \text{by i.h. on } \overline{\alpha{\in}\delta} \\
\quad = \omega + (\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2) & \text{by Subst. App. (Def. A.2.4)}
\end{array}
$$

□

Lemma B.16.10 (Substitution Composition Equivalence w.r.t. $+$)**.**
If $\Omega' \vdash \omega : \Omega$ *and* $\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$,
then $(\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}) \circ \omega = (\omega + \overline{\alpha \in \delta}) \circ (\mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha})$.

Proof. By induction on $\overline{\alpha \in \delta}$.

Case: *nil*

$\Omega' \vdash \omega : \Omega$ by assumption
$\Omega \vdash \mathrm{id}_\Omega : \Omega$ by assumption
$\mathrm{id}_\Omega \circ \omega$

$= \omega \circ \mathrm{id}_{\Omega'}$ by Lemma B.16.6
$= (\omega + \mathit{nil}) \circ \mathrm{id}_{\Omega'}$ by Subst. App. (Def. A.2.4)

Case: $\overline{\alpha \in \delta}; \alpha_2 \in \delta_2$

$\Omega' \vdash \omega : \Omega$ by assumption
$\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha}, e_2/\alpha_2 : \Omega, \overline{\alpha \in \delta}, \alpha_2 \in \delta_2$ by assumption
$\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$ by inversion using tpSubInd

$(\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}, e_2/\alpha_2) \circ \omega$
$= ((\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}) \circ \omega), e_2[\omega]/\alpha_2$ by Composition (Def. A.2.3)
$= ((\omega + \overline{\alpha \in \delta}) \circ (\mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha})), e_2[\omega]/\alpha_2$ by i.h. on $\overline{\alpha \in \delta}$
$= (\uparrow_{\alpha_2} (\omega + \overline{\alpha \in \delta}), \alpha_2/\alpha_2) \circ (\mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha}, e_2[\omega]/\alpha_2)$
by Composition (Def. A.2.3)
and Subst. Application (Def. A.2.4)
$= (\omega + (\overline{\alpha \in \delta}; \alpha_2 \in \delta_2)) \circ (\mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha}, e_2[\omega]/\alpha_2)$
by Lemma B.16.9

□

Lemma B.16.11 (Dot1 Substitution)**.**
If $\Omega \vdash \delta$ wff *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega', \alpha{\in}\delta[\omega] \vdash \uparrow_\alpha \omega, \alpha/\alpha : \Omega, \alpha{\in}\delta$*.*

Proof.

$\Omega \vdash \delta$ wff	by assumption
$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash \delta[\omega]$ wff	by Lemma B.6.1
$(\Omega', \alpha{\in}\delta[\omega])$ ctx	by ctxAdd
$\Omega', \alpha{\in}\delta[\omega] \vdash \alpha \in \delta[\omega]$	by Lemma B.5.1
$\Omega', \alpha{\in}\delta[\omega] \vdash \uparrow_\alpha \omega : \Omega$	by tpSubShift
$\delta[\omega] = \delta[\uparrow_\alpha \omega]$	by Lemma B.5.4
$\Omega', \alpha{\in}\delta[\omega] \vdash \uparrow_\alpha \omega, \alpha/\alpha : \Omega, \alpha{\in}\delta$	by tpSubInd

□

Lemma B.16.12 (Dot1 Substitution Extended for Lists)**.**
If $(\Omega, \overline{\alpha{\in}\delta})$ ctx *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega', \overline{\alpha{\in}\delta}[\omega] \vdash (\omega + \overline{\alpha{\in}\delta}) : \Omega, \overline{\alpha{\in}\delta}$*.*

Proof. By induction on $\overline{\alpha{\in}\delta}$.

Case: *nil*

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega' \vdash (\omega + \mathit{nil}) : \Omega$	by Subst. Application (Def. A.2.4)

Case: $\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2$

$\Omega' \vdash \omega : \Omega$	by assumption
$(\Omega, \overline{\alpha{\in}\delta}, \alpha_2{\in}\delta_2)$ ctx	by assumption
$(\Omega, \overline{\alpha{\in}\delta})$ ctx	by inversion using ctxAdd
$\Omega, \overline{\alpha{\in}\delta} \vdash \delta_2$ wff	by inversion using ctxAdd
$\Omega', \overline{\alpha{\in}\delta}[\omega] \vdash (\omega + \overline{\alpha{\in}\delta}) : \Omega, \overline{\alpha{\in}\delta}$	by i.h. on $\overline{\alpha{\in}\delta}$
$\Omega', \overline{\alpha{\in}\delta}[\omega], \alpha_2{\in}\delta_2[\omega + \overline{\alpha{\in}\delta}] \vdash \uparrow_{\alpha_2} (\omega + \overline{\alpha{\in}\delta}), \alpha_2/\alpha_2 : \Omega, (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2)$	by Lemma B.16.11
$\Omega', \overline{\alpha{\in}\delta}[\omega], \alpha_2{\in}\delta_2[\omega + \overline{\alpha{\in}\delta}] \vdash (\omega + (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2)) : \Omega, (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2)$	by Lemma B.16.9
$\Omega', (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2)[\omega] \vdash (\omega + (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2)) : \Omega, (\overline{\alpha{\in}\delta}; \alpha_2{\in}\delta_2)$	by Lemma B.16.8

□

Lemma B.16.13 (Well-Formed Context Property over $\forall$)**.**
If $\Omega \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}$ *and* $\Omega\ \mathsf{ctx}$*, then* $(\Omega, \overline{\alpha{\in}\delta})\ \mathsf{ctx}$.

Proof. By induction on $\mathcal{E} :: \Omega \vdash \forall\alpha{\in}\delta.\ \tau\ \mathsf{wff}$.

Case: $\mathcal{E} = \dfrac{\Omega \vdash \tau\ \mathsf{wff}}{\Omega \vdash \forall \mathit{nil}.\ \tau\ \mathsf{wff}}\ \forall_b\mathsf{wff}$

$\Omega\ \mathsf{ctx}$	by assumption

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash \delta_1\ \mathsf{wff} \qquad \Omega, \alpha_1{\in}\delta_1 \vdash \forall\overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \forall\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}}\ \forall_i\mathsf{wff}$

$\Omega\ \mathsf{ctx}$	by assumption
$\Omega \vdash \delta_1\ \mathsf{wff}$	by assumption
$(\Omega, \alpha_1{\in}\delta_1)\ \mathsf{ctx}$	by ctxAdd
$(\Omega, (\alpha_1{\in}\delta_1; \overline{\alpha{\in}\delta}))\ \mathsf{ctx}$	by i.h. on $\mathcal{E}_1$

□

B.17 Meta-Theory: Substitution Preserves Typing of Expressions

Lemma B.17.1 (Substitution Property over Global Parameter Coverage)**.**
If $\Omega \gg^{world} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ *and* $\Omega' \vdash \omega : \Omega$*,*
then $\Omega' \gg^{world} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$*.*
Warning: *Notice that coverage does not rely on cases being well-typed.*

Proof. By induction lexicographically on $\mathcal{E} :: \Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ and $\mathcal{F} :: \Omega' \vdash \omega : \Omega$.

Case: $\mathcal{E} = \Omega \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$

$$\text{and } \mathcal{F} = \dfrac{\mathcal{F}_1 \quad \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta\ \mathsf{wff}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$$

$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by gcSkipNonParameter

Case: $\mathcal{E} = \dfrac{}{\cdot \gg^{\text{world}} \overline{c}\ \text{covers}\ \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})).\ \tau}\ \mathsf{gcEmpty}$

$$\text{and } \mathcal{F} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$$

$\cdot \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by gcEmpty

Case:

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ (\boldsymbol{B_x} \approx \boldsymbol{B_c}) \text{ do not unify}\end{array}}{\Omega, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \mathsf{gcSkipMismatch}$$

$$\text{and } \mathcal{F} = \frac{\begin{array}{c}\mathcal{F}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash (\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{y'}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{y'}} \omega, \boldsymbol{y'}/\boldsymbol{y}) : (\Omega, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#})}\ \mathsf{tpSubIndNew}$$

$\Omega' \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$

$(\boldsymbol{B_x} \approx \boldsymbol{B_c})$ do not unify — by assumption

$(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}[\omega] = (\boldsymbol{\Pi\Gamma'_c.\ B_c[\gamma]})^{\#}$ — For some Γ'_c and $\boldsymbol{\gamma}$ by Subst. App. (Def. A.2.4)

$(\boldsymbol{B_x} \approx \boldsymbol{B_c}[\boldsymbol{\gamma}])$ do not unify — by above (otherwise, they would have unified)

$\Omega', \boldsymbol{y'}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}[\omega] \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau$ by gcSkipMismatch

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1 :: \Omega \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}{\Omega, \alpha{\in}\delta \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \mathsf{gcSkipNonParameter}$

$$\text{and } \mathcal{F} = \frac{\begin{array}{c}\mathcal{F}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$$

$\Omega' \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$

Case:

$$\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{E}_1 :: \Omega \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \mathcal{W} \text{ world} \\ \mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau \\ (\Omega, (\boldsymbol{\Pi\Gamma_c.\ B_c})) \in \mathcal{W}\end{array}}{\Omega, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcSkipWorld}$$

$$\text{and } \mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash (\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}[\omega] \text{ wff}\end{array}}{(\Omega', \boldsymbol{y'}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{y'}} \omega, \boldsymbol{y'}/\boldsymbol{y}) : (\Omega, \boldsymbol{y}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#})}\ \textsf{tpSubIndNew}$$

$\Omega' \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\mathcal{W}$ world — by assumption
$\mathcal{W} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ — by assumption
$(\Omega, (\boldsymbol{\Pi\Gamma_c.\ B_c})) \in \mathcal{W}$ — by assumption
$(\Omega', (\boldsymbol{\Pi\Gamma_c.\ B_c})[\omega]) \in \mathcal{W}$ — by Lemma B.12.6
$(\boldsymbol{\Pi\Gamma_c.\ B_c})[\omega]^{\#} = (\boldsymbol{\Pi\Gamma'_c.\ B_c[\gamma]})^{\#}$ — For some $\boldsymbol{\Gamma'_c}$ and $\boldsymbol{\gamma}$ by Subst. App. (Def. A.2.4)
$\Omega', \boldsymbol{y'}\overset{\triangledown}{\in}(\boldsymbol{\Pi\Gamma_c.\ B_c})^{\#}[\omega] \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau$ by gcSkipWorld

Case:

$$\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{E}_1 :: \Omega \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ \mathcal{W}_2 \text{ world} \\ \mathcal{W}_2 \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau\end{array}}{\Omega, u\overset{\triangledown}{\in}\mathcal{W} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau}\ \textsf{gcCheckWorld}$$

$$\text{and } \mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W} \text{ world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W})}\ \textsf{tpSubWorld}$$

$\Omega' \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\mathcal{W} \leq \mathcal{W}_2$ — by assumption
$\mathcal{W}_2$ world — by assumption
$\mathcal{W}_2 \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ — by assumption
$\Omega', u'\overset{\triangledown}{\in}\mathcal{W} \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x}{\in}(\boldsymbol{\Pi\Gamma_x.\ B_x})).\ \tau$ by gcCheckWorld

□

Lemma B.17.2 (Equality of Patterns under Substitutions)**.**
Let $c = \epsilon\Omega.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto f)$.
Let $c_{\text{unit}} = \epsilon\Omega.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto ())$.
If $c_{\text{unit}} = c_{\text{unit}}[\cdot]$ (i.e. c_{unit} makes sense in the empty context)
and $c[\omega]$ exists, then $c[\omega] = \epsilon\Omega.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto f')$.

Proof.

$c = \epsilon\Omega.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto f)$ by assumption
$c_{\text{unit}} = \epsilon\Omega.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto ())$ by assumption
$c_{\text{unit}} = c_{\text{unit}}[\cdot]$ by assumption
$c[\omega]$ exists by assumption
$\cdot \leq \omega$ by Lemma B.13.5
$c_{\text{unit}}[\omega]$
$= c_{\text{unit}}[\cdot]$ by Lemma B.7.6
$= \epsilon\Omega.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto ())$ by above
$c[\omega] = \epsilon\Omega.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto f')$
by Subst. App. (Def A.2.4) noting that c would behave the same as c_{unit} except for f, which we generalize as going to f'

□

Lemma B.17.3 (Equality of Function Arguments under Substitutions)**.**
Let $\tau = \nabla\mathbf{\Gamma}.\ (\forall\overline{\alpha\in\delta}.\ \sigma)$[4]
Let $\tau_{\text{unit}} = \nabla\mathbf{\Gamma}.\ (\forall\overline{\alpha\in\delta}.\ \text{unit})$
If $\tau_{\text{unit}} = \tau_{\text{unit}}[\cdot]$ (i.e. τ_{unit} makes sense in the empty context)
and $\tau[\omega]$ exists, then $\tau[\omega] = \nabla\mathbf{\Gamma}.\ (\forall\overline{\alpha\in\delta}.\ \sigma')$.

Proof.

$\tau = \nabla\mathbf{\Gamma}.\ (\forall\overline{\alpha\in\delta}.\ \sigma)$ by assumption
$\tau_{\text{unit}} = \nabla\mathbf{\Gamma}.\ (\forall\overline{\alpha\in\delta}.\ \text{unit})$ by assumption
$\tau_{\text{unit}} = \tau_{\text{unit}}[\cdot]$ by assumption
$\tau[\omega]$ exists by assumption
$\cdot \leq \omega$ by Lemma B.13.5
$\tau_{\text{unit}}[\omega]$
$= \tau_{\text{unit}}[\cdot]$ by Lemma B.7.6
$= \nabla\mathbf{\Gamma}.\ (\forall\overline{\alpha\in\delta}.\ \text{unit})$ by above
$\tau[\omega] = \nabla\mathbf{\Gamma}.\ (\forall\overline{\alpha\in\delta}.\ \sigma')$
by Subst. App. (Def A.2.4) noting that τ would behave the same as τ_{unit} except for σ, which we generalize as going to σ'

□

[4]Notice that all functions have a type of this form.

Lemma B.17.4 (Substitution Property over LF Coverage)**.**
If Ω_A ctx *and for all* $_{c_i \in \overline{c}}(\Omega \vdash c_i \in \nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau))$ *and* $\Omega' \vdash \omega : \Omega$ *and* $\omega \leq \omega'$ *and* $\overline{c}[\omega']$ *exists, then* $(\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau))[\omega] = \nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau')$ *and*

1. *if* $\mathbf{\Sigma} \gg \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau)$*, then* $\mathbf{\Sigma} \gg \overline{c}[\omega']$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau')$.
2. *if* $\Omega \gg^{world} \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau)$*, then* $\Omega \gg^{world} \overline{c}[\omega']$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau')$.
3. *if* $\mathbf{\Gamma} \gg \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau)$*, then* $\mathbf{\Gamma} \gg \overline{c}[\omega']$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau')$.
4. *if* $\mathcal{W} \gg \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau)$*, then* $\mathcal{W} \gg \overline{c}[\omega']$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau')$.
5. *if* $\mathbf{\Gamma} \gg^{\nabla} \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau)$*, then* $\mathbf{\Gamma} \gg^{\nabla} \overline{c}[\omega']$ covers $\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau')$.

Proof. The general idea of this proof is that the cases we check are designed so that the patterns make sense with respect to the empty context. Therefore, the application of any substitution does not affect the part of the cases which are meaningful to coverage, and hence coverage still holds.

The coverage rules check for the inclusion of certain cases, which all are of the form: $g = \epsilon\Omega'.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto f)$. Furthermore, since Ω_A ctx and by design of the cases, we know the patterns are well-typed in the empty context. More precisely:

$\Omega \vdash \nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau)$ wff by Lemma B.8.1
$\Omega' \vdash (\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau))[\omega]$ wff by Lemma B.6.1
$\cdot \vdash \nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \text{unit})$ wff by well-formedness rules
$(\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \text{unit}))[\cdot] = \nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \text{unit})$ by Def. of id and Lemma B.5.2
$(\nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau))[\omega] = \nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \tau')$ by Lemma B.17.3
Let $g_{\text{unit}} = \epsilon\Omega'.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto ())$
$\cdot \vdash g_{\text{unit}} \in \nabla\mathbf{\Gamma}.\ (\forall\Omega_A.\ \text{unit})$
by Design and Typing Rules.
g is from either $\mathsf{scUnify}$, $\mathsf{lcUnify}$, $\mathsf{wcUnify}$, or $\mathsf{pfUnify}$.
By Substitution Application (Def. A.2.4) the
most-general-unifiers are casted to LF (using Lemma B.4.7)
and then typing occurs on the LF level utilizing LF-level
properties such as Lemma B.1.12.
$g_{\text{unit}}[\text{id}.] = g_{\text{unit}}[\cdot] = g_{\text{unit}}$ by Def. of id and Lemma B.5.3
$g[\omega'] = \epsilon\Omega'.\ \nu\mathbf{\Gamma}.\ (\overline{e} \mapsto f')$ by Lemma B.17.2
Therefore, g is equivalent to $g[\omega']$ on all parts that are meaningful
to coverage, which is also true for the resulting type.
Therefore, the coverage condition holds by exactly the same rules.

□

Lemma B.17.5 (Substitution Property over Coverage)**.**
If $\Omega \vdash \overline{c}$ covers τ *and for all* $_{c_i \in \overline{c}}(\Omega \vdash c_i \in \tau)$ *and* $\Omega \vdash \tau$ wff *and* $\Omega' \vdash \omega : \Omega$ *and* $\omega \leq \omega'$ *and* $\overline{c}[\omega']$ *exists,*
then $\Omega' \vdash \overline{c}[\omega']$ covers $\tau[\omega]$.

Proof. By induction on $\mathcal{E} : \Omega \vdash \overline{c}$ covers τ. The general idea is that the construction of cases in the coverage rules, the patterns are not affected by substitution, so coverage still holds. The only actual inductive case is coverPop.

Case: $\mathcal{E} = \dfrac{c = \epsilon\alpha{\in}\delta.\ \alpha \mapsto f}{\Omega \vdash c \text{ covers } \forall\alpha{\in}\delta.\ \tau}$ coverSimple

$\Omega \vdash c \in \forall\alpha{\in}\delta.\ \tau$	by assumption
$\Omega \vdash \forall\alpha{\in}\delta.\ \tau$ wff	by assumption
$\Omega' \vdash \omega : \Omega$	by assumption
$\omega \leq \omega'$	by assumption
$c[\omega']$ exists	by assumption
$\Omega \vdash \forall\alpha{\in}\delta.\ \tau$ wff	by Lemma B.8.1
$\Omega' \vdash (\forall\alpha{\in}\delta.\ \tau)[\omega]$ wff	by Lemma B.6.1
$(\forall\alpha{\in}\delta.\ \tau)[\omega] = (\forall\alpha{\in}\delta.\ \tau)[\omega']$	by Lemma B.7.4
$c = \epsilon\alpha{\in}\delta.\ \alpha \mapsto f$	by assumption (premise)
$(\forall\alpha{\in}\delta.\ \tau)[\omega'] = \forall\alpha{\in}\delta[\omega'].\ (\tau[\uparrow_\alpha \omega', \alpha/\alpha])$	by Subst. App. (Def. A.2.4)
$c[\omega'] = \epsilon\alpha{\in}\delta[\omega'].\ \alpha \mapsto f'$	by Subst. App. (Def. A.2.4)
$\Omega \vdash c[\omega']$ covers $\forall\alpha{\in}\delta[\omega'].\ \tau[\uparrow_\alpha \omega', \alpha/\alpha]$	by coverSimple
$\Omega \vdash c[\omega']$ covers $(\forall\alpha{\in}\delta.\ \tau)[\omega]$	by above

Case:

$$\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau\ \mathsf{wff} \\ c = \epsilon \boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y}\ \boldsymbol{\Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y}\ \boldsymbol{\Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}$$

$\Omega \vdash c \in (\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma$ by assumption
$\Omega \vdash (\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma\ \mathsf{wff}$ by assumption
$\Omega' \vdash \omega : \Omega$ by assumption
$\omega \leq \omega'$ by assumption
$c[\omega']$ exists by assumption
$\Omega \vdash (\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma\ \mathsf{wff}$ by Lemma B.8.1
$\Omega' \vdash ((\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma)[\omega]\ \mathsf{wff}$ by Lemma B.6.1
$\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau\ \mathsf{wff}$ by assumption (premise)
$c = \epsilon \boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y}\ \boldsymbol{\Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y}\ \boldsymbol{\Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f$ by assumption (premise)

Let c_{unit}
$\quad = \epsilon \boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y}\ \boldsymbol{\Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y}\ \boldsymbol{\Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto ()$
$\cdot \vdash c_{\mathrm{unit}} \in (\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \mathrm{unit}$ by Typing Rules

Technically the rule pairl would want
$u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}, \boldsymbol{y}:(\boldsymbol{\Pi\Gamma.\ A})}, (\boldsymbol{y}\ \boldsymbol{\Gamma})/\boldsymbol{x}])$
However, we note that this is equivalent to our
declaration of u as $\boldsymbol{y}$ does not occur free in τ.

$c_{\mathrm{unit}}[\mathrm{id}_\cdot] = c_{\mathrm{unit}}[\cdot] = c_{\mathrm{unit}}$ by Def. of id and Lemma B.5.3
$c[\omega'] = \epsilon \boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y}\ \boldsymbol{\Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y}\ \boldsymbol{\Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f'$ by Lemma B.17.2
$\cdot \vdash ((\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \mathrm{unit})\ \mathsf{wff}$ by well-formedness rules
$((\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \mathrm{unit})[\cdot] = (\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \mathrm{unit}$ by Def. of id and Lemma B.5.2
$((\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma)[\omega] = (\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma'$ by Lemma B.17.3
$\Omega' \vdash c[\omega'] \text{ covers } ((\nabla\boldsymbol{\Gamma}.\ \exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma)[\omega]$ by coverPairLF

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff} \\ c = \epsilon \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}^{\#}$

$\Omega \vdash c \in (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma$	by assumption
$\Omega \vdash (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma\ \mathsf{wff}$	by assumption
$\Omega' \vdash \omega : \Omega$	by assumption
$\omega \leq \omega'$	by assumption
$c[\omega']$ exists	by assumption
$\Omega \vdash (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma\ \mathsf{wff}$	by Lemma B.8.1
$\Omega' \vdash ((\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma)[\omega]\ \mathsf{wff}$	by Lemma B.6.1
$\cdot \vdash \exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}$	by assumption (premise)
$c = \epsilon \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto f$	by assumption (premise)
Let $c_{\text{unit}} = \epsilon \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto ()$	
$\cdot \vdash c_{\text{unit}} \in (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \text{unit}$	by Typing Rules and Subst. App (Def. A.2.4)
$c_{\text{unit}}[\text{id}.] = c_{\text{unit}}[\cdot] = c_{\text{unit}}$	by Def. of id and Lemma B.5.3
$c[\omega'] = \epsilon \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto f'$	by Lemma B.17.2
$\cdot \vdash ((\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \text{unit})\ \mathsf{wff}$	by well-formedness rules
$((\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \text{unit})[\cdot] = (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \text{unit}$	by Def. of id and Lemma B.5.2
$((\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma)[\omega] = (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma'$	by Lemma B.17.3
$\Omega' \vdash c[\omega']$ covers $((\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma)[\omega]$	by $\mathsf{coverPairLF}^{\#}$

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)\ \mathsf{wff} \\ c = \epsilon u_1{\in}(\nabla\mathbf{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla\mathbf{\Gamma}.\ \tau_2).\ (\nu\mathbf{\Gamma}.\ (u_1\backslash\mathbf{\Gamma},\ u_2\backslash\mathbf{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c \text{ covers } (\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma}\ \mathsf{coverPairMeta}$

$\Omega \vdash c \in (\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma$ by assumption
$\Omega \vdash (\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma\ \mathsf{wff}$ by assumption
$\Omega' \vdash \omega : \Omega$ by assumption
$\omega \leq \omega'$ by assumption
$c[\omega']$ exists by assumption
$\Omega \vdash (\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma\ \mathsf{wff}$ by Lemma B.8.1
$\Omega' \vdash ((\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma)[\omega]\ \mathsf{wff}$ by Lemma B.6.1
$\cdot \vdash \nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)\ \mathsf{wff}$ by assumption (premise)
$c = \epsilon u_1{\in}(\nabla\mathbf{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla\mathbf{\Gamma}.\ \tau_2).\ (\nu\mathbf{\Gamma}.\ (u_1\backslash\mathbf{\Gamma},\ u_2\backslash\mathbf{\Gamma})) \mapsto f$ by assumption (premise)
Let $c_{\text{unit}} = \epsilon u_1{\in}(\nabla\mathbf{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla\mathbf{\Gamma}.\ \tau_2).\ (\nu\mathbf{\Gamma}.\ (u_1\backslash\mathbf{\Gamma},\ u_2\backslash\mathbf{\Gamma})) \mapsto ()$
$\cdot \vdash c_{\text{unit}} \in (\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \text{unit}$ by Typing Rules and Subst. App (Def. A.2.4)
$c_{\text{unit}}[\text{id.}] = c_{\text{unit}}[\cdot] = c_{\text{unit}}$ by Def. of id and Lemma B.5.3
$c[\omega'] = \epsilon u_1{\in}(\nabla\mathbf{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla\mathbf{\Gamma}.\ \tau_2).\ (\nu\mathbf{\Gamma}.\ (u_1\backslash\mathbf{\Gamma},\ u_2\backslash\mathbf{\Gamma})) \mapsto f'$ by Lemma B.17.2
$\cdot \vdash ((\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \text{unit})\ \mathsf{wff}$ by well-formedness rules
$((\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \text{unit})[\cdot] = (\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \text{unit}$ by Def. of id and Lemma B.5.2
$((\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma)[\omega] = (\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma'$ by Lemma B.17.3
$\Omega' \vdash c[\omega'] \text{ covers } ((\nabla\mathbf{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma)[\omega]$ by coverPairMeta

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Omega_1 \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ \tau} \qquad (\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2)}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c} \backslash \boldsymbol{x} \text{ covers } \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$

for all $_{c_i \in \overline{c}}(\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash c_i \backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}])$ by assumption
$\Omega \vdash \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$ wff by assumption
$\Omega' \vdash \omega : \Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2$ by assumption
$\omega \leq \omega'$ by assumption
$(\overline{c} \backslash \boldsymbol{x})[\omega']$ exists by assumption
for all $_{c_i \in \overline{c}}(\Omega_1 \vdash c_i \in \nabla \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ \tau)$ by inversion using pop
$(\overline{c} \backslash \boldsymbol{x})[\omega'] = (\overline{c}[\omega']) \backslash (\boldsymbol{x}[\omega'])$ by Subst. App. (Def. A.2.4)
$\Omega \vdash \nabla \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ \tau$ wff
by Lemma B.8.1 since $\overline{c}$ is nonempty (Abbreviations A.2.5)
Ω' ctx by Lemma B.3.2
There exists an Ω_1', Ω_2', $\boldsymbol{x_0}$, and ω_1 such that
$\Omega_1', \boldsymbol{x_0} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) : \Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$
and $\Omega' = \Omega_1', \boldsymbol{x_0} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega_1], \Omega_2'$
and $\omega_1 \leq \omega$ and $(\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq \omega$
and $(\Omega_1', \boldsymbol{x_0} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega_1]) \leq (\Omega_1', \boldsymbol{x_0} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega_1], \Omega_2')$. by Lemma B.16.1
$\boldsymbol{x}[\omega'] = \boldsymbol{x_0}$ by above and Subst. App. (Def. A.2.4)
$\Omega_1' \vdash \omega_1 : \Omega_1$ by inversion using tpSubIndNew
$\omega_1 \leq \omega'$ by Lemma B.13.4
$\Omega_1' \vdash \overline{c}[\omega'] \text{ covers } \nabla \boldsymbol{x'} \in \boldsymbol{A}^{\#}[\omega_1].\ \tau[\uparrow_{\boldsymbol{x'}} \omega_1, \boldsymbol{x'}/\boldsymbol{x'}]$
by i.h. on $\mathcal{E}_1$ and Subst. App. (Def A.2.4)
$\Omega_1', \boldsymbol{x_0} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega_1], \Omega_2' \vdash (\overline{c}[\omega']) \backslash \boldsymbol{x_0} \text{ covers } \tau[\uparrow_{\boldsymbol{x'}} \omega_1, \boldsymbol{x'}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}, \boldsymbol{x_0}/\boldsymbol{x'}]$
by coverPop

$\tau[\uparrow_{\boldsymbol{x'}} \omega_1, \boldsymbol{x'}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}, \boldsymbol{x_0}/\boldsymbol{x'}]$
$= \tau[(\uparrow_{\boldsymbol{x'}} \omega_1, \boldsymbol{x'}/\boldsymbol{x'}) \circ (\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}, \boldsymbol{x_0}/\boldsymbol{x'})]$ by Lemmas B.12.1 and B.16.7
$= \tau[(\omega_1 \circ \uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Composition (Def. A.2.3) and Subst. Application (Def. A.2.4)
$= \tau[\uparrow_{\boldsymbol{x_0}} (\omega_1 \circ \mathrm{id}_{\Omega_1'}), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Composition (Def. A.2.3)
$= \tau[(\omega_1 \circ \mathrm{id}_{\Omega_1'}), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Lemma B.5.4

(Case Continued →)

$= \tau[(\mathrm{id}_{\Omega_1} \circ \omega_1), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Lemma B.16.6
$= \tau[\uparrow_{\boldsymbol{x_0}} (\mathrm{id}_{\Omega_1} \circ \omega_1), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Lemma B.5.4
$= \tau[(\mathrm{id}_{\Omega_1} \circ \uparrow_{\boldsymbol{x_0}} \omega_1), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Composition (Def. A.2.3)
$= \tau[(\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}) \circ (\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x})]$ by Composition (Def. A.2.3) and Subst. Application (Def. A.2.4)
$= \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}]$ by Lemma B.12.1

$\Omega_1', \boldsymbol{x_0} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}[\omega_1], \Omega_2' \vdash (\overline{c}[\omega'])\backslash \boldsymbol{x_0}$ covers $\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}]$ by above

$\Omega' \vdash (\overline{c}\backslash \boldsymbol{x})[\omega']$ covers $(\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}])[\omega]$ by Lemma B.7.4

Case: $\mathcal{E} =$

$$\frac{\begin{array}{l}\mathcal{F}_1 :: (\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})\ \mathsf{ctx} \\ \mathcal{F}_2 :: \Omega_A \text{ only contains declarations of type } \boldsymbol{A} \text{ or } \boldsymbol{A}^{\#} \\ \mathcal{F}_3 :: \boldsymbol{\Sigma} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau \\ \mathcal{F}_4 :: \boldsymbol{\Gamma_x} \gg \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau \\ \mathcal{F}_5 :: \Omega \gg^{\text{world}} \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau\end{array}}{\Omega \vdash \overline{c} \text{ covers } \forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau}\ \mathsf{coverLF}$$

for all $_{c_i \in \overline{c}}(\Omega \vdash c_i \in \forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau)$
$\Omega \vdash \forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau\ \mathsf{wff}$ by assumption
$\Omega' \vdash \omega : \Omega$ by assumption
$\omega \leq \omega'$ by assumption
$c[\omega']$ exists by assumption
$(\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau)[\omega] = \forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau'$ by Lemma B.17.4 on $\mathcal{F}_3$

$\mathcal{F}_3' :: \boldsymbol{\Sigma} \gg \overline{c}[\omega']$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau'$ by Lemma B.17.4 on $\mathcal{F}_3$

$\mathcal{F}_{5a} :: \Omega' \gg^{\text{world}} \overline{c}$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau$ by Lemma B.17.1 on $\mathcal{F}_5$

$\mathcal{F}_{5b} :: \Omega' \gg^{\text{world}} \overline{c}[\omega']$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau'$ by Lemma B.17.4 on $\mathcal{F}_{5a}$

$\mathcal{F}_4' :: \boldsymbol{\Gamma_x} \gg \overline{c}[\omega']$ covers $\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau'$ by Lemma B.17.4 on $\mathcal{F}_4$

$\Omega' \vdash \overline{c}[\omega']$ covers $(\forall(\Omega_A, \boldsymbol{x} \in \boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x}).\ \tau)[\omega]$ by coverLF with $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3'$, $\mathcal{F}_4'$, and $\mathcal{F}_{5b}$

$\mathcal{F}_1 :: (\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$
$\mathcal{F}_2 ::$ All elements of Ω_A occur free in $\boldsymbol{A}$
$\mathcal{F}_3 :: g = \epsilon\Omega_A.\ \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \nu\boldsymbol{\Gamma_g}.\ (\Omega_A; \boldsymbol{x} \mapsto f)$
$\mathcal{F}_4 :: g$ in $\overline{c}$
$\mathcal{F}_5 :: \boldsymbol{\Gamma} \gg^{\nabla} \overline{c}$ covers $\nabla\boldsymbol{\Gamma}.\ (\forall\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)$

Case: $\mathcal{E} = \dfrac{}{\Omega \vdash \overline{c} \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverNewLF}^{\#}$

for all ${}_{c_i \in \overline{c}}(\Omega \vdash c_i \in \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)$
$\Omega \vdash \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau\ \mathsf{wff}$ by assumption
$\Omega' \vdash \omega : \Omega$ by assumption
$\omega \leq \omega'$ by assumption
$c[\omega']$ exists by assumption
$(\nabla\boldsymbol{\Gamma}.\ (\forall\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau))[\omega] = \nabla\boldsymbol{\Gamma}.\ (\forall\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau')$
by Lemma B.17.4 on $\mathcal{F}_5$
$\mathcal{F}_5' :: \boldsymbol{\Gamma} \gg^{\nabla} \overline{c}$ covers $\nabla\boldsymbol{\Gamma}.\ (\forall\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau')$ by Lemma B.17.4 on $\mathcal{F}_5$
Let $g_{\text{unit}} = \epsilon\Omega_A.\ \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \nu\boldsymbol{\Gamma_g}.\ (\Omega_A; \boldsymbol{x} \mapsto ())$
$\cdot \vdash g_{\text{unit}} \in \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \text{unit}$ by Typing Rules
$g_{\text{unit}}[\text{id}.] = g_{\text{unit}}[\cdot] = g_{\text{unit}}$ by Def. of id and Lemma B.5.3
$\mathcal{F}_3' :: g[\omega'] = \epsilon\Omega_A.\ \epsilon\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \nu\boldsymbol{\Gamma_g}.\ (\Omega_A; \boldsymbol{x} \mapsto f')$ by Lemma B.17.2
$\mathcal{F}_4' :: g[\omega'] \in \overline{c}[\omega']$ by above and $\mathcal{F}_4$
$\Omega' \vdash \overline{c}[\omega']$ covers $(\nabla\boldsymbol{\Gamma}.\ (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau))[\omega]$
by $\mathsf{coverNewLF}^{\#}$ with $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3'$, $\mathcal{F}_4'$, and $\mathcal{F}_5'$

Case: $\mathcal{E} = \dfrac{\cdot \leq_{\cdot} \Omega}{\Omega \vdash \mathit{nil} \text{ covers } (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverEmpty}^{\#}$

$\Omega \vdash \forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau\ \mathsf{wff}$ by assumption
$\Omega' \vdash \omega : \Omega$ by assumption
$\omega \leq \omega'$ by assumption
$\Omega\ \mathsf{ctx}$ by Lemma B.3.1
$\mathit{nil}[\omega'] = \mathit{nil}$ by Subst. App. (Def. A.2.4)
$\Omega \vdash (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)\ \mathsf{wff}$ by Lemma B.8.1
$\Omega' \vdash (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)[\omega]\ \mathsf{wff}$ by Lemma B.6.1
$\cdot \leq_{\cdot} \Omega$ by assumption (premise)
$\cdot\ \mathsf{world}$ by $\mathsf{worldEmpty}$
$\cdot \leq_{\cdot} \Omega'$ by Lemma B.13.6
$\Omega' \vdash \mathit{nil}[\omega'] \in (\forall(\Omega_A, \boldsymbol{x}{\in}\boldsymbol{A}^{\#}).\ \tau)[\omega]$
by $\mathsf{coverEmpty}^{\#}$ and Subst. App. (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type} \\ c = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}).\ \nu\boldsymbol{\Gamma}.\ ((\boldsymbol{x}\ \boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)}\ \mathsf{coverNewLF}$

$\Omega \vdash c \in \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)$	by assumption
$\Omega \vdash \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)\ \mathsf{wff}$	by assumption
$\Omega' \vdash \omega : \Omega$	by assumption
$\omega \leq \omega'$	by assumption
$c[\omega']$ exists	by assumption
$\Omega \vdash \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)\ \mathsf{wff}$	by Lemma B.8.1
$\Omega' \vdash (\nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau))[\omega]\ \mathsf{wff}$	by Lemma B.6.1
$\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type}$	by assumption (premise)
$c = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}).\ \nu\boldsymbol{\Gamma}.\ ((\boldsymbol{x} \backslash \boldsymbol{\Gamma}) \mapsto f)$	by assumption (premise)
Let $c_{\text{unit}} = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}).\ \nu\boldsymbol{\Gamma}.\ ((\boldsymbol{x} \backslash \boldsymbol{\Gamma}) \mapsto ())$	
$\cdot \vdash c_{\text{unit}} \in \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \text{unit})$	by Typing Rules and Subst. App (Def. A.2.4)
$c_{\text{unit}}[\text{id}_\cdot] = c_{\text{unit}}[\cdot] = c_{\text{unit}}$	by Def. of id and Lemma B.5.3
$c[\omega'] = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}).\ \nu\boldsymbol{\Gamma}.\ ((\boldsymbol{x} \backslash \boldsymbol{\Gamma}) \mapsto f')$	by Lemma B.17.2
$\cdot \vdash \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \text{unit})\ \mathsf{wff}$	by well-formedness rules
$(\nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \text{unit}))[\cdot] = (\nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \text{unit}))$	by Def. of id and Lemma B.5.2
$(\nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau))[\omega] = \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau')$	by Lemma B.17.3
$\Omega' \vdash c[\omega']$ covers $(\nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau))[\omega]$	by $\mathsf{coverNewLF}$

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla\mathbf{\Gamma}.\ \sigma\ \mathsf{wff} \\ c = \epsilon u{\in}(\nabla\mathbf{\Gamma}.\ \sigma).\ \nu\mathbf{\Gamma}.\ ((u\backslash\mathbf{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\mathbf{\Gamma}.\ (\sigma \supset \tau)}\ \mathsf{coverNewMeta}$

$\Omega \vdash c \in \nabla\mathbf{\Gamma}.\ (\sigma \supset \tau)$	by assumption
$\Omega \vdash \nabla\mathbf{\Gamma}.\ (\sigma \supset \tau)\ \mathsf{wff}$	by assumption
$\Omega' \vdash \omega : \Omega$	by assumption
$\omega \leq \omega'$	by assumption
$c[\omega']$ exists	by assumption
$\Omega \vdash \nabla\mathbf{\Gamma}.\ (\sigma \supset \tau)\ \mathsf{wff}$	by Lemma B.8.1
$\Omega' \vdash (\nabla\mathbf{\Gamma}.\ (\sigma \supset \tau))[\omega]\ \mathsf{wff}$	by Lemma B.6.1
$\cdot \vdash \nabla\mathbf{\Gamma}.\ \sigma\ \mathsf{wff}$	by assumption (premise)
$c = \epsilon u{\in}(\nabla\mathbf{\Gamma}.\ \sigma).\ \nu\mathbf{\Gamma}.\ ((u\backslash\mathbf{\Gamma}) \mapsto f)$	by assumption (premise)
Let $c_{\text{unit}} = \epsilon u{\in}(\nabla\mathbf{\Gamma}.\ \sigma).\ \nu\mathbf{\Gamma}.\ ((u\backslash\mathbf{\Gamma}) \mapsto ())$	
$\cdot \vdash c_{\text{unit}} \in \nabla\mathbf{\Gamma}.\ (\sigma \supset \text{unit})$	by Typing Rules and Subst. App (Def. A.2.4)
$c_{\text{unit}}[\text{id}_\cdot] = c_{\text{unit}}[\cdot] = c_{\text{unit}}$	by Def. of id and Lemma B.5.3
$c[\omega'] = \epsilon u{\in}(\nabla\mathbf{\Gamma}.\ \sigma).\ \nu\mathbf{\Gamma}.\ ((u\backslash\mathbf{\Gamma}) \mapsto f')$	by Lemma B.17.2
$\cdot \vdash \nabla\mathbf{\Gamma}.\ (\sigma \supset \text{unit})\ \mathsf{wff}$	by well-formedness rules
$(\nabla\mathbf{\Gamma}.\ (\sigma \supset \text{unit}))[\cdot] = (\nabla\mathbf{\Gamma}.\ (\sigma \supset \text{unit}))$	by Def. of id and Lemma B.5.2
$(\nabla\mathbf{\Gamma}.\ (\sigma \supset \tau))[\omega] = \nabla\mathbf{\Gamma}.\ (\sigma \supset \tau')$	by Lemma B.17.3
$\Omega' \vdash c[\omega']$ covers $(\nabla\mathbf{\Gamma}.\ (\sigma \supset \tau))[\omega]$	by coverNewMeta

□

Lemma B.17.6 (Computation-Level Variable under Substitution)**.**
If $(\Omega_1, u'{\in}\tau') \leq (\Omega_1, u'{\in}\tau', \Omega_2)$ *and* $\Omega' \vdash \omega : (\Omega_1, u'{\in}\tau', \Omega_2)$*,*
then $\Omega' \vdash u'[\omega] \in \tau'[\omega]$*.*

Proof. By induction lexicographically on Ω_2 and $\mathcal{E} :: \Omega' \vdash \omega : (\Omega_1, u'{\in}\tau', \Omega_2)$.

Case: $\cdot$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega_1 \end{array} \qquad \Omega' \vdash e \in \tau'[\omega]}{\Omega' \vdash (\omega, e/u') : \Omega_1, u'{\in}\tau'}\,\mathsf{tpSubInd}$

$u'[\omega, e/u'] = e$	by Subst. App. (Def. A.2.4)
$\Omega' \vdash e \in \tau'[\omega]$	by assumption
$\Omega' \vdash \tau'[\omega]$ wff	by Lemma B.8.1
$\omega \leq (\omega, e/u')$	by leSubAdd
$\Omega' \vdash e \in \tau'[\omega, e/u']$	by Lemma B.7.4
$\Omega' \vdash u'[\omega, e/u'] \in \tau'[\omega, e/u']$	by above

Case: $(\Omega_2, \alpha{\in}\delta)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega_1, u'{\in}\tau', \Omega_2) \end{array} \qquad \Omega' \vdash e \in \delta[\omega]}{\Omega' \vdash (\omega, e/\alpha) : (\Omega_1, u'{\in}\tau', \Omega_2, \alpha{\in}\delta)}\,\mathsf{tpSubInd}$

$(\Omega_1, u'{\in}\tau') \leq (\Omega_1, u'{\in}\tau', \Omega_2, \alpha{\in}\delta)$	by assumption
$(\Omega_1, u'{\in}\tau') \leq (\Omega_1, u'{\in}\tau', \Omega_2)$	by inversion using leAdd
$u'[\omega, e/\alpha] = u'[\omega]$	by Subst. App. (Def. A.2.4)
$\Omega' \vdash u'[\omega] \in \tau'[\omega]$	by i.h. on Ω_2 and $\mathcal{E}_1$
$\Omega' \vdash \tau'[\omega]$ wff	by Lemma B.8.1
$\omega \leq (\omega, e/\alpha)$	by leSubAdd
$\Omega' \vdash u'[\omega] \in \tau'[\omega, e/\alpha]$	by Lemma B.7.4
$\Omega' \vdash u'[\omega, e/\alpha] \in \tau'[\omega, e/\alpha]$	by above

Case: $(\Omega_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ and any $\mathcal{E}$

$(\Omega_1, u' \in \tau') \leq (\Omega_1, u' \in \tau', \Omega_2, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})$ by assumption
By inspection of the rules, this is impossible
and hence this case is vacuously true.

Case: $(\Omega_2, u \overset{\triangledown}{\in} \mathcal{W})$ and any $\mathcal{E}$

$(\Omega_1, u' \in \tau') \leq (\Omega_1, u' \in \tau', \Omega_2, u \overset{\triangledown}{\in} \mathcal{W})$ by assumption
By inspection of the rules, this is impossible
and hence this case is vacuously true.

Case: Ω_2 and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : (\Omega_1, u' \in \tau', \Omega_2) \qquad \Omega' \vdash \delta \ \mathsf{wff}\end{array}}{\Omega', \alpha \in \delta \vdash \uparrow_\alpha \omega : (\Omega_1, u' \in \tau', \Omega_2)}\ \mathsf{tpSubShift}$

$(\Omega_1, u' \in \tau') \leq (\Omega_1, u' \in \tau', \Omega_2)$ by assumption
$\Omega' \vdash u'[\omega] \in \tau'[\omega]$ by i.h. on Ω_2 and $\mathcal{E}_1$
$\Omega' \vdash u'[\uparrow_\alpha \omega] \in \tau'[\uparrow_\alpha \omega]$ by Lemma B.5.4
$\Omega' \leq \Omega', \alpha \in \delta$ by leAdd
$(\Omega', \alpha \in \delta)\ \mathsf{ctx}$ by Lemma B.3.2 on $\mathcal{E}$
$\Omega', \alpha \in \delta \vdash u'[\uparrow_\alpha \omega] \in \tau'[\uparrow_\alpha \omega]$ by Lemma B.14.4

□

Lemma B.17.7 (Main Substitution Property over Typing)**.**

- *If* $\Omega \vdash e \in \delta$ *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash e[\omega] \in \delta[\omega]$*.*
- *If* $\Omega \vdash c \in \tau$ *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash c[\omega] \in \tau[\omega]$*.*
- *If* $(\Omega, \overline{\alpha \in \delta})\ \mathsf{ctx}$ *and* $\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$ *and* $\Omega' \vdash \omega : \Omega$*, then* $\Omega' \vdash \mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha} : \Omega', \overline{\alpha \in \delta}[\omega]$*.*

Proof. By induction on e and c and $\overline{f}$.

Case: () and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx}}{\Omega \vdash () \in \mathrm{unit}}\ \mathsf{top}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega'\ \mathsf{ctx}$	by Lemma B.3.2
$\Omega' \vdash () \in \mathrm{unit}$	by top
$\Omega' \vdash ()[\omega] \in \mathrm{unit}[\omega]$	by Subst. Application (Def. A.2.4)

Case: u and $\mathcal{E} = \dfrac{(\Omega_1, u{\in}\tau, \Omega_2)\ \mathsf{ctx} \quad (\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)}{\Omega_1, u{\in}\tau, \Omega_2 \vdash u \in \tau}\ \tau\mathsf{var}$

$\Omega' \vdash \omega : (\Omega_1, u{\in}\tau, \Omega_2)$	by assumption
$(\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)$	by assumption
$\Omega' \vdash u[\omega] \in \tau[\omega]$	by Lemma B.17.6

Case: $\boldsymbol{x}$ and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad ((\boldsymbol{x}{\in}\boldsymbol{A}^\#) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#)) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^\#}\ \mathsf{var}^\#$

$\Omega' \vdash \omega : \Omega$	by assumption
$((\boldsymbol{x}{\in}\boldsymbol{A}^\#) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^\#)) \text{ in } \Omega$	by assumption
$\Omega' \vdash \boldsymbol{x}[\omega] \in \boldsymbol{A}^\#[\omega]$	by Lemma B.12.2

Case: $\boldsymbol{M}$ and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad \|\Omega\| \vdash^{\mathsf{f}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\|\Omega'\| \vdash^{\mathsf{f}} \|\omega\| : \|\Omega\|$	by Lemma B.4.7
$\|\Omega\| \vdash^{\mathsf{f}} \boldsymbol{M} : \boldsymbol{A}$	by assumption
$\|\Omega'\| \vdash^{\mathsf{f}} \boldsymbol{M}[\|\omega\|] : \boldsymbol{A}[\|\omega\|]$	by Lemma B.1.12.1
$\Omega'\ \mathsf{ctx}$	by Lemma B.3.2
$\Omega' \vdash \boldsymbol{M}[\|\omega\|] \in \boldsymbol{A}[\|\omega\|]$	by isLF
$\Omega' \vdash \boldsymbol{M}[\omega] \in \boldsymbol{A}[\omega]$	by Subst. Application (Def. A.2.4)

Case: (fn $\overline{c}$) and

$$\mathcal{E} = \frac{\Omega\ \mathsf{ctx} \quad \Omega \vdash \tau\ \mathsf{wff} \quad \text{for all } {}_{c_i \in \overline{c}}(\Omega \vdash c_i \in \tau) \quad \overset{\mathcal{E}_1}{\Omega \vdash \overline{c}\ \text{covers}\ \tau}}{\Omega \vdash \text{fn}\ \overline{c} \in \tau}\ \mathsf{impl}$$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega \vdash \tau\ \mathsf{wff}$	by assumption
$\Omega' \vdash \tau[\omega]\ \mathsf{wff}$	by Lemma B.6.1
for all ${}_{c_i \in \overline{c}}(\Omega \vdash c_i[\omega] \in \tau[\omega])$	by i.h. on c_i with $\mathcal{E}_1$ (for all $c_i \in \overline{c}$)
for all ${}_{c_i \in \overline{c}[\omega]}(\Omega \vdash c_i \in \tau[\omega])$	by Subst. App. (Def A.2.4)
$\Omega \vdash \overline{c}$ covers τ	by assumption
$\omega \leq \omega$	by $\mathsf{leSubEq}$
$\overline{c}[\omega]$ exists	by above
$\Omega' \vdash \overline{c}[\omega]$ covers $\tau[\omega]$	by Lemma B.17.5
$\Omega' \vdash \text{fn}\ (\overline{c}[\omega]) \in \tau[\omega]$	by impl
$\Omega' \vdash (\text{fn}\ \overline{c})[\omega] \in \tau[\omega]$	by Subst. Application (Def. A.2.4)

Case: $(e\ \overline{f})$ and $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash e \in \forall\overline{\alpha\in\delta}.\ \tau & \Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha\in\delta}\end{array}}{\Omega \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega \vdash \forall\overline{\alpha\in\delta}.\ \tau\ \mathsf{wff}$	by Lemma B.8.1 on $\mathcal{E}_1$
$\Omega\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$(\Omega, \overline{\alpha\in\delta})\ \mathsf{ctx}$	by Lemma B.16.13
$\Omega \vdash e[\omega] \in \forall\overline{\alpha\in\delta}[\omega].\ \tau[\omega + \overline{\alpha\in\delta}]$	by i.h. on e with $\mathcal{E}_1$ and Subst. Application (Def. A.2.4)
$\Omega' \vdash \mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha} : \Omega', \overline{\alpha\in\delta}[\omega]$	by i.h. on $\overline{f}$ with $\mathcal{E}_2$
$\Omega' \vdash (e[\omega])\ (f[\omega]) \in \tau[\omega + \overline{\alpha\in\delta}][\mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha}]$	by impE
$\Omega', \overline{\alpha\in\delta}[\omega] \vdash (\omega + \overline{\alpha\in\delta}) : \Omega, \overline{\alpha\in\delta}$	by Lemma B.16.12
$\tau[\omega + \overline{\alpha\in\delta}][\mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha}]$	
$\quad = \tau[(\omega + \overline{\alpha\in\delta}) \circ (\mathrm{id}_{\Omega'}, (\overline{f}[\omega])/\overline{\alpha})]$	by Lemmas B.12.1 and B.16.7
$\quad = \tau[(\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}) \circ \omega]$	by Lemma B.16.10
$\quad = \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}][\omega]$	by Lemma B.12.1
$\Omega' \vdash (e\ f)[\omega] \in (\tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}])[\omega]$	by Subst. Application (Def. A.2.4)

Case: $(\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau\end{array}}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{new}$

$\Omega' \vdash \omega : \Omega$	by assumption
$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$	by inversion using $\mathsf{ctxAddNew}$
$\Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}$	by Lemma B.6.1
$\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega] \vdash (\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}) : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by $\mathsf{tpSubIndNew}$
$\Omega', \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega] \vdash e[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}] \in \tau[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}]$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega].\ e[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}] \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega].\ \tau[\uparrow_{\boldsymbol{x}}\omega, \boldsymbol{x}/\boldsymbol{x}]$	by new
$\Omega' \vdash (\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)[\omega] \in (\nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau)[\omega]$	by Subst. Application (Def. A.2.4)

Case: $(\nu u{\in}\mathcal{W}.\ e)$ and $\mathcal{E} = \dfrac{\mathcal{E}_1}{\Omega \vdash \nu u{\in}\mathcal{W}.\ e \in \nabla\mathcal{W}.\ \tau}\ \mathsf{newW}$

$\Omega' \vdash \omega : \Omega$	by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W})$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\mathcal{W}$ world	by inversion using ctxAddWorld
$\Omega', u\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_u\omega, u/u) : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$	by tpSubWorld
$\Omega', u\overset{\triangledown}{\in}\mathcal{W} \vdash e[\uparrow_u\omega, u/u] \in \tau[\uparrow_u\omega, u/u]$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash \nu u{\in}\mathcal{W}.\ e[\uparrow_u\omega, u/u] \in \nabla\mathcal{W}.\ \tau[\uparrow_u\omega, u/u]$	by newW
$\Omega \vdash \nabla\mathcal{W}.\ \tau$ wff	by Lemma B.8.1 on $\mathcal{E}$
$\Omega \vdash \tau$ wff	by inversion using ∇_{world}wff
$\Omega' \vdash \tau[\omega]$ wff	by Lemma B.6.1
$\omega \leq (\uparrow_u\omega, u/u)$	by leSubShift and leSubAdd
$\tau[\omega] = \tau[\uparrow_u\omega, u/u]$	by Lemma B.7.4
$\Omega' \vdash \nu u{\in}\mathcal{W}.\ e[\uparrow_u\omega, u/u] \in \nabla\mathcal{W}.\ \tau[\omega]$	by above
$\Omega' \vdash (\nu u{\in}\mathcal{W}.\ e)[\omega] \in (\nabla\mathcal{W}.\ \tau)[\omega]$	by Subst. Application (Def. A.2.4)

Case: $(e\backslash \boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla \boldsymbol{x}'{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x}']}\ \mathsf{pop}$

$\Omega' \vdash \omega : (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by assumption

$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ by assumption

There exists an Ω_1', Ω_2', $\boldsymbol{x_0}$, and ω_1 such that

$\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) : \Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$

and $\Omega' = \Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2'$

and $\omega_1 \leq \omega$

and $(\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}) \leq \omega$

and $(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]) \leq (\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1], \Omega_2')$. by Lemma B.16.1

$\Omega_1' \vdash \omega_1 : \Omega_1$ by inversion using tpSubIndNew

$\Omega_1' \vdash e[\omega_1] \in \nabla \boldsymbol{x}'{\in}\boldsymbol{A}^{\#}[\omega_1].\ \tau[\uparrow_{\boldsymbol{x}'} \omega_1, \boldsymbol{x}'/\boldsymbol{x}']$ by i.h. on e with $\mathcal{E}_1$ and Subst. Application (Def. A.2.4)

$\Omega' \vdash e[\omega_1]\backslash \boldsymbol{x_0} \in \tau[\uparrow_{\boldsymbol{x}'} \omega_1, \boldsymbol{x}'/\boldsymbol{x}'][\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}, \boldsymbol{x_0}/\boldsymbol{x}']$ by pop

$\Omega_1 \vdash \nabla \boldsymbol{x}'{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff}$ by Lemma B.8.1 on $\mathcal{E}_1$

$\Omega_1 \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$ by inversion using $\nabla\mathsf{wff}$

$\Omega_1, \boldsymbol{x}'\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash \tau\ \mathsf{wff}$ by inversion using $\nabla\mathsf{wff}$

$\Omega'\ \mathsf{ctx}$ by Lemma B.3.2

$(\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1])\ \mathsf{ctx}$ by inversion

$\Omega_1' \vdash \boldsymbol{A}^{\#}[\omega_1]\ \mathsf{wff}$ by inversion using ctxAddNew

$\Omega_1'\ \mathsf{ctx}$ by inversion using ctxAddNew

$\Omega_1', \boldsymbol{x}'\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x}'} \omega_1, \boldsymbol{x}'/\boldsymbol{x}') : \Omega_1, \boldsymbol{x}'\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ tpSubIndNew

$\Omega_1\ \mathsf{ctx}$ by Lemma B.3.1 on $\mathcal{E}_1$

$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash (\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x}') : \Omega_1, \boldsymbol{x}'\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ by Lemmas B.5.5 and B.5.2 and tpSubIndNew

$\Omega_1', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}, \boldsymbol{x_0}/\boldsymbol{x}') : \Omega_1', \boldsymbol{x}'\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]$ by Lemmas B.5.5 and B.5.2 and tpSubIndNew

$$\begin{aligned}&\tau[\uparrow_{\boldsymbol{x}'} \omega_1, \boldsymbol{x}'/\boldsymbol{x}'][\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}, \boldsymbol{x_0}/\boldsymbol{x}'] \\ &\quad = \tau[(\uparrow_{\boldsymbol{x}'} \omega_1, \boldsymbol{x}'/\boldsymbol{x}') \circ (\uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}, \boldsymbol{x_0}/\boldsymbol{x}')] && \text{by Lemmas B.12.1 and B.16.7} \\ &\quad = \tau[(\omega_1 \circ \uparrow_{\boldsymbol{x_0}} \mathrm{id}_{\Omega_1'}), \boldsymbol{x_0}/\boldsymbol{x}'] && \text{by Composition (Def. A.2.3) and Subst. Application (Def. A.2.4)} \\ &\quad = \tau[\uparrow_{\boldsymbol{x_0}} (\omega_1 \circ \mathrm{id}_{\Omega_1'}), \boldsymbol{x_0}/\boldsymbol{x}'] && \text{by Composition (Def. A.2.3)}\end{aligned}$$

(Case Continued →)

$= \tau[(\omega_1 \circ \mathrm{id}_{\Omega_1'}), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Lemma B.5.4
$= \tau[(\mathrm{id}_{\Omega_1} \circ \omega_1), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Lemma B.16.6
$= \tau[\uparrow_{\boldsymbol{x_0}} (\mathrm{id}_{\Omega_1} \circ \omega_1), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Lemma B.5.4
$= \tau[(\mathrm{id}_{\Omega_1} \circ \uparrow_{\boldsymbol{x_0}} \omega_1), \boldsymbol{x_0}/\boldsymbol{x'}]$ by Composition (Def. A.2.3)
$= \tau[(\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}) \circ (\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x})]$ by Composition (Def. A.2.3) and Subst. Application (Def. A.2.4)
$= \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}]$ by Lemma B.12.1
$\Omega' \vdash e[\omega_1]\backslash \boldsymbol{x_0} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}]$ by above
$\Omega' \vdash e[\omega_1]\backslash \boldsymbol{x}[\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}] \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}][\uparrow_{\boldsymbol{x_0}} \omega_1, \boldsymbol{x_0}/\boldsymbol{x}]$ by Subst. Application (Def. A.2.4)
$\Omega' \vdash e[\omega]\backslash \boldsymbol{x}[\omega] \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}][\omega]$ by Lemmas B.7.4 and B.7.5
$\Omega' \vdash (e\backslash \boldsymbol{x})[\omega] \in (\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}])[\omega]$ by Subst. Application (Def. A.2.4)

Case: $(e\backslash u)$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla \mathcal{W}_2.\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ (\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2)\end{array}}{\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2 \vdash e\backslash u \in \tau}\ \mathsf{popW}$

$\Omega' \vdash \omega : (\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2)$ by assumption
$\mathcal{W} \leq \mathcal{W}_2$ by assumption
$(\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u \overset{\triangledown}{\in} \mathcal{W}, \Omega_2)$ by assumption
There exists an Ω_1', Ω_2', u_0, and ω_1 such that
$\Omega_1', u_0 \overset{\triangledown}{\in} \mathcal{W} \vdash (\uparrow_{u_0} \omega_1, u_0/u) : \Omega_1, u \overset{\triangledown}{\in} \mathcal{W}$
and $\Omega' = \Omega_1', u_0 \overset{\triangledown}{\in} \mathcal{W}, \Omega_2'$
and $\omega_1 \leq \omega$
and $(\uparrow_{u_0} \omega_1, u_0/u) \leq \omega$
and $(\Omega_1', u_0 \overset{\triangledown}{\in} \mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1', u_0 \overset{\triangledown}{\in} \mathcal{W}, \Omega_2')$. by Lemma B.16.2
$\Omega'\ \mathsf{ctx}$ by Lemma B.3.2
$\Omega_1' \vdash \omega_1 : \Omega_1$ by inversion using tpSubIndNew
$\Omega_1' \vdash e[\omega_1] \in \nabla \mathcal{W}_2.\ \tau[\omega_1]$ by i.h. on e with $\mathcal{E}_1$ and Subst. Application (Def. A.2.4)
$\Omega' \vdash e[\omega_1]\backslash u_0 \in \tau[\omega_1]$ by popW
$\Omega' \vdash e[\omega_1]\backslash u[\uparrow_{u_0} \omega_1, u_0/u] \in \tau[\omega_1]$ by Subst. Application (Def. A.2.4)
$\Omega' \vdash e[\omega]\backslash u[\omega] \in \tau[\omega]$ by Lemmas B.7.4 and B.7.5
$\Omega' \vdash (e\backslash u)[\omega] \in \tau[\omega]$ by Subst. Application (Def. A.2.4)

Case: $(e,\ f)$ and

$$\mathcal{E} = \dfrac{\Omega \vdash (\exists \alpha{\in}\delta.\ \tau)\ \mathsf{wff} \quad \overset{\mathcal{E}_1}{\Omega \vdash e \in \delta} \quad \overset{\mathcal{E}_2}{\Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]}}{\Omega \vdash (e,\ f) \in \exists \alpha{\in}\delta.\ \tau}\ \mathsf{pairI}$$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega \vdash \exists \alpha{\in}\delta.\ \tau\ \mathsf{wff}$	by assumption
$\Omega, \alpha{\in}\delta \vdash \tau\ \mathsf{wff}$	by inversion using $\exists\mathsf{wff}$
$\Omega'\ \mathsf{ctx}$	by Lemma B.3.2
$\Omega' \vdash e[\omega] \in \delta[\omega]$	by i.h. on e with $\mathcal{E}_1$
$\Omega' \vdash f[\omega] \in \tau[\mathrm{id}_\Omega, e/\alpha][\omega]$	by i.h. on f with $\mathcal{E}_2$
$\Omega\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \delta\ \mathsf{wff}$	by Lemma B.8.1 on $\mathcal{E}_1$
$\Omega' \vdash \delta[\omega]\ \mathsf{wff}$	by Lemma B.6.1
$\Omega \vdash (\mathrm{id}_\Omega, e/\alpha) : \Omega, \alpha{\in}\delta$	by Lemmas B.5.5 and B.5.2 and $\mathcal{E}_1$ and $\mathsf{tpSubInd}$
$\Omega' \vdash (\mathrm{id}_{\Omega'}, e[\omega]/\alpha) : \Omega', \alpha{\in}\delta[\omega]$	by Lemmas B.5.5 and B.5.2 and $\mathsf{tpSubInd}$
$(\Omega', \alpha{\in}\delta[\omega])\ \mathsf{ctx}$	by ctxAdd
$\Omega', \alpha{\in}\delta[\omega] \vdash \alpha \in \delta[\uparrow_\alpha \omega]$	by Lemmas B.5.1 and B.5.4
$\Omega', \alpha{\in}\delta[\omega] \vdash (\uparrow_\alpha \omega, \alpha/\alpha) : \Omega, \alpha{\in}\delta$	by $\mathsf{tpSubShift}$ and $\mathsf{tpSubInd}$
$\tau[\mathrm{id}_\Omega, e/\alpha][\omega]$	
$\quad = \tau[(\mathrm{id}_\Omega, e/\alpha) \circ \omega]$	by Lemmas B.12.1 and B.16.7
$\quad = \tau[(\mathrm{id}_\Omega \circ \omega), e[\omega]/\alpha]$	by Composition (Def. A.2.3)
$\quad = \tau[(\omega \circ \mathrm{id}_{\Omega'}), e[\omega]/\alpha]$	by Lemma B.16.6
$\quad = \tau[(\uparrow_\alpha \omega, \alpha/\alpha) \circ (\mathrm{id}_{\Omega'}, e[\omega]/\alpha)]$	by Composition (Def. A.2.3) and Subst. Application (Def. A.2.4)
$\quad = \tau[\uparrow_\alpha \omega, \alpha/\alpha][\mathrm{id}_{\Omega'}, e[\omega]/\alpha]$	by Lemma B.12.1
$\Omega' \vdash f[\omega] \in \tau[\uparrow_\alpha \omega, \alpha/\alpha][\mathrm{id}_{\Omega'}, e[\omega]/\alpha]$	by above
$\Omega' \vdash \exists \alpha{\in}\delta[\omega].\ \tau[\uparrow_\alpha \omega, \alpha/\alpha]\ \mathsf{wff}$	by Lemma B.6.1 and Subst. Application (Def. A.2.4))
$\Omega \vdash (e[\omega],\ f[\omega]) \in \exists \alpha{\in}\delta[\omega].\ \tau[\uparrow_\alpha \omega, \alpha/\alpha]$	by pairI
$\Omega \vdash (e,\ f)[\omega] \in (\exists \alpha{\in}\delta.\ \tau)[\omega]$	by Subst. Application (Def. A.2.4)

Case: $(\mu u{\in}\tau.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega, u{\in}\tau \vdash e \in \tau\end{array}}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\ \mathsf{fix}$

$\Omega' \vdash \omega : \Omega$	by assumption
$(\Omega, u{\in}\tau)$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \tau$ wff	by inversion using ctxAdd
$\Omega' \vdash \tau[\omega]$ wff	by Lemma B.6.1
$\Omega', u{\in}\tau[\omega] \vdash \uparrow_u \omega : \Omega$	by tpSubShift
$(\Omega', u{\in}\tau[\omega])$ ctx	by ctxAdd
$\Omega', u{\in}\tau[\omega] \vdash u \in \tau[\omega]$	by Lemma B.5.1
$\Omega', u{\in}\tau[\omega] \vdash u \in \tau[\uparrow_u \omega]$	by Lemma B.5.4
$\Omega', u{\in}\tau[\omega] \vdash (\uparrow_u \omega, u/u) : \Omega, u{\in}\tau$	by tpSubInd
$\Omega', u{\in}\tau[\omega] \vdash e[\uparrow_u \omega, u/u] \in \tau[\uparrow_u \omega, u/u]$	by i.h. on e with $\mathcal{E}_1$
$\omega \leq (\uparrow_u \omega, u/u)$	by weakening rules
$\Omega', u{\in}\tau[\omega] \vdash e[\uparrow_u \omega, u/u] \in \tau[\omega]$	by Lemma B.7.4
$\Omega' \vdash \mu u{\in}\tau[\omega].\ e[\uparrow_u \omega, u/u] \in \tau[\omega]$	by fix
$\Omega' \vdash (\mu u{\in}\tau.\ e)[\omega] \in \tau[\omega]$	by Subst. Application (Def. A.2.4)

. . . . . .

Case: $(\epsilon\alpha{\in}\delta.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega \vdash \tau \text{ wff} \qquad \Omega, \alpha{\in}\delta \vdash c \in \tau\end{array}}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\ \mathsf{cEps}$

$\Omega' \vdash \omega : \Omega$	by assumption
$(\Omega, \alpha{\in}\delta)$ ctx	by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \delta$ wff	by inversion using wff
$\Omega' \vdash \delta[\omega]$ wff	by Lemma B.6.1
$(\Omega', \alpha{\in}\delta[\omega])$ ctx	by ctxAdd
$\Omega', \alpha{\in}\delta[\omega] \vdash \alpha \in \delta[\uparrow_\alpha \omega]$	by Lemmas B.5.1 and B.5.4
$\Omega', \alpha{\in}\delta[\omega] \vdash (\uparrow_\alpha \omega, \alpha/\alpha) : \Omega, \alpha{\in}\delta$	by tpSubShift and tpSubInd
$\Omega \vdash \tau$ wff	by assumption
$\Omega' \vdash \tau[\omega]$ wff	by Lemma B.6.1
$\Omega', \alpha{\in}\delta[\omega] \vdash c[\uparrow_\alpha \omega, \alpha/\alpha] \in \tau[\uparrow_\alpha \omega, \alpha/\alpha]$	by i.h. on c with $\mathcal{E}_1$
$\omega \leq (\uparrow_\alpha \omega, \alpha/\alpha)$	by weakening rules
$\Omega', \alpha{\in}\delta[\omega] \vdash c[\uparrow_\alpha \omega, \alpha/\alpha] \in \tau[\omega]$	by Lemma B.7.4
$\Omega' \vdash \epsilon\alpha{\in}\delta[\omega].\ c[\uparrow_\alpha \omega, \alpha/\alpha] \in \tau[\omega]$	by cEps
$\Omega' \vdash (\epsilon\alpha{\in}\delta.\ c)[\omega] \in \tau[\omega]$	by Subst. Application (Def. A.2.4)

Case: $(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash c \in \tau\end{array}}{\Omega \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{cNew}$

See Case for new (above)

Case: $(\overline{e} \mapsto f)$ and

$$\mathcal{E} = \frac{\begin{array}{ccc}\mathcal{E}_1 & \mathcal{E}_2 & \\ \Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}] & \Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta} & \Omega \vdash \forall \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}\end{array}}{\Omega \vdash \overline{e} \mapsto f \in \forall \overline{\alpha{\in}\delta}.\ \tau}\ \mathsf{cMatch}$$

$\Omega' \vdash \omega : \Omega$	by assumption
$\Omega \vdash \forall \overline{\alpha{\in}\delta}.\ \tau\ \mathsf{wff}$	by assumption
$\Omega\ \mathsf{ctx}$	by Lemma B.3.1 on $\mathcal{E}_1$
$(\Omega, \overline{\alpha{\in}\delta})\ \mathsf{ctx}$	by Lemma B.16.13
$\Omega \vdash f[\omega] \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}][\omega]$	by i.h. on f with $\mathcal{E}_1$
$\Omega' \vdash \mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha} : \Omega', \overline{\alpha{\in}\delta}[\omega]$	by i.h. on $\overline{e}$ with $\mathcal{E}_2$
$(\forall \overline{\alpha{\in}\delta}.\ \tau)[\omega] = \forall \overline{\alpha{\in}\delta}[\omega].\ \tau[\omega + \overline{\alpha{\in}\delta}]$	by Subst. Application (Def. A.2.4)
$\Omega', \overline{\alpha{\in}\delta}[\omega] \vdash (\omega + \overline{\alpha{\in}\delta}) : \Omega, \overline{\alpha{\in}\delta}$	by Lemma B.16.12
$\tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}][\omega]$	
$= \tau[(\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}) \circ \omega]$	by Lemmas B.12.1 and B.16.7
$= \tau[(\omega + \overline{\alpha{\in}\delta}) \circ (\mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha})]$	by Lemma B.16.12
$= \tau[\omega + \overline{\alpha{\in}\delta}][\mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha}]$	by Lemma B.12.1
$\Omega \vdash f[\omega] \in \tau[\omega + \overline{\alpha{\in}\delta}][\mathrm{id}_{\Omega'}, (\overline{e}[\omega])/\overline{\alpha}]$	by above
$\Omega' \vdash \overline{e}[\omega] \mapsto f[\omega] \in \forall \overline{\alpha{\in}\delta}[\omega].\ \tau[\omega + \overline{\alpha{\in}\delta}]$	by cMatch
$\Omega' \vdash (\overline{e} \mapsto f)[\omega] \in (\forall \overline{\alpha{\in}\delta}.\ \tau)[\omega]$	by Subst. Application (Def. A.2.4)

Case: $(c\backslash \boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash c \in \nabla \boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash c\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$

See Case for pop (above)

. . . . . .

Case: *nil* and $\mathcal{E} = \Omega \vdash \mathrm{id}_\Omega : \Omega$

$\Omega' \vdash \omega : \Omega$	by assumption
Ω ctx	by assumption
$\Omega' \vdash \mathrm{id}_{\Omega'} : \Omega'$	by Lemma B.5.5

Case: $(\overline{f'}; f)$ and $\mathcal{E} = \dfrac{\Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'} : \Omega, \overline{\alpha' \in \delta'} \quad \mathcal{E}_1 \qquad \Omega \vdash f \in \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}] \quad \mathcal{E}_2}{\Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega, \overline{\alpha' \in \delta'}, \alpha \in \delta}\ \mathsf{tpSubInd}$

$\Omega' \vdash \omega : \Omega$	by assumption
$(\Omega, \overline{\alpha' \in \delta'}, \alpha \in \delta)$ ctx	by assumption
$(\Omega, \overline{\alpha' \in \delta'})$ ctx	by inversion using ctxAdd
$(\overline{\alpha' \in \delta'}; \alpha \in \delta)[\omega] = (\overline{\alpha' \in \delta'})[\omega]; \alpha \in (\delta[\omega + \overline{\alpha' \in \delta'}])$	by Lemma B.16.8
$\Omega' \vdash \mathrm{id}_{\Omega'}, (\overline{f'}[\omega])/\overline{\alpha'} : \Omega', \overline{\alpha' \in \delta'}[\omega]$	by i.h. on $\overline{f'}$ with $\mathcal{E}_1$
$\Omega' \vdash f[\omega] \in \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}][\omega]$	by i.h. on f with $\mathcal{E}_2$
$\Omega', \overline{\alpha' \in \delta'}[\omega] \vdash (\omega + \overline{\alpha' \in \delta'}) : \Omega, \overline{\alpha' \in \delta'}$	by Lemma B.16.12
$\delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}][\omega]$	
$\quad = \delta[(\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}) \circ \omega]$	by Lemmas B.12.1 and B.16.7
$\quad = \delta[(\omega + \overline{\alpha' \in \delta'}) \circ (\mathrm{id}_{\Omega'}, (\overline{f'}[\omega])/\overline{\alpha})]$	by Lemma B.16.10
$\quad = \delta[\omega + \overline{\alpha' \in \delta'}][\mathrm{id}_{\Omega'}, (\overline{f'}[\omega])/\overline{\alpha'}]$	by Lemma B.12.1
$\Omega' \vdash f[\omega] \in \delta[\omega + \overline{\alpha' \in \delta'}][\mathrm{id}_{\Omega'}, (\overline{f'}[\omega])/\overline{\alpha'}]$	by above
$\Omega' \vdash \mathrm{id}_{\Omega'}, (\overline{f'}[\omega])/\overline{\alpha'}, f[\omega]/\alpha : \Omega', (\overline{\alpha' \in \delta'; \alpha \in \delta})[\omega]$	by tpSubInd and above

□

B.18 Meta-Theory: Important Corollaries

Lemma B.18.1 (Substitution Composition is Well-Typed)**.**
If Ω ctx *and* $\Omega' \vdash \omega : \Omega$ *and* $\Omega_2 \vdash \omega_2 : \Omega'$*, then* $\Omega_2 \vdash (\omega \circ \omega_2) : \Omega$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \Omega' \vdash \omega : \Omega$ and $\mathcal{F} :: \Omega_2 : \omega_2 : \Omega'$.

Case: $\mathcal{E} = \Omega' \vdash \omega : \Omega$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash \delta \text{ wff}\end{array}}{\Omega_2, \alpha{\in}\delta \vdash \uparrow_\alpha \omega_2 : \Omega'}\,\mathsf{tpSubShift}$

Ω ctx	by assumption
$\Omega_2 \vdash \omega \circ \omega_2 : \Omega$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$\Omega_2 \vdash \delta$ wff	by assumption
$\Omega_2, \alpha{\in}\delta \vdash \uparrow_\alpha (\omega \circ \omega_2) : \Omega$	by assumptions and tpSubShift
$\Omega_2, \alpha{\in}\delta \vdash \omega \circ (\uparrow_\alpha \omega_2) : \Omega$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\,\mathsf{tpSubBase}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \cdot$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

Ω ctx	by assumption
$\Omega_2 \vdash \omega_2 : \cdot$	by assumption ($\mathcal{F}$)
$\Omega_2 \vdash (\cdot \circ \omega_2) : \cdot$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

$(\Omega, \alpha{\in}\delta)\ \mathsf{ctx}$	by assumption
$\Omega\ \mathsf{ctx}$	by inversion using ctxAdd
$\Omega \vdash \delta\ \mathsf{wff}$	by inversion using ctxAdd
$\Omega_2 \vdash (\omega \circ \omega_2) : \Omega$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$
$\Omega' \vdash e \in \delta[\omega]$	by assumption
$\Omega_2 \vdash e[\omega_2] \in \delta[\omega][\omega_2]$	by Lemma B.17.7
$\Omega_2 \vdash e[\omega_2] \in \delta[\omega \circ \omega_2]$	by Lemma B.12.1
$\Omega_2 \vdash ((\omega \circ \omega_2), e[\omega_2]/\alpha) : \Omega, \alpha{\in}\delta$	by tpSubInd
$\Omega_2 \vdash ((\omega, e/\alpha) \circ \omega_2) : \Omega, \alpha{\in}\delta$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{(\Omega', \boldsymbol{x'}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega]) \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) : (\Omega, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1\\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash \boldsymbol{A}^{\#}[\omega][\omega_2]\ \mathsf{wff}\end{array}}{(\Omega_2, \boldsymbol{x_2}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega][\omega_2]) \vdash (\uparrow_{\boldsymbol{x_2}} \omega_2, \boldsymbol{x_2}/\boldsymbol{x'}) : (\Omega', \boldsymbol{x'}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega])}\ \mathsf{tpSubIndNew}$

$(\Omega, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#})\ \mathsf{ctx}$	by assumption
$\Omega\ \mathsf{ctx}$	by inversion using ctxAddNew
$\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$	by inversion using ctxAddNew
$(\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x_2}} \omega_2, \boldsymbol{x_2}/\boldsymbol{x'})$	
$= (\omega \circ \uparrow_{\boldsymbol{x_2}} \omega_2), \boldsymbol{x'}[\uparrow_{\boldsymbol{x_2}} \omega_2, \boldsymbol{x_2}/\boldsymbol{x'}]/\boldsymbol{x}$	by Composition (Def. A.2.3)
$= (\omega \circ \uparrow_{\boldsymbol{x_2}} \omega_2), \boldsymbol{x_2}/\boldsymbol{x}$	by Subst. Application (Def. A.2.4)
$= \uparrow_{\boldsymbol{x_2}} (\omega \circ \omega_2), \boldsymbol{x_2}/\boldsymbol{x}$	by Composition (Def. A.2.3)
$\Omega_2 \vdash (\omega \circ \omega_2) : \Omega$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\Omega_2 \vdash \boldsymbol{A}^{\#}[\omega][\omega_2]\ \mathsf{wff}$	by assumption
$\boldsymbol{A}^{\#}[\omega][\omega_2] = \boldsymbol{A}^{\#}[\omega \circ \omega_2]$	by Lemma B.12.1
$\Omega_2, \boldsymbol{x_2}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega][\omega_2] \vdash \uparrow_{\boldsymbol{x_2}} (\omega \circ \omega_2), \boldsymbol{x_2}/\boldsymbol{x} : \Omega, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}$	by tpSubIndNew
$\Omega_2, \boldsymbol{x_2}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}[\omega][\omega_2] \vdash (\uparrow_{\boldsymbol{x'}} \omega, \boldsymbol{x'}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x_2}} \omega_2, \boldsymbol{x_2}/\boldsymbol{x'}) : \Omega, \boldsymbol{x}{\overset{\triangledown}{\in}}\boldsymbol{A}^{\#}$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega', u'\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \mathcal{W}\ \mathsf{world}\end{array}}{(\Omega_2, u_2\overset{\triangledown}{\in}\mathcal{W}) \vdash (\uparrow_{u_2} \omega_2, u_2/u') : (\Omega', u'\overset{\triangledown}{\in}\mathcal{W})}\ \mathsf{tpSubWorld}$

$(\Omega, u\overset{\triangledown}{\in}\mathcal{W})\ \mathsf{ctx}$ by assumption
$\Omega\ \mathsf{ctx}$ by inversion using ctxAddWorld
$\mathcal{W}\ \mathsf{world}$ by inversion using ctxAddWorld
$(\uparrow_{u'} \omega, u'/u) \circ (\uparrow_{u_2} \omega_2, u_2/u')$
$= (\omega \circ \uparrow_{u_2} \omega_2), u'[\uparrow_{u_2} \omega_2, u_2/u']/u$ by Composition (Def. A.2.3)
$= (\omega \circ \uparrow_{u_2} \omega_2), u_2/u$ by Subst. Application (Def. A.2.4)
$= \uparrow_{u_2} (\omega \circ \omega_2), u_2/u$ by Composition (Def. A.2.3)
$\Omega_2 \vdash (\omega \circ \omega_2) : \Omega$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\Omega_2, u_2\overset{\triangledown}{\in}\mathcal{W} \vdash \uparrow_{u_2} (\omega \circ \omega_2), u_2/u : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$ by tpSubWorld
$\Omega_2, u_2\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u'} \omega, u'/u) \circ (\uparrow_{u_2} \omega_2, u_2/u') : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta\ \mathsf{wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash e \in \delta[\omega_2]\end{array}}{\Omega_2 \vdash (\omega_2, e/\alpha) : (\Omega', \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

$\Omega\ \mathsf{ctx}$ by assumption
$\Omega_2 \vdash \omega \circ \omega_2 : \Omega$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$\Omega_2 \vdash (\uparrow_\alpha \omega) \circ (\omega_2, e/\alpha) : \Omega$ by Composition (Def. A.2.3)

□

Lemma B.18.2 (Variable under Composed Substitutions)**.**
If $\Omega' \vdash \omega : \Omega$ *and* $\Omega_2 \vdash \omega_2 : \Omega'$ *and* $\alpha_0[\omega] = e_0$,
then $\alpha_0[\omega \circ \omega_2] = e_0[\omega_2]$.

Proof. By induction lexicographically on
$\mathcal{E} :: \Omega' \vdash \omega : \Omega$ and $\mathcal{F} :: \Omega_2 \vdash \omega_2 : \Omega'$.

Case: $\mathcal{E} = \Omega' \vdash \omega : \Omega$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash \delta \ \mathsf{wff}\end{array}}{\Omega_2, \alpha{\in}\delta \vdash \uparrow_\alpha \omega_2 : \Omega'}\ \mathsf{tpSubShift}$

$\alpha_0[\omega] = e_0$ by assumption
$\alpha_0[\omega \circ \uparrow_\alpha \omega_2]$
$= \alpha_0[\uparrow_\alpha (\omega \circ \omega_2)]$ by Composition (Def. A.2.3)
$= \alpha_0[\omega \circ \omega_2]$ by Lemma B.5.4
$= e_0[\omega_2]$ by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$= e_0[\uparrow_\alpha \omega_2]$ by Lemma B.5.4

Case: $\mathcal{E} = \dfrac{}{\cdot \vdash \cdot : \cdot}\ \mathsf{tpSubBase}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \cdot$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

$\alpha_0[\cdot] = e_0$ by assumption
$\alpha_0[\cdot]$ is undefined by Subst. Application (Def. A.2.4)
Contradiction, so this case is vacuously true.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e_0 \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e_0/\alpha_0) : (\Omega, \alpha_0{\in}\delta)}\ \mathsf{tpSubInd}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

$\alpha_0[\omega, e_0/\alpha_0] = e_0$ by Subst. Application (Def. A.2.4)
(and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)
$\alpha_0[(\omega, e_0/\alpha_0) \circ \omega_2]$
$= \alpha_0[(\omega \circ \omega_2), e_0[\omega_2]/\alpha_0]$ by Composition (Def. A.2.3)
$= e_0[\omega_2]$ by Subst. Application (Def. A.2.4)
(and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash e \in \delta[\omega]\end{array}}{\Omega' \vdash (\omega, e/\alpha) : (\Omega, \alpha{\in}\delta)}\ \mathsf{tpSubInd}$, and $\alpha \neq \alpha_0$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq\!\uparrow_{\alpha''} \omega_2''$

$\alpha_0[\omega, e/\alpha] = \alpha_0[\omega] = e_0$ — by assumption and Subst. Application (Def. A.2.4) (and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)

$\alpha_0[(\omega, e/\alpha) \circ \omega_2]$
$= \alpha_0[(\omega \circ \omega_2), e[\omega_2]/\alpha]$ — by Composition (Def. A.2.3)
$= \alpha_0[\omega \circ \omega_2]$ — by Subst. Application (Def. A.2.4) (and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)
$= e_0[\omega_2]$ — by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{\Omega', \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega] \vdash (\uparrow_{\boldsymbol{x_0}} \omega, \boldsymbol{x_0}/\alpha_0) : (\Omega, \alpha_0\overset{\triangledown}{\in}\boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq\!\uparrow_{\alpha''} \omega_2''$

$(\Omega, \alpha_0\overset{\triangledown}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$ — by assumption
$\alpha_0[\uparrow_{\boldsymbol{x_0}} \omega, \boldsymbol{x_0}/\alpha_0] = \boldsymbol{x_0}$ — by Subst. Application (Def. A.2.4) and Casting (Def. A.2.2) since $\alpha_0 = \boldsymbol{x}$

$\alpha_0[(\uparrow_{\boldsymbol{x_0}} \omega, \boldsymbol{x_0}/\alpha_0) \circ \omega_2]$
$= \alpha_0[((\uparrow_{\boldsymbol{x_0}} \omega) \circ \omega_2), \boldsymbol{x_0}[\omega_2]/\alpha_0]$ — by Composition (Def. A.2.3)
$= \boldsymbol{x_0}[\omega_2]$ — by Subst. Application (Def. A.2.4) and Casting (Def. A.2.2) since $\alpha_0 = \boldsymbol{x}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \boldsymbol{A}^{\#}[\omega]\ \mathsf{wff}\end{array}}{\Omega' \vdash (\uparrow_{\boldsymbol{x}'} \omega, \boldsymbol{x}'/\boldsymbol{x}) : (\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#})}\ \mathsf{tpSubIndNew}$, and $\boldsymbol{x} \neq \alpha_0$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

$\alpha_0[\uparrow_{\boldsymbol{x}'} \omega, \boldsymbol{x}'/\boldsymbol{x}] = \alpha_0[\omega] = e_0$ by assumption and Subst. Application (Def. A.2.4) and Lemma B.5.4 (and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)

$\alpha_0[(\uparrow_{\boldsymbol{x}'} \omega, \boldsymbol{x}'/\boldsymbol{x}) \circ \omega_2]$
- $= \alpha_0[((\uparrow_{\boldsymbol{x}'} \omega) \circ \omega_2), \boldsymbol{x}'[\omega_2]/\boldsymbol{x}]$ by Composition (Def. A.2.3)
- $= \alpha_0[\omega \circ \omega_2]$ by Subst. Application (Def. A.2.4) and Lemma B.5.4 (and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)
- $= e_0[\omega_2]$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{\Omega', u_0 \overset{\triangledown}{\in} \mathcal{W} \vdash (\uparrow_{u_0} \omega, u_0/\alpha_0) : (\Omega, \alpha_0 \overset{\triangledown}{\in} \mathcal{W})}\ \mathsf{tpSubWorld}$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

$(\Omega, \alpha_0 \overset{\triangledown}{\in} \mathcal{W})\ \mathsf{ctx}$ by assumption

$\alpha_0[\uparrow_{u_0} \omega, u_0/\alpha_0] = u_0$ by Subst. Application (Def. A.2.4)
Note that α_0 must be a u by tpSubWorld.

$\alpha_0[(\uparrow_{u_0} \omega, u_0/\alpha_0) \circ \omega_2]$
- $= \alpha_0[((\uparrow_{u_0} \omega) \circ \omega_2), u_0[\omega_2]/\alpha_0]$ by Composition (Def. A.2.3)
- $= u_0[\omega_2]$ by Subst. Application (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega' \vdash \omega : \Omega \qquad \mathcal{W}\ \mathsf{world}\end{array}}{\Omega' \vdash (\uparrow_{u'} \omega, u'/u) : (\Omega, u{\in}^{\triangledown}\mathcal{W})}\ \mathsf{tpSubWorld}$, and $u \neq \alpha_0$

and $\mathcal{F} = \Omega_2 \vdash \omega_2 : \Omega'$, where $\omega_2 \neq \uparrow_{\alpha''} \omega_2''$

$\alpha_0[\uparrow_{u'} \omega, u'/u] = \alpha_0[\omega] = e_0$ by assumption and Subst. Application (Def. A.2.4) and Lemma B.5.4 (and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)

$\alpha_0[(\uparrow_{u'} \omega, u'/u) \circ \omega_2]$
$= \alpha_0[((\uparrow_{u'} \omega) \circ \omega_2), u'[\omega_2]/u]$ by Composition (Def. A.2.3)
$= \alpha_0[\omega \circ \omega_2]$ by Subst. Application (Def. A.2.4) and Lemma B.5.4 (and Casting (Def. A.2.2) when $\alpha_0 = \boldsymbol{x}$)
$= e_0[\omega_2]$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega' \vdash \omega : \Omega \qquad \Omega' \vdash \delta\ \mathsf{wff}\end{array}}{\Omega', \alpha{\in}\delta \vdash \uparrow_\alpha \omega : \Omega}\ \mathsf{tpSubShift}$

and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1\\ \Omega_2 \vdash \omega_2 : \Omega' \qquad \Omega_2 \vdash e \in \delta[\omega_2]\end{array}}{\Omega_2 \vdash (\omega_2, e/\alpha) : (\Omega', \alpha{\in}\delta)}\ \mathsf{tpSubInd}$

$\alpha_0[\uparrow_\alpha \omega] = \alpha_0[\omega] = e_0$ by assumption and Lemma B.5.4
$\omega_2 \leq (\omega_2, e/\alpha)$ by $\mathsf{leSubAdd}$
$\alpha_0[(\uparrow_\alpha \omega) \circ (\omega_2, e/\alpha)]$
$= \alpha_0[\omega \circ \omega_2]$ by Composition (Def. A.2.3)
$= e_0[\omega_2]$ by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$= e_0[\omega_2, e/\alpha]$ by Lemma B.7.5

□

Lemma B.18.3 (Weakening Substitution on Left of Composition)**.**
If $\omega_1 \leq \omega_2$ *and* $\Omega_1 \vdash \omega_1 : \Omega_0$ *and* $\Omega_1 \vdash \omega_2 : \Omega_0'$ *and* Ω_0 ctx *and* $\Omega_2 \vdash \omega : \Omega_1$
and there is no use of $\mathsf{leSubMiddle}$ *and* $\mathsf{leSubMiddleShift}$*,*
then $(\omega_1 \circ \omega) \leq (\omega_2 \circ \omega)$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \omega_1 \leq \omega_2$ noting that $\mathsf{leSubMiddle}$ and $\mathsf{leSubMiddleShift}$ are not used; this can also be viewed as stating that ω_1 is a subterm of ω_2.

Case: $\mathcal{E} = \dfrac{}{\omega_1 \leq \omega_1}\,\mathsf{leSubEq}$

Ω_0 ctx	by assumption
$\Omega_1 \vdash \omega_1 : \Omega_0$	by assumption
$\Omega_2 \vdash \omega : \Omega_1$	by assumption
$\Omega_2 \vdash (\omega_1 \circ \omega) : \Omega_0$	by Lemma B.18.1
$(\omega_1 \circ \omega) \leq (\omega_1 \circ \omega)$	by $\mathsf{leSubEq}$

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega_1 \leq \omega_2\end{array}}{\omega_1 \leq \uparrow_\alpha \omega_2}\,\mathsf{leSubShift}$

Ω_0 ctx	by assumption
$\Omega_1, \alpha{\in}\delta \vdash \uparrow_\alpha \omega_2 : \Omega_0'$	by assumption and inversion using $\mathsf{tpSubShift}$
$\Omega_1, \alpha{\in}\delta \vdash \omega_1 : \Omega_0$	by assumption
$\Omega_2 \vdash \omega : \Omega_1, \alpha{\in}\delta$	by assumption

In order for ω_1 to be well-typed, it must have a $\uparrow_\alpha$ in it.
Since $\omega_1 \leq \omega_2$, then so must ω_2.
This contradicts the well-formedness of $\uparrow_\alpha \omega_2$.
Therefore, this case is vacuously true.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega_1 \leq \omega_2\end{array}}{\omega_1 \leq \omega_2, e/\alpha}\,\mathsf{leSubAdd}$

Ω_0 ctx	by assumption
$\Omega_1 \vdash \omega_1 : \Omega_0$	by assumption
$\Omega_2 \vdash \omega : \Omega_1$	by assumption
$\Omega_1 \vdash \omega_2, e/\alpha : \Omega_0', \alpha{\in}\delta$	by assumption and inversion using tpSubInd
$\Omega_1 \vdash \omega_2 : \Omega_0'$	by inversion using tpSubInd
$(\omega_1 \circ \omega) \leq (\omega_2 \circ \omega)$	by i.h. on $\mathcal{E}_1$
$((\omega_2, e/\alpha) \circ \omega)$ exists	by Lemma B.18.1
$((\omega_2, e/\alpha) \circ \omega) = (\omega_2 \circ \omega, e[\omega]/\alpha)$	by Composition (Def. A.2.3)
$(\omega_2 \circ \omega) \leq (\omega_2 \circ \omega), e[\omega]/\alpha$	by leSubAdd
$(\omega_1 \circ \omega) \leq ((\omega_2, e/\alpha) \circ \omega)$	by above and Lemma B.13.4

□

Lemma B.18.4 (Weakening Substitution on Right of Composition)**.**
If $\omega_1 \leq \omega_2$ *and* $\Omega_2 \vdash \omega_1 : \Omega_1$ *and* $\Omega_2' \vdash \omega_2 : \Omega_1$ *and* Ω_0 ctx *and* $\Omega_1 \vdash \omega : \Omega_0$
and there is no use of leSubMiddle *and* leSubMiddleShift*,*
then $(\omega \circ \omega_1) \leq (\omega \circ \omega_2)$*.*

Proof. By induction lexicographically on $\mathcal{E} :: \omega_1 \leq \omega_2$ noting that leSubMiddle and leSubMiddleShift are not used; this can also be viewed as stating that ω_1 is a subterm of ω_2.

Case: $\mathcal{E} = \dfrac{}{\omega_1 \leq \omega_1}\,\mathsf{leSubEq}$

Ω_0 ctx	by assumption
$\Omega_1 \vdash \omega : \Omega_0$	by assumption
$\Omega_2 \vdash \omega_1 : \Omega_1$	by assumption
$\Omega_2 \vdash (\omega \circ \omega_1) : \Omega_0$	by Lemma B.18.1
$(\omega \circ \omega_1) \leq (\omega \circ \omega_1)$	by leSubEq

Case: $\mathcal{E} = \dfrac{\mathcal{E}_1}{\omega_1 \leq \omega_2}{\omega_1 \leq \uparrow_\alpha \omega_2}\,\mathsf{leSubShift}$

Ω_0 ctx	by assumption
$\Omega_1 \vdash \omega : \Omega_0$	by assumption
$\Omega_2 \vdash \omega_1 : \Omega_1$	by assumption
$\Omega_2', \alpha{\in}\delta \vdash \uparrow_\alpha \omega_2 : \Omega_1$	by assumption and inversion using tpSubShift
$\Omega_2' \vdash \omega_2 : \Omega_1$	by inversion using tpSubShift
$(\omega \circ \omega_1) \leq (\omega \circ \omega_2)$	by i.h. on $\mathcal{E}_1$
$(\omega \circ (\uparrow_\alpha \omega_2)) = \uparrow_\alpha (\omega \circ \omega_2)$	by Composition (Def. A.2.3)
$(\omega \circ \omega_2) \leq \uparrow_\alpha (\omega \circ \omega_2)$	by leSubShift
$(\omega \circ \omega_1) \leq (\omega \circ (\uparrow_\alpha \omega_2))$	by above and Lemma B.13.4

Case: $\mathcal{E} = \dfrac{\dfrac{\mathcal{E}_1}{\omega_1 \leq \omega_2}}{\omega_1 \leq \omega_2, e/\alpha}\,\mathsf{leSubAdd}$

Ω_0 ctx by assumption
$\Omega_2' \vdash (\omega_2, e/\alpha) : \Omega_1, \alpha{\in}\delta$
by assumption and inversion using $\mathsf{tpSubInd}$
$\Omega_2 \vdash \omega_1 : \Omega_1, \alpha{\in}\delta$ by assumption
$\Omega_1, \alpha{\in}\delta \vdash \omega : \Omega_0$ by assumption
In order for ω_1 to be well-typed, it must have some f/α in it.
Since $\omega_1 \leq \omega_2$, then so must ω_2.
This contradicts the well-formedness of $(\omega_2, e/\alpha)$.
Therefore, this case is vacuously true.

□

Lemma B.18.5 (Weakening Substitutions in Composition)**.**
If $\omega_A \leq \omega_1$ *and* $\omega_B \leq \omega_2$ *and* Ω_A ctx *and* $\Omega'_A \vdash \omega_A : \Omega_A$ *and* $\Omega''_A \vdash \omega_B : \Omega'_A$
and $\Omega'_A, \Omega'_C \vdash \omega_1 : \Omega_A, \Omega_C$ *and* $\Omega''_A, \Omega''_C \vdash \omega_2 : \Omega'_A, \Omega'_C$
and for both $\leq$ *there were no uses of* $\mathsf{leSubMiddleShift}$ *or* $\mathsf{leSubMiddle}$*,*
then $(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_2)$*.*

Proof. By induction on $\mathcal{E} :: \omega_A \leq \omega_1$ and $\mathcal{F} :: \omega_B \leq \omega_2$ noting that these derivations do not use $\mathsf{leSubMiddleShift}$ or $\mathsf{leSubMiddle}$; this can also be viewed as stating that ω_A is a subterm of ω_1 and ω_B is a subterm of ω_2.

Case: $\mathcal{E} = \dfrac{}{\omega_A \leq \omega_A}\,\mathsf{leSubEq}$ and $\mathcal{F} = \dfrac{}{\omega_B \leq \omega_B}\,\mathsf{leSubEq}$

Ω_A ctx	by assumption
$\Omega'_A \vdash \omega_A : \Omega_A$	by assumption
$\Omega''_A \vdash \omega_B : \Omega'_A$	by assumption
$\Omega''_A \vdash (\omega_A \circ \omega_B) : \Omega_A$	by Lemma B.18.1
$(\omega_A \circ \omega_B) \leq (\omega_A \circ \omega_B)$	by $\mathsf{leSubEq}$

Case: $\mathcal{E} = \omega_A \leq \omega_1$ and $\mathcal{F} = \dfrac{}{\omega_B \leq \omega_B}\,\mathsf{leSubEq}$

Ω_A ctx	by assumption
$\Omega'_A \vdash \omega_A : \Omega_A$	by assumption
$\Omega''_A \vdash \omega_B : \Omega'_A$	by assumption
$\Omega'_A \vdash \omega_1 : \Omega_A, \Omega_C$	by assumption
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_B)$	by Lemma B.18.3

Case: $\mathcal{E} = \dfrac{}{\omega_A \leq \omega_A}\,\mathsf{leSubEq}$ and $\mathcal{F} = \omega_B \leq \omega_2$

Ω_A ctx	by assumption
$\Omega'_A \vdash \omega_A : \Omega_A$	by assumption
$\Omega''_A \vdash \omega_B : \Omega'_A$	by assumption
$\Omega''_A, \Omega''_C \vdash \omega_2 : \Omega'_A$	by assumption
$(\omega_A \circ \omega_B) \leq (\omega_A \circ \omega_2)$	by Lemma B.18.4

Case: $\mathcal{E} = \omega_A \leq \omega_1$ and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \omega_B \leq \omega_2\end{array}}{\omega_B \leq (\uparrow_\alpha \omega_2)}\,\mathsf{leSubShift}$

Ω_A ctx	by assumption
$\Omega'_A \vdash \omega_A : \Omega_A$	by assumption
$\Omega''_A \vdash \omega_B : \Omega'_A$	by assumption
$\Omega'_A, \Omega'_C \vdash \omega_1 : \Omega_A, \Omega_C$	by assumption
$\Omega''_A, \Omega''_C, \alpha{\in}\delta \vdash \uparrow_\alpha \omega_2 : \Omega'_A, \Omega'_C$	by assumption and inversion using tpSubShift
$\Omega''_A, \Omega''_C \vdash \omega_2 : \Omega'_A, \Omega'_C$	by inversion using tpSubShift
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_2)$	by i.h. on $\mathcal{E}$ and $\mathcal{F}_1$
$(\omega_A \circ \omega_B) \leq \uparrow_\alpha (\omega_1 \circ \omega_2)$	by leSubShift
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \uparrow_\alpha \omega_2)$	by Composition (Def. A.2.4)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega_A \leq \omega_1\end{array}}{\omega_A \leq \omega_1, e/\alpha}\,\mathsf{leSubAdd}$ and $\mathcal{F} = \omega_B \leq \omega_2$

Ω_A ctx	by assumption
$\Omega'_A \vdash \omega_A : \Omega_A$	by assumption
$\Omega''_A \vdash \omega_B : \Omega'_A$	by assumption
$\Omega''_A, \Omega''_C \vdash \omega_2 : \Omega'_A, \Omega'_C$	by assumption
$\Omega'_A, \Omega'_C \vdash \omega_1, e/\alpha : \Omega_A, \Omega_C, \alpha{\in}\delta$	by assumption and inversion using tpSubInd
$\Omega'_A, \Omega'_C \vdash \omega_1 : \Omega_A, \Omega_C$	by inversion using tpSubInd
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_2)$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}$
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_2), e[\omega_2]/\alpha$	by leSubAdd
$(\omega_A \circ \omega_B) \leq ((\omega_1, e/\alpha) \circ \omega_2)$	by Composition (Def. A.2.3)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \omega_A \leq \omega_1\end{array}}{\omega_A \leq (\uparrow_{\alpha'} \omega_1)}\,\mathsf{leSubShift}$ and $\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}_1 \\ \omega_B \leq \omega_2\end{array}}{\omega_B \leq \omega_2, e/\alpha}\,\mathsf{leSubAdd}$

Ω_A ctx	by assumption
$\Omega'_A \vdash \omega_A : \Omega_A$	by assumption
$\Omega''_A \vdash \omega_B : \Omega'_A$	by assumption
$\Omega'_A, \Omega'_C, \alpha' \in \delta' \vdash \uparrow_{\alpha'} \omega_1 : \Omega_A, \Omega_C$	by assumption and inversion using tpSubShift
$\Omega''_A, \Omega''_C \vdash (\omega_2, e/\alpha) : \Omega'_A, \Omega'_C, \alpha \in \delta$	by assumption and inversion using tpSubInd
$\alpha' = \alpha$ and $\delta' = \delta$	by above and assumptions
$\Omega'_A, \Omega'_C \vdash \omega_1 : \Omega_A, \Omega_C$	by inversion using tpSubShift
$\Omega''_A, \Omega''_C \vdash \omega_2 : \Omega'_A, \Omega'_C$	by inversion using tpSubInd
$(\uparrow_{\alpha'} \omega_1) \circ (\omega_2, e/\alpha) = (\omega_1 \circ \omega_2)$	by above and Composition (Def. A.2.3)
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_2)$	by i.h. on $\mathcal{E}_1$ and $\mathcal{F}_1$
$(\omega_A \circ \omega_B) \leq (\uparrow_{\alpha'} \omega_1) \circ (\omega_2, e/\alpha)$	by above

□

Lemma B.18.6 (Equality of Substitution Composition Applied to e and c)**.** *If* $\Omega_1 \vdash \omega_1 : \Omega$ *and* $\Omega_2 \vdash \omega_2 : \Omega_1$*, then*

- *if* $\Omega \vdash e \in \delta$*, then* $e[\omega_1][\omega_2] = e[\omega_1 \circ \omega_2]$.
- *if* $\Omega \vdash c \in \tau$*, then* $c[\omega_1][\omega_2] = c[\omega_1 \circ \omega_2]$.
- *if* $\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}$*, then* $\overline{f}[\omega_1][\omega_2] = \overline{f}(\omega_1 \circ \omega_2)$.

Proof. By induction on e and c and $\overline{f}$.

Case: () and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx}}{\Omega \vdash () \in \mathrm{unit}}\ \mathsf{top}$

$\Omega_1 \vdash \omega_1 : \Omega$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$\Omega_2 \vdash (\omega_1 \circ \omega_2) : \Omega$	by Lemma B.18.1
$\mathrm{unit} = \mathrm{unit}[\omega_1][\omega_2] = \mathrm{unit}[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: u and $\mathcal{E} = \dfrac{(\Omega_A, u{\in}\tau, \Omega_B)\ \mathsf{ctx} \quad (\Omega_A, u{\in}\tau) \leq (\Omega_A, u{\in}\tau, \Omega_B)}{\Omega_A, u{\in}\tau, \Omega_B \vdash u \in \tau}\ \tau\mathsf{var}$

$\Omega_1 \vdash \omega_1 : \Omega_A, u{\in}\tau, \Omega_B$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$\Omega_2 \vdash u[\omega_1][\omega_2] \in \tau[\omega_1][\omega_2]$	by Lemma B.17.7 (twice)
$u[\omega_1][\omega_2] = u[\omega_1 \circ \omega_2]$	by Lemma B.18.2

Case: $\boldsymbol{x}$ and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad ((\boldsymbol{x}{\in}\boldsymbol{A}^{\#}) \text{ or } (\boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})) \text{ in } \Omega}{\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}}\ \mathsf{var}^{\#}$

$\Omega_1 \vdash \omega_1 : \Omega$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$\|\Omega\|\ \mathbf{ctx_{lf}}$	by Lemma B.4.1
$\|\Omega\| \vdash^{\mathsf{lf}} \boldsymbol{x} : \boldsymbol{A}$	by **LF_Var** and Casting (Def. A.2.1)
$\Omega_2 \vdash (\omega_1 \circ \omega_2) : \Omega$	by Lemma B.18.1
$\|\Omega_1\| \vdash^{\mathsf{lf}} \|\omega_1\| : \|\Omega\|$	by Lemma B.4.7
$\|\Omega_2\| \vdash^{\mathsf{lf}} \|\omega_2\| : \|\Omega_1\|$	by Lemma B.4.7
$\|\Omega_2\| \vdash^{\mathsf{lf}} \|\omega_1 \circ \omega_2\| : \|\Omega\|$	by Lemma B.4.7
$\|\Omega_2\| \vdash^{\mathsf{lf}} \boldsymbol{x}[\|\omega_1 \circ \omega_2\|] : \boldsymbol{A}[\|\omega_1 \circ \omega_2\|]$	by Lemma B.1.12.1
$\boldsymbol{x}[\omega_1 \circ \omega_2]$	
$= \boldsymbol{x}[\|\omega_1 \circ \omega_2\|]$	by Subst. App. (Def. A.2.4)
$= \boldsymbol{x}[\|\omega_1\| \circ^{\mathsf{lf}} \|\omega_2\|]$	by Lemma B.11.3
$= \boldsymbol{x}[\|\omega_1\|][\|\omega_2\|]$	by Lemma B.1.16
$= \boldsymbol{x}[\omega_1][\omega_2]$	by Subst. App. (Def. A.2.4)

Case: $\boldsymbol{M}$ and $\mathcal{E} = \dfrac{\Omega\ \mathsf{ctx} \quad \|\Omega\| \vdash^{\mathbf{lf}} \boldsymbol{M} : \boldsymbol{A}}{\Omega \vdash \boldsymbol{M} \in \boldsymbol{A}}\ \mathsf{isLF}$

$\Omega_1 \vdash \omega_1 : \Omega$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$\Omega_2 \vdash (\omega_1 \circ \omega_2) : \Omega$	by Lemma B.18.1
$\|\Omega_1\| \vdash^{\mathbf{lf}} \|\omega_1\| : \|\Omega\|$	by Lemma B.4.7
$\|\Omega_2\| \vdash^{\mathbf{lf}} \|\omega_2\| : \|\Omega_1\|$	by Lemma B.4.7
$\|\Omega_2\| \vdash^{\mathbf{lf}} \|\omega_1 \circ \omega_2\| : \|\Omega\|$	by Lemma B.4.7
$\|\Omega_2\| \vdash^{\mathbf{lf}} \boldsymbol{M}[\|\omega_1 \circ \omega_2\|] : \boldsymbol{A}[\|\omega_1 \circ \omega_2\|]$	by Lemma B.1.12.1
$\boldsymbol{M}[\omega_1 \circ \omega_2]$	
$= \boldsymbol{M}[\|\omega_1 \circ \omega_2\|]$	by Subst. App. (Def. A.2.4)
$= \boldsymbol{M}[\|\omega_1\| \circ^{\mathbf{lf}} \|\omega_2\|]$	by Lemma B.11.3
$= \boldsymbol{M}[\|\omega_1\|][\|\omega_2\|]$	by Lemma B.1.16
$= \boldsymbol{M}[\omega_1][\omega_2]$	by Subst. App. (Def. A.2.4)

Case: $(\mathrm{fn}\ \overline{c})$ and

$$\mathcal{E} = \frac{\Omega\ \mathsf{ctx} \quad \Omega \vdash \tau\ \mathsf{wff} \quad \text{for all } {}_{c_i \in \overline{c}}(\Omega \vdash c_i \in \tau) \quad \overset{\mathcal{E}_1}{\Omega \vdash \overline{c}\ \text{covers}\ \tau}}{\Omega \vdash \mathrm{fn}\ \overline{c} \in \tau}\ \mathsf{impI}$$

$\Omega_1 \vdash \omega_1 : \Omega$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
for all ${}_{c_i \in \overline{c}}(c_i[\omega_1][\omega_2] = c_i[\omega_1 \circ \omega_2])$	by i.h. on c_i with $\mathcal{E}_1$ (for all $c_i \in \overline{c}$)
$(\mathrm{fn}\ \overline{c})[\omega_1][\omega_2] = (\mathrm{fn}\ \overline{c})[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: $(e\ \overline{f})$ and $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Omega \vdash e \in \forall\overline{\alpha \in \delta}.\ \tau} \qquad \overset{\mathcal{E}_2}{\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}}}{\Omega \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$

$\Omega_1 \vdash \omega_1 : \Omega$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$e[\omega_1][\omega_2] = e[\omega_1 \circ \omega_2]$	by i.h. on e with $\mathcal{E}_1$
$\overline{f}[\omega_1][\omega_2] = \overline{f}[\omega_1 \circ \omega_2]$	by i.h. on $\overline{f}$ with $\mathcal{E}_2$
$(e\ \overline{f})[\omega_1][\omega_2]$	
$= (e[\omega_1][\omega_2])\ (\overline{f}[\omega_1][\omega_2])$	by Subst. App. (Def. A.2.4)
$= (e[\omega_1 \circ \omega_2])\ (\overline{f}[\omega_1 \circ \omega_2])$	by above
$= (e\ \overline{f})[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: $(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)$ and $\mathcal{E} = \dfrac{\mathcal{E}_1}{\Omega \vdash \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{new}$

$\Omega_1 \vdash \omega_1 : \Omega$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$(\Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})\ \mathsf{ctx}$ by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \boldsymbol{A}^{\#}\ \mathsf{wff}$ by inversion using ctxAddNew
$\Omega_1 \vdash \boldsymbol{A}^{\#}[\omega_1]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_2 \vdash \boldsymbol{A}^{\#}[\omega_1][\omega_2]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1] \vdash (\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}) : \Omega, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$ by tpSubIndNew
$\Omega_2, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1][\omega_2] \vdash (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x}) : \Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_1]$ by tpSubIndNew
$e[\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}][\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x}] = e[(\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x})]$ by i.h. on e with $\mathcal{E}_1$
$\Omega_2 \vdash (\omega_1 \circ \omega_2) : \Omega$ by Lemma B.18.1
$\boldsymbol{A}^{\#}[\omega_1][\omega_2] = \boldsymbol{A}^{\#}[\omega_1 \circ \omega_2]$ by Lemma B.12.1
$(\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)[\omega_1][\omega_2]$
$= \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1][\omega_2].\ e[\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}][\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x}]$ by Subst. App. (Def. A.2.4)
$= \nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}[\omega_1 \circ \omega_2].\ e[(\uparrow_{\boldsymbol{x}} \omega_1, \boldsymbol{x}/\boldsymbol{x}) \circ (\uparrow_{\boldsymbol{x}} \omega_2, \boldsymbol{x}/\boldsymbol{x})]$ by above
$= (\nu \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ e)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $(\nu u{\in}\mathcal{W}.\ e)$ and $\mathcal{E} = \dfrac{\mathcal{E}_1}{\Omega \vdash \nu u{\in}\mathcal{W}.\ e \in \nabla \mathcal{W}.\ \tau}\ \mathsf{newW}$

$\Omega_1 \vdash \omega_1 : \Omega$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$(\Omega, u\overset{\triangledown}{\in}\mathcal{W})\ \mathsf{ctx}$ by Lemma B.3.1 on $\mathcal{E}_1$
$\mathcal{W}\ \mathsf{world}$ by inversion using ctxAddWorld
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_u \omega_1, u/u) : \Omega, u\overset{\triangledown}{\in}\mathcal{W}$ by tpSubWorld
$\Omega_2, u\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_u \omega_2, u/u) : \Omega_1, u\overset{\triangledown}{\in}\mathcal{W}$ by tpSubWorld
$e[\uparrow_u \omega_1, u/u][\uparrow_u \omega_2, u/u] = e[(\uparrow_u \omega_1, u/u) \circ (\uparrow_u \omega_2, u/u)]$ by i.h. on e with $\mathcal{E}_1$
$(\nu u{\in}\mathcal{W}.\ e)[\omega_1][\omega_2]$
$= \nu u{\in}\mathcal{W}.\ e[\uparrow_u \omega_1, u/u][\uparrow_u \omega_2, u/u]$ by Subst. App. (Def. A.2.4)
$= \nu u{\in}\mathcal{W}.\ e[(\uparrow_u \omega_1, u/u) \circ (\uparrow_u \omega_2, u/u)]$ by above
$= (\nu u{\in}\mathcal{W}.\ e)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $(e\backslash\boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_B)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A \vdash e \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_A, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_A, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_B)\end{array}}{\Omega_A, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_B \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_A}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$\Omega_1 \vdash \omega_1 : \Omega_A, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_B$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
There exists an Ω'_A, Ω'_B, $\boldsymbol{x_0}$, and ω_A such that
$\Omega'_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A] \vdash (\uparrow_{\boldsymbol{x_0}}\omega_A, \boldsymbol{x_0}/\boldsymbol{x}) : \Omega_A, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$
and $\Omega_1 = \Omega'_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A], \Omega'_B$
and $\omega_A \leq \omega_1$ without using leSubMiddle or leSubMiddleShift
and $(\uparrow_{\boldsymbol{x_0}}\omega_A, \boldsymbol{x_0}/\boldsymbol{x}) \leq \omega_1$
and $(\Omega'_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A]) \leq (\Omega'_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A], \Omega'_B)$. by Lemma B.16.1
There exists an Ω''_A, Ω''_B, $\boldsymbol{x_1}$, and ω_B such that
$\Omega''_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A][\omega_B] \vdash (\uparrow_{\boldsymbol{x_1}}\omega_B, \boldsymbol{x_1}/\boldsymbol{x_0}) : \Omega'_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A]$
and $\Omega_2 = \Omega''_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A][\omega_B], \Omega''_B$
and $\omega_B \leq \omega_2$ without using leSubMiddle or leSubMiddleShift
and $(\uparrow_{\boldsymbol{x_1}}\omega_A, \boldsymbol{x_1}/\boldsymbol{x_0}) \leq \omega_2$
and $(\Omega''_A, \boldsymbol{x_1}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A][\omega_B]) \leq (\Omega''_A, \boldsymbol{x_0}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}[\omega_A][\omega_B], \Omega''_B)$.
by Lemma B.16.1
$\Omega'_A \vdash \omega_A : \Omega_A$ by inversion using tpSubIndNew
$\Omega''_A \vdash \omega_B : \Omega'_A$ by inversion using tpSubIndNew
$e[\omega_A][\omega_B] = e[\omega_A \circ \omega_B]$ by i.h. on e with $\mathcal{E}_1$
$\boldsymbol{x}[\omega_1] = \boldsymbol{x_0}$ by Subst. App. (Def. A.2.4) and above
$\boldsymbol{x_0}[\omega_2] = \boldsymbol{x_1}$ by Subst. App. (Def. A.2.4) and above
$\boldsymbol{x}[\omega_1 \circ \omega_2] = \boldsymbol{x}[\omega_1][\omega_2]$ by Lemma B.18.2
$\Omega_A\ \mathsf{ctx}$ by inversion on assumption $(\Omega_A, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_B)\ \mathsf{ctx}$
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_2)$ by Lemma B.18.5
$(e\backslash\boldsymbol{x})[\omega_1][\omega_2]$
$= e[\omega_1][\omega_2]\backslash(\boldsymbol{x}[\omega_1][\omega_2])$ by Subst. App. (Def. A.2.4)
$= (e[\omega_A][\omega_B])\backslash(\boldsymbol{x}[\omega_1][\omega_2])$ by Lemma B.7.5
$= (e[\omega_A \circ \omega_B])\backslash(\boldsymbol{x}[\omega_1 \circ \omega_2])$ by above
$= (e[\omega_1 \circ \omega_2])\backslash(\boldsymbol{x}[\omega_1 \circ \omega_2])$ by Lemma B.7.5
$= (e\backslash\boldsymbol{x})[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $(e\backslash u)$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_A \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)\end{array}}{\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B \vdash e\backslash u \in \tau}\ \mathsf{popW}$

$\Omega_1 \vdash \omega_1 : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
There exists an Ω'_A, Ω'_B, u_0, and ω_A such that
$\Omega'_A, u_0\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u_0}\omega_A, u_0/u) : \Omega_A, u\overset{\triangledown}{\in}\mathcal{W}$
and $\Omega_1 = \Omega'_A, u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega'_B$
and $\omega_A \leq \omega_1$ without using leSubMiddle or leSubMiddleShift
and $(\uparrow_{u_0}\omega_A, u_0/u) \leq \omega_1$
and $(\Omega'_A, u_0\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega'_A, u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega'_B)$. by Lemma B.16.2
There exists an Ω''_A, Ω''_B, u_1, and ω_B such that
$\Omega''_A, u_0\overset{\triangledown}{\in}\mathcal{W} \vdash (\uparrow_{u_1}\omega_B, u_1/u_0) : \Omega'_A, u_0\overset{\triangledown}{\in}\mathcal{W}$
and $\Omega_2 = \Omega''_A, u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega''_B$
and $\omega_B \leq \omega_2$ without using leSubMiddle or leSubMiddleShift
and $(\uparrow_{u_1}\omega_A, u_1/u_0) \leq \omega_2$
and $(\Omega''_A, u_1\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega''_A, u_0\overset{\triangledown}{\in}\mathcal{W}, \Omega''_B)$. by Lemma B.16.2
$\Omega'_A \vdash \omega_A : \Omega_A$ by inversion using tpSubWorld
$\Omega''_A \vdash \omega_B : \Omega'_A$ by inversion using tpSubWorld
$e[\omega_A][\omega_B] = e[\omega_A \circ \omega_B]$ by i.h. on e with $\mathcal{E}_1$
$u[\omega_1] = u_0$ by Subst. App. (Def. A.2.4) and above
$u_0[\omega_2] = u_1$ by Subst. App. (Def. A.2.4) and above
$u[\omega_1 \circ \omega_2] = u[\omega_1][\omega_2]$ by Lemma B.18.2
$\Omega_A\ \mathsf{ctx}$ by inversion on assumption $(\Omega_A, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_B)\ \mathsf{ctx}$
$(\omega_A \circ \omega_B) \leq (\omega_1 \circ \omega_2)$ by Lemma B.18.5
$(e\backslash u)[\omega_1][\omega_2]$
$= e[\omega_1][\omega_2]\backslash(u[\omega_1][\omega_2])$ by Subst. App. (Def. A.2.4)
$= (e[\omega_A][\omega_B])\backslash(u[\omega_1][\omega_2])$ by Lemma B.7.5
$= (e[\omega_A \circ \omega_B])\backslash(u[\omega_1 \circ \omega_2])$ by above
$= (e[\omega_1 \circ \omega_2])\backslash(u[\omega_1 \circ \omega_2])$ by Lemma B.7.5
$= (e\backslash u)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $(e,\ f)$ and

$$\mathcal{E} = \dfrac{\begin{array}{ccc}\mathcal{E}_1 & & \mathcal{E}_2\\ \Omega \vdash (\exists \alpha{\in}\delta.\ \tau)\ \mathsf{wff} & \Omega \vdash e \in \delta & \Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]\end{array}}{\Omega \vdash (e,\ f) \in \exists \alpha{\in}\delta.\ \tau}\ \mathsf{pairI}$$

$\Omega_1 \vdash \omega_1 : \Omega$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$e[\omega_1][\omega_2] = e[\omega_1 \circ \omega_2]$ by i.h. on e with $\mathcal{E}_1$
$f[\omega_1][\omega_2] = f[\omega_1 \circ \omega_2]$ by i.h. on f with $\mathcal{E}_2$
$(e,\ f)[\omega_1][\omega_2]$
$= (e[\omega_1][\omega_2],\ f[\omega_1][\omega_2])$ by Subst. App. (Def. A.2.4)
$= (e[\omega_1 \circ \omega_2],\ f[\omega_1 \circ \omega_2])$ by above
$= (e,\ f)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $(\mu u{\in}\tau.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega, u{\in}\tau \vdash e \in \tau\end{array}}{\Omega \vdash \mu u{\in}\tau.\ e \in \tau}\ \mathsf{fix}$

$\Omega_1 \vdash \omega_1 : \Omega$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$(\Omega, u{\in}\tau)\ \mathsf{ctx}$ by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \tau\ \mathsf{wff}$ by inversion using ctxAdd
$\Omega_1 \vdash \tau[\omega_1]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_2 \vdash \tau[\omega_1][\omega_2]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_1, u{\in}\tau[\omega_1] \vdash \uparrow_u \omega_1 : \Omega$ by tpSubShift
$(\Omega_1, u{\in}\tau[\omega_1])\ \mathsf{ctx}$ by ctxAdd
$\Omega_1, u{\in}\tau[\omega_1] \vdash u \in \tau[\omega_1]$ by Lemma B.5.1
$\Omega_1, u{\in}\tau[\omega_1] \vdash u \in \tau[\uparrow_u \omega_1]$ by Lemma B.5.4
$\Omega_1, u{\in}\tau[\omega_1] \vdash (\uparrow_u \omega_1, u/u) : \Omega, u{\in}\tau$ by tpSubInd
$\Omega_2, u{\in}\tau[\omega_1][\omega_2] \vdash (\uparrow_u \omega_2, u/u) : \Omega_1, u{\in}\tau[\omega_1]$
by tpSubShift, tpSubInd, ctxAdd, Lemma B.5.1, and Lemma B.5.4
$e[\uparrow_u \omega_1, u/u][\uparrow_u \omega_2, u/u] = e[(\uparrow_u \omega_1, u/u) \circ (\uparrow_u \omega_2, u/u)]$ by i.h. on e with $\mathcal{E}_1$
$\Omega_2 \vdash (\omega_1 \circ \omega_2) : \Omega$ by Lemma B.18.1
$\tau[\omega_1][\omega_2] = \tau[\omega_1 \circ \omega_2]$ by Lemma B.12.1
$(\mu u{\in}\tau.\ e)[\omega_1][\omega_2]$
$= \mu u{\in}\tau[\omega_1][\omega_2].\ e[\uparrow_u \omega_1, u/u][\uparrow_u \omega_2, u/u]$
by Subst. App. (Def. A.2.4)
$= \mu u{\in}\tau[\omega_1 \circ \omega_2].\ e[(\uparrow_u \omega_1, u/u) \circ (\uparrow_u \omega_2, u/u)]$ by above
$= (\mu u{\in}\tau.\ e)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

. . . . . .

Case: $(\epsilon\alpha{\in}\delta.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega \vdash \tau\ \mathsf{wff} \qquad \Omega, \alpha{\in}\delta \vdash c \in \tau\end{array}}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau}\ \mathsf{cEps}$

$\Omega_1 \vdash \omega_1 : \Omega$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$(\Omega, \alpha{\in}\delta)\ \mathsf{ctx}$ by Lemma B.3.1 on $\mathcal{E}_1$
$\Omega \vdash \delta\ \mathsf{wff}$ by inversion using ctxAdd
$\Omega_1 \vdash \delta[\omega_1]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_2 \vdash \delta[\omega_1][\omega_2]\ \mathsf{wff}$ by Lemma B.6.1
$\Omega_1, \alpha{\in}\delta[\omega_1] \vdash \uparrow_\alpha \omega_1 : \Omega$ by $\mathsf{tpSubShift}$
$(\Omega_1, \alpha{\in}\delta[\omega_1])\ \mathsf{ctx}$ by ctxAdd
$\Omega_1, \alpha{\in}\delta[\omega_1] \vdash \alpha \in \delta[\omega_1]$ by Lemma B.5.1
$\Omega_1, \alpha{\in}\delta[\omega_1] \vdash \alpha \in \delta[\uparrow_\alpha \omega_1]$ by Lemma B.5.4
$\Omega_1, \alpha{\in}\delta[\omega_1] \vdash (\uparrow_\alpha \omega_1, \alpha/\alpha) : \Omega, \alpha{\in}\delta$ by $\mathsf{tpSubInd}$
$\Omega_2, \alpha{\in}\delta[\omega_1][\omega_2] \vdash (\uparrow_\alpha \omega_2, \alpha/\alpha) : \Omega_1, \alpha{\in}\delta[\omega_1]$
by $\mathsf{tpSubShift}$, $\mathsf{tpSubInd}$, ctxAdd, Lemma B.5.1, and Lemma B.5.4
$c[\uparrow_\alpha \omega_1, \alpha/\alpha][\uparrow_\alpha \omega_2, \alpha/\alpha] = c[(\uparrow_\alpha \omega_1, \alpha/\alpha) \circ (\uparrow_\alpha \omega_2, \alpha/\alpha)]$ by i.h. on c with $\mathcal{E}_1$
$\Omega_2 \vdash (\omega_1 \circ \omega_2) : \Omega$ by Lemma B.18.1
$\delta[\omega_1][\omega_2] = \delta[\omega_1 \circ \omega_2]$ by Lemma B.12.1
$(\epsilon\alpha{\in}\delta.\ c)[\omega_1][\omega_2]$
$= \epsilon\alpha{\in}\delta[\omega_1][\omega_2].\ c[\uparrow_\alpha \omega_1, \alpha/\alpha][\uparrow_\alpha \omega_2, \alpha/\alpha]$
by Subst. App. (Def. A.2.4)
$= \epsilon\alpha{\in}\delta[\omega_1 \circ \omega_2].\ c[(\uparrow_\alpha \omega_1, \alpha/\alpha) \circ (\uparrow_\alpha \omega_2, \alpha/\alpha)]$ by above
$= (\epsilon\alpha{\in}\delta.\ c)[\omega_1 \circ \omega_2]$ by Subst. App. (Def. A.2.4)

Case: $(\nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash c \in \tau\end{array}}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \in \nabla\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{cNew}$

See Case for new (above)

Case: $(\overline{e} \mapsto f)$ and

$$\mathcal{E} = \frac{\overset{\mathcal{E}_1}{\Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e}/\overline{\alpha}]} \qquad \overset{\mathcal{E}_2}{\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha \in \delta}} \qquad \Omega \vdash \forall \overline{\alpha \in \delta}.\ \tau\ \mathsf{wff}}{\Omega \vdash \overline{e} \mapsto f \in \forall \overline{\alpha \in \delta}.\ \tau}\ \mathsf{cMatch}$$

$\Omega_1 \vdash \omega_1 : \Omega$ by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$ by assumption
$f[\omega_1][\omega_2] = f[\omega_1 \circ \omega_2]$ by i.h. on f with $\mathcal{E}_1$
$\overline{e}[\omega_1][\omega_2] = \overline{e}[\omega_1 \circ \omega_2]$ by i.h. on $\overline{e}$ with $\mathcal{E}_2$

$$\begin{aligned}
&(\overline{e} \mapsto f)[\omega_1][\omega_2] \\
&\quad = (\overline{e}[\omega_1][\omega_2]) \mapsto (f[\omega_1][\omega_2]) && \text{by Subst. App. (Def. A.2.4)} \\
&\quad = (\overline{e}[\omega_1 \circ \omega_2]) \mapsto (f[\omega_1 \circ \omega_2]) && \text{by above} \\
&\quad = (\overline{e} \mapsto f)[\omega_1 \circ \omega_2] && \text{by Subst. App. (Def. A.2.4)}
\end{aligned}$$

Case: $(c\backslash \boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x} \overset{\nabla}{\in} \boldsymbol{A}^\#, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash c \in \nabla \boldsymbol{x'} \in \boldsymbol{A}^\#.\ \tau \\ (\Omega_1, \boldsymbol{x} \overset{\nabla}{\in} \boldsymbol{A}^\#) \leq (\Omega_1, \boldsymbol{x} \overset{\nabla}{\in} \boldsymbol{A}^\#, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x} \overset{\nabla}{\in} \boldsymbol{A}^\#, \Omega_2 \vdash c\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{cPop}$

See Case for pop (above)

. . . . . .

Case: nil and $\mathcal{E} = \Omega \vdash \mathrm{id}_\Omega : \Omega$

$\Omega_1 \vdash \omega_1 : \Omega$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$\Omega_2 \vdash (\omega_1 \circ \omega_2) : \Omega$	by Lemma B.18.1
$nil[\omega_1][\omega_2] = nil[\omega_1 \circ \omega_2]$	by Subst. App. (Def. A.2.4)

Case: $(\overline{f'}; f)$ and $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'} : \Omega, \overline{\alpha' \in \delta'}} \qquad \overset{\mathcal{E}_2}{\Omega \vdash f \in \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}]}}{\Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega, \overline{\alpha' \in \delta'}, \alpha \in \delta}\ \mathsf{tpSubInd}$

$\Omega_1 \vdash \omega_1 : \Omega$	by assumption
$\Omega_2 \vdash \omega_2 : \Omega_1$	by assumption
$\overline{f'}[\omega_1][\omega_2] = \overline{f'}[\omega_1 \circ \omega_2]$	by i.h. on $\overline{f'}$ with $\mathcal{E}_1$
$f[\omega_1][\omega_2] = f[\omega_1 \circ \omega_2]$	by i.h. on f with $\mathcal{E}_2$
$(\overline{f'}; f)[\omega_1][\omega_2]$	
$= (\overline{f'}[\omega_1][\omega_2]); (f[\omega_1][\omega_2])$	by induction on $\overline{f'}$ using Subst. App. (Def. A.2.4)
$= (\overline{f'}[\omega_1 \circ \omega_2]); (f[\omega_1 \circ \omega_2])$	by above
$= (\overline{f'}; f)[\omega_1 \circ \omega_2]$	by induction on $\overline{f'}$ using Subst. App. (Def. A.2.4)

□

B.19 Meta-Theory: Type Preservation

Lemma B.19.1 (Substitution on Type Only Uses LF)**.**

- $\delta[\omega] = \delta[\|\omega\|^{\nabla}]$.
- $\overline{\alpha{\in}\delta}[\omega] = \overline{\alpha{\in}\delta}[\|\omega\|^{\nabla}]$.

Proof. By induction on τ and $\overline{\alpha{\in}\delta}$ following the rules for Substitution Application (Def. A.2.4) utilizing Lemma B.4.2. □

Lemma B.19.2 (Evaluation Has No Effect on Type)**.**
If $\Omega \vdash e \rightarrow f$ *and* $\Omega \vdash e \in \delta[\omega_A]$ *and* $\Omega \vdash (\omega_A, e/\alpha, \omega_B) : \Omega_A, \alpha{\in}\delta, \Omega_B$ *and* $\tau[\omega_A, e/\alpha, \omega_B]$ *exists, then:*

$$\tau[\omega_A, e/\alpha, \omega_B] = \tau[\omega_A, f/\alpha, \omega_B]$$

Proof. By case analysis.

Case: $(\alpha{\in}\delta) = (\boldsymbol{x}{\in}\boldsymbol{A})$ or $(\boldsymbol{x}{\in}\boldsymbol{A}^{\#})$

$\Omega \vdash e \in \delta[\omega_A]$ by assumption
$e = \boldsymbol{M}$ or $\boldsymbol{x}$ by inversion using typing rules
This case is vacuously true as we see by inspection of evaluation rules that there is no way that $\Omega \vdash e \rightarrow f$, which must hold by assumption.

Case: $(\alpha{\in}\delta) = (u{\in}\sigma)$

$\tau[\omega_A, e/u, \omega_B]$
$= \tau[\|\omega_A, e/u, \omega_B\|^{\nabla}]$ by Lemma B.19.1
$= \tau[\|\omega_A, f/u, \omega_B\|^{\nabla}]$ by Casting (Def. B.2.1)
$= \tau[\omega_A, f/u, \omega_B]$ by Lemma B.19.1

□

Lemma B.19.3 (Evaluation Allowed Inside Substitution)**.**
If $\Omega \vdash \omega_A, e/\alpha, \omega_B : \Omega_A, \alpha{\in}\delta, \Omega_B$ *and* $\Omega \vdash e \in \delta[\omega_A]$ *and* $\Omega \vdash f \in \delta[\omega_A]$ *and* $\Omega \vdash e \to f$ *and* Ω_B *only contains declarations of the form* $\alpha'{\in}\delta'$*,*[5] *then* $\Omega \vdash \omega_A, f/\alpha, \omega_B : \Omega_A, \alpha{\in}\delta, \Omega_B$*.*

Proof. By induction on Ω_B.

Case: $\cdot$

$\Omega \vdash \omega_A, e/\alpha : \Omega_A, \alpha{\in}\delta$	by assumption
$\Omega \vdash f \in \delta[\omega_A]$	by assumption
$\Omega \vdash \omega_A : \Omega_A$	by inversion using tpSubInd
$\Omega \vdash \omega_A, f/\alpha : \Omega_A, \alpha{\in}\delta$	by tpSubInd

Case: $\Omega_B, \alpha_2{\in}\delta_2$

$\Omega \vdash \omega_A, e/\alpha, \omega_B, e_2/\alpha_2 : \Omega_A, \alpha{\in}\delta, \Omega_B, \alpha_2{\in}\delta_2$	by assumption
$\Omega \vdash e \in \delta[\omega_A]$	by assumption
$\Omega \vdash f \in \delta[\omega_A]$	by assumption
$\Omega \vdash e \to f$	by assumption
Ω_B only contains declarations of the form $\alpha'{\in}\delta'$	by assumption
$\Omega \vdash \omega_A, e/\alpha, \omega_B : \Omega_A, \alpha{\in}\delta, \Omega_B$	by inversion using tpSubInd
$\Omega \vdash e_2 \in \delta_2[\omega_A, e/\alpha, \omega_B]$	by inversion using tpSubInd
$\Omega \vdash e_2 \in \delta_2[\omega_A, f/\alpha, \omega_B]$	by Lemma B.19.2
$\Omega \vdash \omega_A, f/\alpha, \omega_B : \Omega_A, \alpha{\in}\delta, \Omega_B$	by i.h. on Ω_B
$\Omega \vdash \omega_A, f/\alpha, \omega_B, e_2/\alpha_2 : \Omega_A, \alpha{\in}\delta, \Omega_B, \alpha_2{\in}\delta_2$	by tpSubInd

□

[5] This is also forced to be true by the form of the substitution.

We are now ready to prove our theorem.

Theorem B.19.4 (Type Preservation)**.**

- *If* $\Omega \vdash e \in \delta$ *and* $\Omega \vdash e \to f$*, then* $\Omega \vdash f \in \delta$*.*
- *If* $\Omega \vdash c \in \tau$ *and* $\Omega \vdash c \to c'$*, then* $\Omega \vdash c' \in \tau$*.*
- *If* $\Omega \vdash c \in \tau$ *and* $\Omega \vdash c \xrightarrow{*} c'$*, then* $\Omega \vdash c' \in \tau$*.*

Proof. By induction on
$\mathcal{E} :: \Omega \vdash e \to f$ and $\mathcal{E} :: \Omega \vdash c \to c'$ and $\mathcal{E} :: \Omega \vdash c \xrightarrow{*} c'$.

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash e \to f\end{array}}{\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e \to \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ f}\ \mathsf{redNew}$

$\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ e \in \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau$	by assumption and inversion using new
$\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash e \in \tau$	by inversion using new
$\Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash f \in \tau$	by i.h. on $\mathcal{E}_1$
$\Omega \vdash \nu \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ f \in \nabla \boldsymbol{x} \in \boldsymbol{A}^{\#}.\ \tau$	by new

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash e \to f\end{array}}{\Omega \vdash \nu u \in \mathcal{W}.\ e \to \nu u \in \mathcal{W}.\ f}\ \mathsf{redNewW}$

$\Omega \vdash \nu u \in \mathcal{W}.\ e \in \nabla \mathcal{W}.\ \tau$	by assumption and inversion using newW
$\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash e \in \tau$	by inversion using newW
$\Omega, u \overset{\triangledown}{\in} \mathcal{W} \vdash f \in \tau$	by i.h. on $\mathcal{E}_1$
$\Omega \vdash \nu u \overset{\triangledown}{\in} \mathcal{W}.\ f \in \nabla \mathcal{W}.\ \tau$	by newW

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash e \to e'\end{array}}{\Omega \vdash (e,\ f) \to (e',\ f)}\ \mathsf{redPairL}$

$\Omega \vdash (e,\ f) \in \exists \alpha{\in}\delta.\ \tau$	by assumption and inversion using pairI
$\Omega \vdash e \in \delta$	by inversion using pairI
$\Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]$	by inversion using pairI
$\Omega \vdash \exists \alpha{\in}\delta.\ \tau$ wff	by inversion using pairI
$\Omega \vdash e' \in \delta$	by i.h. on $\mathcal{E}_1$
Ω ctx	by Lemma B.3.1
$\Omega \vdash \delta$ wff	by Lemma B.8.1
$\delta = \delta[\mathrm{id}_\Omega]$	by Lemma B.5.2
$\Omega \vdash (\mathrm{id}_\Omega, e/\alpha) : \Omega, \alpha{\in}\delta$	by Lemma B.5.5 and tpSubInd
$\tau[\mathrm{id}_\Omega, e/\alpha] = \tau[\mathrm{id}_\Omega, e'/\alpha]$	by Lemma B.19.2 with $\mathcal{E}_1$
$\Omega \vdash f \in \tau[\mathrm{id}_\Omega, e'/\alpha]$	by above
$\Omega \vdash (e',\ f) \in \exists \alpha{\in}\delta.\ \tau$	by pairI

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash f \to f'\end{array}}{\Omega \vdash (e,\ f) \to (e,\ f')}\ \mathsf{redPairR}$

$\Omega \vdash (e,\ f) \in \exists \alpha{\in}\delta.\ \tau$	by assumption and inversion using pairI
$\Omega \vdash e \in \delta$	by inversion using pairI
$\Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]$	by inversion using pairI
$\Omega \vdash \exists \alpha{\in}\delta.\ \tau$ wff	by inversion using pairI
$\Omega \vdash f' \in \tau[\mathrm{id}_\Omega, e/\alpha]$	by i.h. on $\mathcal{E}_1$
$\Omega \vdash (e,\ f') \in \exists \alpha{\in}\delta.\ \tau$	by pairI

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash e \to e'\end{array}}{\Omega \vdash e\ \overline{f} \to e'\ \overline{f}}\ \mathsf{redAppL}$

$\Omega \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]$	by assumption and inversion using impE
$\Omega \vdash e \in \forall \overline{\alpha{\in}\delta}.\ \tau$	by inversion using impE
$\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta}$	by inversion using impE
$\Omega \vdash e' \in \forall \overline{\alpha{\in}\delta}.\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega \vdash e'\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]$	by impE

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash f_i \to f_i'\end{array}}{\Omega \vdash e\ (\overline{f_1}; f_i; \overline{f_n}) \to e\ (\overline{f_1}; f_i'; \overline{f_n})}\ \mathsf{redAppR}$

$\Omega \vdash e\ (\overline{f_1}; f_i; \overline{f_n}) \in \tau[\mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}, f_i/\alpha_i, \overline{f_n}/\overline{\alpha_n}]$
by assumption and inversion using impE

$\Omega \vdash e \in \forall(\overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}).\ \tau$ by inversion using impE

$\Omega \vdash \mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}, f_i/\alpha_i, \overline{f_n}/\overline{\alpha_n} : \Omega, \overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}$
by inversion using impE

$\Omega \vdash f_i \in \delta_i[\mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}]$ by inversion using tpSubInd

$\Omega \vdash f_i' \in \delta_i[\mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}]$ by i.h. on $\mathcal{E}_1$

$\Omega \vdash \mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}, f_i'/\alpha_i, \overline{f_n}/\overline{\alpha_n} : \Omega, \overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}$
by Lemma B.19.3

$\Omega \vdash e\ (\overline{f_1}; f_i'; \overline{f_n}) \in \tau[\mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}, f_i'/\alpha_i, \overline{f_n}/\overline{\alpha_n}]$ by impE

$\tau[\mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}, f_i/\alpha_i, \overline{f_n}/\overline{\alpha_n}]$
$= \tau[\mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}, f_i'/\alpha_i, \overline{f_n}/\overline{\alpha_n}]$ by Lemma B.19.2 with $\mathcal{E}_1$

$\Omega \vdash e\ (\overline{f_1}; f_i'; \overline{f_n}) \in \tau[\mathrm{id}_\Omega, \overline{f_1}/\overline{\alpha_1}, f_i/\alpha_i, \overline{f_n}/\overline{\alpha_n}]$ by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \vdash e \to f\end{array}}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^\#, \Omega_2 \vdash e \backslash \boldsymbol{x} \to f \backslash \boldsymbol{x}}\ \mathsf{redPop}$

$\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^\#, \Omega_2 \vdash e \backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$
by assumption and inversion using pop

$\Omega_1 \vdash e \in \nabla \boldsymbol{x'} \in \boldsymbol{A}^\#.\ \tau$ by inversion using pop

$(\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^\#, \Omega_2)\ \mathsf{ctx}$ by inversion using pop

$(\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^\#) \leq (\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^\#, \Omega_2)$ by inversion using pop

$\Omega_1 \vdash f \in \nabla \boldsymbol{x'} \in \boldsymbol{A}^\#.\ \tau$ by i.h. on $\mathcal{E}_1$

$\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^\#, \Omega_2 \vdash f \backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$ by pop

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \vdash e \to f\end{array}}{\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \to f\backslash u}\ \mathsf{redPopW}$

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \in \tau$	by assumption and inversion using popW
$\Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau$	by inversion using popW
$\mathcal{W} \leq \mathcal{W}_2$	by inversion using popW
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ ctx	by inversion using popW
$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$	by inversion using popW
$\Omega_1 \vdash f \in \nabla\mathcal{W}_2.\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash f\backslash u \in \tau$	by popW

Case: $\mathcal{E} = \dfrac{}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu\boldsymbol{x'}\in\boldsymbol{A}^{\#}.\ e)\backslash\boldsymbol{x} \to e[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{redPopElim}$

$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu\boldsymbol{x'}\in\boldsymbol{A}^{\#}.\ e)\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by assumption and inversion using pop
$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ ctx	by inversion using pop
$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$	by inversion using pop
$\Omega_1 \vdash \nu\boldsymbol{x'}\in\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x'}\in\boldsymbol{A}^{\#}.\ \tau$	by inversion using pop
$\Omega_1, \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau$	by inversion using new
$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#})$ ctx	by inversion
$\Omega_1 \vdash \boldsymbol{A}^{\#}$ wff	by inversion using ctxAddNew
Ω_1 ctx	by inversion using ctxAddNew
$\Omega_1 \vdash \mathrm{id}_{\Omega_1} : \Omega_1$	by Lemma B.5.5
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash (\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}) : \Omega_1, \boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}$	by tpSubIndNew
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#} \vdash e[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}] \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by Lemma B.17.7
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}] \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by Lemma B.14.4

Case: $\mathcal{E} = \dfrac{}{\Omega_1, u\overset{\nabla}{\in}\mathcal{W}, \Omega_2 \vdash (\nu u'{\in}\mathcal{W}_2.\ e)\backslash u \to e[\uparrow_u \mathrm{id}_{\Omega_1}, u/u']}\ \mathsf{redPopElimW}$

$\Omega_1, u\overset{\nabla}{\in}\mathcal{W}, \Omega_2 \vdash (\nu u'{\in}\mathcal{W}_2.\ e)\backslash u \in \tau[\uparrow_u \mathrm{id}_{\Omega_1}, u/u']$	by assumption and inversion using popW
$(\Omega_1, u\overset{\nabla}{\in}\mathcal{W}, \Omega_2)$ ctx	by inversion using popW
$(\Omega_1, u\overset{\nabla}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u\overset{\nabla}{\in}\mathcal{W}, \Omega_2)$	by inversion using popW
$\mathcal{W} \leq \mathcal{W}_2$	by inversion using popW
$\Omega_1 \vdash \nu u'{\in}\mathcal{W}_2.\ e \in \nabla\mathcal{W}_2.\ \tau$	by inversion using popW
$\Omega_1, u'\overset{\nabla}{\in}\mathcal{W}_2 \vdash e \in \tau$	by inversion using newW
$(\Omega_1, u\overset{\nabla}{\in}\mathcal{W})$ ctx	by inversion
$\mathcal{W}$ world	by inversion using ctxAddWorld
Ω_1 ctx	by inversion using ctxAddWorld
$\Omega_1 \vdash \nabla\mathcal{W}_2.\ \tau$ wff	by Lemma B.8.1
$\mathcal{W}_2$ world	by inversion using ∇_{world}wff
$\Omega_1 \vdash \mathrm{id}_{\Omega_1} : \Omega_1$	by Lemma B.5.5
$\Omega_1, u\overset{\nabla}{\in}\mathcal{W}_2 \vdash (\uparrow_u \mathrm{id}_{\Omega_1}, u/u') : \Omega_1, u'\overset{\nabla}{\in}\mathcal{W}_2$	by tpSubWorld
$\Omega_1, u\overset{\nabla}{\in}\mathcal{W}_2 \vdash e[\uparrow_u \mathrm{id}_{\Omega_1}, u/u'] \in \tau[\uparrow_u \mathrm{id}_{\Omega_1}, u/u']$	by Lemma B.17.7
$(\Omega_1, u\overset{\nabla}{\in}\mathcal{W}_2, \Omega_2)$ ctx	by Lemma B.15.2
$(\Omega_1, u\overset{\nabla}{\in}\mathcal{W}_2) \leq_{\mathcal{W}_2} (\Omega_1, u\overset{\nabla}{\in}\mathcal{W}_2, \Omega_2)$	by Lemma B.15.6
$\Omega_1, u\overset{\nabla}{\in}\mathcal{W}_2, \Omega_2 \vdash e[\uparrow_u \mathrm{id}_{\Omega_1}, u/u'] \in \tau[\uparrow_u \mathrm{id}_{\Omega_1}, u/u']$	by Lemma B.14.5
$\Omega_1, u\overset{\nabla}{\in}\mathcal{W}, \Omega_2 \vdash e[\uparrow_u \mathrm{id}_{\Omega_1}, u/u'] \in \tau[\uparrow_u \mathrm{id}_{\Omega_1}, u/u']$	by Lemma B.15.10

Case: $\mathcal{E} = \dfrac{}{\Omega \vdash (\mathrm{fn}\ \overline{c})\backslash \boldsymbol{x} \to \mathrm{fn}\ (\overline{c}\backslash \boldsymbol{x})}\ \mathsf{redPopFn}$

$\Omega \vdash (\mathrm{fn}\ \overline{c})\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by assumption and inversion using pop
$\Omega = \Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}, \Omega_2$	by inversion using pop
Ω ctx	by inversion using pop
$(\Omega_1, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#}) \leq \Omega$	by inversion using pop
$\Omega_1 \vdash (\mathrm{fn}\ \overline{c}) \in \nabla\boldsymbol{x'}\overset{\nabla}{\in}\boldsymbol{A}^{\#}.\ \tau$	by inversion using pop
for all c_i in $\overline{c}$, $\Omega_1 \vdash c_i \in \nabla\boldsymbol{x'}\overset{\nabla}{\in}\boldsymbol{A}^{\#}.\ \tau$	by inversion using impl
$\Omega_1 \vdash \overline{c}$ covers τ	by inversion using impl
for all c_i in $\overline{c}$, $\Omega \vdash c_i\backslash \boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by cPop
for all c_i in $(\overline{c}\backslash \boldsymbol{x})$, $\Omega \vdash c_i \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by above
$\Omega \vdash \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$ wff	by Lemma B.8.1
$\Omega \vdash \overline{c}\backslash \boldsymbol{x}$ covers $\tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by coverPop
$\Omega \vdash \mathrm{fn}\ (\overline{c}\backslash \boldsymbol{x}) \in \tau[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$	by impl

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash c_i \xrightarrow{*} (\overline{v} \mapsto e)\end{array}}{\Omega \vdash (\text{fn } (\overline{c_1}; c_i; \overline{c_n}))\ \overline{v} \to e}\ \mathsf{redexLam}$

$\Omega \vdash (\text{fn } (\overline{c_1}; c_i; \overline{c_n}))\ \overline{v} \in \tau[\text{id}_\Omega, \overline{v}/\overline{\alpha}]$	by assumption and inversion using impE
$\Omega \vdash \text{fn } (\overline{c_1}; c_i; \overline{c_n}) \in \forall\overline{\alpha \in \delta}.\ \tau$	by inversion using impE
$\Omega \vdash c_i \in \forall\overline{\alpha \in \delta}.\ \tau$	by inversion using impI
$\Omega \vdash (\overline{v} \mapsto e) \in \forall\overline{\alpha \in \delta}.\ \tau$	by i.h. on $\mathcal{E}_1$
$\Omega \vdash e \in \tau[\text{id}_\Omega, \overline{v}/\overline{\alpha}]$	by inversion using cMatch

Case: $\mathcal{E} = \dfrac{}{\Omega \vdash \mu u{\in}\tau.\ e \to e[\text{id}_\Omega, \mu u{\in}\tau.\ e/u]}\ \mathsf{redFix}$

$\Omega \vdash \mu u{\in}\tau.\ e \in \tau$	by assumption and inversion using fix
$\Omega, u{\in}\tau \vdash e \in \tau$	by inversion using fix
Ω ctx	by Lemma B.3.1
$\Omega \vdash \tau$ wff	by Lemma B.8.1
$\tau[\text{id}_\Omega] = \tau$	by Lemma B.5.2
$\Omega \vdash \text{id}_\Omega : \Omega$	by Lemma B.5.5
$\Omega \vdash (\text{id}_\Omega, \mu u{\in}\tau.\ e/u) : \Omega, u{\in}\tau$	by tpSubInd
$\Omega \vdash e[\text{id}_\Omega, \mu u{\in}\tau.\ e/u] \in \tau[\text{id}_\Omega, \mu u{\in}\tau.\ e/u]$	by Lemma B.17.7
$\text{id}_\Omega \leq (\text{id}_\Omega, \mu u{\in}\tau.\ e/u)$	by leSubAdd
$\tau[\text{id}_\Omega, \mu u{\in}\tau.\ e/u] = \tau[\text{id}_\Omega]$	by Lemma B.7.4
$\Omega \vdash e[\text{id}_\Omega, \mu u{\in}\tau.\ e/u] \in \tau$	by above

. . . . . .

Case: $\mathcal{E} = \dfrac{\Omega \vdash v \in \delta}{\Omega \vdash \epsilon\alpha{\in}\delta.\ c \to c[\mathrm{id}_\Omega, v/\alpha]}\ \mathsf{redEps}$

$\Omega \vdash \epsilon\alpha{\in}\delta.\ c \in \tau$	by assumption
$\Omega, \alpha{\in}\delta \vdash c \in \tau$	by inversion using cEps
$\Omega \vdash \tau$ wff	by inversion using cEps
$\Omega \vdash v \in \delta$	by assumption
Ω ctx	by Lemma B.3.1
$\Omega \vdash \delta$ wff	by Lemma B.8.1
$\delta[\mathrm{id}_\Omega] = \delta$	by Lemma B.5.2
$\Omega \vdash \mathrm{id}_\Omega : \Omega$	by Lemma B.5.5
$\Omega \vdash (\mathrm{id}_\Omega, v/\alpha) : \Omega, \alpha{\in}\delta$	by tpSubInd
$\Omega \vdash c[\mathrm{id}_\Omega, v/\alpha] \in \tau[\mathrm{id}_\Omega, v/\alpha]$	by Lemma B.17.7
$\tau = \tau[\mathrm{id}_\Omega]$	by Lemma B.5.2
$\mathrm{id}_\Omega \leq (\mathrm{id}_\Omega, v/\alpha)$	by leSubAdd
$\tau[\mathrm{id}_\Omega] = \tau[\mathrm{id}_\Omega, v/\alpha]$	by Lemma B.7.4
$\Omega \vdash c[\mathrm{id}_\Omega, v/\alpha] \in \tau$	by above

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash c \to c'\end{array}}{\Omega \vdash \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c \to \nu\boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ c'}\ \mathsf{redCaseNew}$

See case for redNew (above)

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega \vdash e_i \to e_i'\end{array}}{\Omega \vdash ((\overline{e_1}; e_i; \overline{e_n}) \mapsto f) \to ((\overline{e_1}; e_i'; \overline{e_n}) \mapsto f)}\ \mathsf{redCasePattern}$

$\Omega \vdash (\overline{e_1}; e_i; \overline{e_n}) \mapsto f \in \forall(\overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}).\ \tau$
by assumption and inversion using cMatch

$\Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}, e_i/\alpha_i, \overline{e_n}/\overline{\alpha_n}]$ by inversion using cMatch

$\Omega \vdash \mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}, e_i/\alpha_i, \overline{e_n}/\overline{\alpha_n} : \Omega, \overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}$
by inversion using cMatch

$\Omega \vdash \forall(\overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}).\ \tau$ wff by inversion using cMatch

$\Omega \vdash e_i \in \delta_i[\mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}]$ by inversion using tpSubInd

$\Omega \vdash e_i' \in \delta_i[\mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}]$ by i.h. on $\mathcal{E}_1$

$\Omega \vdash \mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}, e_i'/\alpha_i, \overline{e_n}/\overline{\alpha_n} : \Omega, \overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}$ by Lemma B.19.3

$\tau[\mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}, e_i/\alpha_i, \overline{e_n}/\overline{\alpha_n}]$
$= \tau[\mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}, e_i'/\alpha_i, \overline{e_n}/\overline{\alpha_n}]$ by Lemma B.19.2 with $\mathcal{E}_1$

$\Omega \vdash f \in \tau[\mathrm{id}_\Omega, \overline{e_1}/\overline{\alpha_1}, e_i'/\alpha_i, \overline{e_n}/\overline{\alpha_n}]$ by above

$\Omega \vdash (\overline{e_1}; e_i'; \overline{e_n}) \mapsto f \in \forall(\overline{\alpha_1 \in \delta_1}, \alpha_i \in \delta_i, \overline{\alpha_n \in \delta_n}).\ \tau$ by cMatch

Case: $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \vdash c \to c'\end{array}}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash c \backslash \boldsymbol{x} \to c' \backslash \boldsymbol{x}}\ \mathsf{redCasePop}$

See case for redPop (above)

Case: $\mathcal{E} = \dfrac{}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu \boldsymbol{x'} \in \boldsymbol{A}^{\#}.\ c) \backslash \boldsymbol{x} \to c[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{redCasePopElim}$

See case for redPopElim (above)

. . . . . .

Case: $\mathcal{E} = \dfrac{}{\Omega \vdash c \xrightarrow{*} c}\,\mathsf{closeBase}$

$\Omega \vdash c \in \tau$	by assumption

Case: $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash c_1 \to c_2 & \Omega \vdash c_2 \xrightarrow{*} c_3\end{array}}{\Omega \vdash c_1 \xrightarrow{*} c_3}\,\mathsf{closeInd}$

$\Omega \vdash c_1 \in \tau$	by assumption
$\Omega \vdash c_2 \in \tau$	by i.h. on $\mathcal{E}_1$
$\Omega \vdash c_3 \in \tau$	by i.h. on $\mathcal{E}_2$

□

B.20 Meta-Theory: Liveness

Lemma B.20.1 (Generalized redEps)**.**
If $\Omega \vdash \epsilon\Omega'.\ c \in \tau$ *and* $\Omega \vdash (\mathrm{id}_\Omega, \omega) : \Omega, \Omega'$*, then* $\Omega \vdash (\epsilon\Omega'.\ c) \xrightarrow{*} c[\mathrm{id}_\Omega, \omega]$.

Proof. By induction on Ω'.

Case: $\cdot$

$\Omega \vdash c \in \tau$	by assumption
$c = c[\mathrm{id}_\Omega]$	by Lemma B.5.3
$\Omega \vdash c \xrightarrow{*} c[\mathrm{id}_\Omega]$	by closeBase

Case: $\Omega', \alpha{\in}\delta$

$\Omega \vdash \epsilon\Omega'.\ (\epsilon\alpha{\in}\delta.\ c) \in \tau$	by assumption and Def.
$\Omega \vdash (\mathrm{id}_\Omega, \omega, e/\alpha) : \Omega, \Omega', \alpha{\in}\delta$	by assumption
$\Omega \vdash (\mathrm{id}_\Omega, \omega) : \Omega, \Omega'$	by inversion using tpSubInd
$\Omega \vdash e \in \delta[\mathrm{id}_\Omega, \omega]$	by inversion using tpSubInd
$\Omega \vdash \delta[\mathrm{id}_\Omega, \omega]$ wff	by Lemma B.8.1
Ω ctx	by Lemma B.3.2
$\Omega \vdash \mathrm{id}_\Omega : \Omega$	by Lemma B.5.5
$\Omega, \alpha{\in}\delta[\mathrm{id}_\Omega, \omega] \vdash \uparrow_\alpha (\mathrm{id}_\Omega, \omega), \alpha/\alpha : \Omega, \Omega', \alpha{\in}\delta$	by tpSubShift and tpSubInd and Lemma B.5.4
$\Omega \vdash \mathrm{id}_\Omega, e/\alpha : \Omega, \alpha{\in}\delta[\mathrm{id}_\Omega, \omega]$	by tpSubInd and Lemma B.5.2
$\Omega \vdash (\epsilon\Omega'.\ (\epsilon\alpha{\in}\delta.\ c))$ $\xrightarrow{*} \epsilon\alpha{\in}\delta[\mathrm{id}_\Omega, \omega].\ c[\uparrow_\alpha (\mathrm{id}_\Omega, \omega), \alpha/\alpha]$	by i.h. on Ω' and Subst. App. (Def. A.2.4)
$\xrightarrow{*} c[\uparrow_\alpha (\mathrm{id}_\Omega, \omega), \alpha/\alpha][\mathrm{id}_\Omega, e/\alpha]$	by redEps with e

$c[\uparrow_\alpha (\mathrm{id}_\Omega, \omega), \alpha/\alpha][\mathrm{id}_\Omega, e/\alpha]$	
$= c[(\uparrow_\alpha (\mathrm{id}_\Omega, \omega), \alpha/\alpha) \circ (\mathrm{id}_\Omega, e/\alpha)]$	by Lemma B.18.6
$= c[((\uparrow_\alpha (\mathrm{id}_\Omega, \omega)) \circ (\mathrm{id}_\Omega, e/\alpha)), e/\alpha]$	by Composition and Subst. App.
$= c[((\mathrm{id}_\Omega, \omega) \circ \mathrm{id}_\Omega), e/\alpha]$	by Composition
$= c[\mathrm{id}_\Omega, \omega, e/\alpha]$	by Lemma B.16.5

$\Omega \vdash \epsilon\Omega', \alpha{\in}\delta.\ c \xrightarrow{*} c[\mathrm{id}_\Omega, \omega, e/\alpha]$	by above

□

Lemma B.20.2 (Substitution Composition Prefix Generality)**.**
If $\omega \neq \uparrow_{\alpha''} \omega''$, then $(\omega_1, \omega_2) \circ \omega = (\omega_1 \circ \omega), \_$.
We write $\_$ to just refer to any ω', but we don't care about its exact form.

Proof. By induction on ω_2.

Case: $\omega_2 = \cdot$

$(\omega_1, \cdot) \circ \omega = \omega_1 \circ \omega$ trivial

Case: $\omega_2 = \omega_2', e/\alpha$

$(\omega_1, \omega_2', e/\alpha) \circ \omega$
$= (\omega_1, \omega_2') \circ \omega, \_$ by Def. of subst. composition
$= (\omega_1 \circ \omega), \_$ by i.h. on ω_2'

□

Lemma B.20.3 (Most-General-Unifier Property)**.**
If there exists a well-typed ϱ such that
$$((((\uparrow_\Omega \cdot), \sigma_A) + \mathbf{\Gamma}_{\boldsymbol{x}}), \sigma_c) = ((((\uparrow_{\Omega'} \cdot), \omega_A) + \mathbf{\Gamma}_{\boldsymbol{x}}), \omega_c) \circ \varrho,$$
then there exists a well-typed ϱ of the form $((\uparrow_\Omega \cdot), \varrho') + \mathbf{\Gamma}_{\boldsymbol{x}}$.

Proof. By induction on $\mathbf{\Gamma}_{\boldsymbol{x}}$.

Case: $\mathbf{\Gamma}_{\boldsymbol{x}} = \cdot$ and $\varrho \neq \uparrow_{\alpha''} \_$

Let $\sigma = \sigma_A, \sigma_c$	
Let $\omega = \omega_A, \omega_c$	
$((\uparrow_\Omega \cdot), \sigma) = ((\uparrow_{\Omega'} \cdot), \omega) \circ \varrho$	by assumption
$\varrho \neq \uparrow_{\alpha''} \_$	by assumption
$((\uparrow_\Omega \cdot), \sigma)$	
$= ((\uparrow_{\Omega'} \cdot), \omega) \circ \varrho$	by assumption
$= ((\uparrow_{\Omega'} \cdot) \circ \varrho), \_$	by Lemma B.20.2
$= (\cdot \circ \varrho'), \_$	by Substitution Composition
	noting that $\varrho = \varrho', \varrho''$
$(\cdot \circ \varrho') = (\uparrow_\Omega \cdot)$	by above analyzing substitution rules
$\varrho' = (\uparrow_{\Omega'} \cdot)$	by inversion using substitution rules
and hence $\varrho = ((\uparrow_{\Omega'} \cdot), \varrho'')$	

Case: $\boldsymbol{\Gamma_x} = \cdot$ and $\varrho = \uparrow_{\alpha''} \varrho'$ and $\Omega = \Omega_2, \alpha{\in}\delta$

Let $\sigma = \sigma_A, \sigma_c$
Let $\omega = \omega_A, \omega_c$
$((\uparrow_\Omega \cdot), \sigma) = ((\uparrow_{\Omega'} \cdot), \omega) \circ \varrho$ by assumption
$\varrho = \uparrow_{\alpha''} \varrho'$ by assumption
$\sigma = \cdot$ by above and substitution composition
$\Omega = \Omega_2, \alpha{\in}\delta$ by assumption
$\uparrow_\alpha (\uparrow_{\Omega_2} \cdot)$
$= ((\uparrow_{\Omega'} \cdot), \omega) \circ (\uparrow_{\alpha''} \varrho')$ by above
$= \uparrow_{\alpha''} (((\uparrow_{\Omega'} \cdot), \omega) \circ \varrho')$ by Subst. Composition
$\alpha'' = \alpha$ by above
$(\uparrow_{\Omega_2} \cdot) = (((\uparrow_{\Omega'} \cdot), \omega) \circ \varrho')$ by above
$\omega = \cdot$ by substitution composition
$(\uparrow_{\Omega_2} \cdot) = ((\uparrow_{\Omega'} \cdot) \circ \varrho')$ by above
$\varrho' = (\uparrow_{\Omega_2} \varrho'')$ by inversion using substitution rules
$\cdot = ((\uparrow_{\Omega'} \cdot) \circ \varrho'')$ by above
$\varrho'' = \cdot, \varrho'''$ by inversion using substitution rules
and hence $\varrho = \uparrow_\Omega (\cdot, \varrho''')$
However, $\varrho^* = (\uparrow_\Omega \cdot), \varrho'''$ is also well-formed by weakening
and $(\uparrow_{\Omega'} \cdot) \circ ((\uparrow_\Omega \cdot), \varrho''')$
$= \uparrow_\Omega \cdot$ by above form of ϱ'''
Therefore, there exists a ϱ^* of the appropriate form.

Case: $\boldsymbol{\Gamma_x = \Gamma'_x, y{:}B}$

$(\uparrow_{\boldsymbol{y}} (((\uparrow_\Omega \cdot), \sigma_A) + \boldsymbol{\Gamma'_x}), \boldsymbol{y}/\boldsymbol{y}, \sigma_c)$
$= (\uparrow_{\boldsymbol{y}} ((\uparrow_{\Omega'} (\cdot, \omega_A)) + \boldsymbol{\Gamma'_x}), \boldsymbol{y}/\boldsymbol{y}, \omega_c) \circ \varrho$ by assumption
$= (\uparrow_{\boldsymbol{y}} ((\uparrow_{\Omega'} (\cdot, \omega_A)) + \boldsymbol{\Gamma'_x}), \boldsymbol{y}/\boldsymbol{y} \circ \varrho), \_$ by Lemma B.20.2
$= (\uparrow_{\boldsymbol{y}} ((\uparrow_{\Omega'} (\cdot, \omega_A)) + \boldsymbol{\Gamma'_x}) \circ \varrho), \boldsymbol{y}[\varrho]/\boldsymbol{y}, \_$ by Subst. Composition
$= (((\uparrow_{\Omega'} (\cdot, \omega_A)) + \boldsymbol{\Gamma'_x}) \circ \varrho'), \boldsymbol{y}/\boldsymbol{y}, \_$ noting $\varrho = \varrho', \boldsymbol{y}/\boldsymbol{y}$
$\uparrow_{\boldsymbol{y}} (((\uparrow_\Omega \cdot), \sigma_A) + \boldsymbol{\Gamma'_x})$
$= (((\uparrow_{\Omega'} (\cdot, \omega_A)) + \boldsymbol{\Gamma'_x}) \circ \varrho')$ by above
$= \uparrow_{\boldsymbol{y}} (((\uparrow_{\Omega'} (\cdot, \omega_A)) + \boldsymbol{\Gamma'_x}) \circ \varrho'')$ noting that $\varrho' = \uparrow_{\boldsymbol{y}} \varrho''$
by the well-typedness of ϱ and that $\boldsymbol{y}$ must be mapped to $\boldsymbol{y}$
$\varrho'' = ((\uparrow_\Omega \cdot), \varrho''') + \boldsymbol{\Gamma'_x}$ by i.h. on Γ'_x

$\varrho = \uparrow_{\boldsymbol{y}} (((\uparrow_\Omega \cdot), \varrho''') + \boldsymbol{\Gamma'_x}), \boldsymbol{y}/\boldsymbol{y}$ by above
$= ((\uparrow_\Omega \cdot), \varrho''') + \boldsymbol{\Gamma_x}$ by Definition of Substitution

□

Lemma B.20.4 (Liveness Property)**.**
If

- Ω *does not contain any declarations of the form* $\alpha{\in}\delta$
- *and* $\Omega \vdash \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ \forall\overline{\alpha{\in}\delta}.\ \tau$
- *and* $\Omega, \mathbf{\Gamma} \vdash \mathrm{id}_{(\Omega,\mathbf{\Gamma})}, \overline{v}/\overline{\alpha} : \Omega, \mathbf{\Gamma}, \overline{\alpha{\in}\delta}$,

then there exists a c_i *in* $\overline{c}$ *such that* $\Omega \vdash c_i \xrightarrow{*} \nu\mathbf{\Gamma}.\ (\overline{v} \mapsto f)$.

Proof. By induction on $\mathcal{E} :: \Omega \vdash \overline{c}$ covers $\nabla\mathbf{\Gamma}.\ \forall\overline{\alpha{\in}\delta}.\ \tau$. This lemma requires heavy uses of Lemma B.6.1, Lemma B.7.4, Lemma B.1.16, Subst. App. (Def. A.2.4) and properties of LF. We omit some details for the sake of brevity.

Case: $\mathcal{E} = \dfrac{c = \epsilon\alpha{\in}\delta.\ \alpha \mapsto f}{\Omega \vdash c \text{ covers } \forall\alpha{\in}\delta.\ \tau}\ \mathsf{coverSimple}$

$\Omega \vdash \mathrm{id}_\Omega, v/\alpha : \Omega, \alpha{\in}\delta$	by assumption
$\Omega \vdash v \in \delta$	by inversion using tpSubInd
$\Omega \vdash (\epsilon\alpha{\in}\delta.\ \alpha \mapsto f) \to (v \mapsto f[\mathrm{id}_\Omega, v/\alpha])$	by redEps and Subst. App.

Case:

$$\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau\ \mathsf{wff} \\ c = \epsilon\boldsymbol{y}{\in}(\boldsymbol{\Pi\Gamma.\ A}).\ \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y\ \Gamma})/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{y\ \Gamma},\ u\backslash\boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c\ \text{covers}\ (\nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}$$

$\Omega \vdash \mathrm{id}_\Omega, (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{M},\ v))/u : \Omega, u{\in}(\nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau)$
by assumption and inversion

$\Omega \vdash \nu\boldsymbol{\Gamma}.\ (\boldsymbol{M},\ v) \in \nabla\boldsymbol{\Gamma}.\ \exists\boldsymbol{x}{\in}\boldsymbol{A}.\ \tau$ by inversion using tpSubInd

$\Omega, \boldsymbol{\Gamma} \vdash \boldsymbol{M} \in \boldsymbol{A}$ by inversion

$\Omega, \boldsymbol{\Gamma} \vdash v \in \tau[\mathrm{id}_{(\Omega,\boldsymbol{\Gamma})}, \boldsymbol{M}/\boldsymbol{x}]$ by inversion

$\Omega \vdash \boldsymbol{\lambda\Gamma.\ M} \in \boldsymbol{\Pi\Gamma.\ A}$ by **LF_Lam**

$\Omega \vdash \nu\boldsymbol{\Gamma}.\ v \in \nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{(\Omega,\boldsymbol{\Gamma})}, \boldsymbol{M}/\boldsymbol{x}]$ by new

$\tau[\mathrm{id}_{\boldsymbol{\Gamma}}, (\boldsymbol{y\ \Gamma})/\boldsymbol{x}][(\mathrm{id}_\Omega, (\boldsymbol{\lambda\Gamma.\ M})/\boldsymbol{y}) + \boldsymbol{\Gamma}]$
$=\tau[\mathrm{id}_{(\Omega,\boldsymbol{\Gamma})}, \boldsymbol{M}/\boldsymbol{x}]$ Since $(\boldsymbol{\lambda\Gamma.\ M})\ \boldsymbol{\Gamma} = \boldsymbol{M}$

$\Omega \vdash c$
$\rightarrow \epsilon u{\in}(\nabla\boldsymbol{\Gamma}.\ \tau[\mathrm{id}_{(\Omega,\boldsymbol{\Gamma})}, \boldsymbol{M}/\boldsymbol{x}]).\ (\nu\boldsymbol{\Gamma}.\ (\boldsymbol{M},\ u\backslash\boldsymbol{\Gamma})) \mapsto f'$
by redEps with $\boldsymbol{\lambda\Gamma.\ M}$
Note that $f' = f[\uparrow_u(\mathrm{id}_\Omega, (\boldsymbol{\lambda\Gamma.\ M})/\boldsymbol{y}), u/u]$

$\xrightarrow{*} \nu\boldsymbol{\Gamma}.\ (\boldsymbol{M},\ v) \mapsto f''$
by redEps with $\nu\boldsymbol{\Gamma}.\ v$
and redCasePattern noting that $(\nu\boldsymbol{\Gamma}.\ v)\backslash\boldsymbol{\Gamma} \xrightarrow{*} v$
Note that $f'' = f[\mathrm{id}_\Omega, (\boldsymbol{\lambda\Gamma.\ M})/\boldsymbol{y}, (\nu\boldsymbol{\Gamma}.\ v)/u]$

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau\ \mathsf{wff} \\ c = \epsilon \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \epsilon u{\in}\tau.\ (\boldsymbol{x},\ u) \mapsto f\end{array}}{\Omega \vdash c\ \text{covers}\ (\exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau) \supset \sigma}\ \mathsf{coverPairLF}^{\#}$

$\Omega \vdash \mathrm{id}_\Omega, (\boldsymbol{x},\ v)/u : \Omega, u{\in}(\exists \boldsymbol{x}{\in}\boldsymbol{A}.\ \tau)$ by assumption and inversion
$\Omega \vdash (\boldsymbol{x},\ v) \in \exists \boldsymbol{x}{\in}\boldsymbol{A}^{\#}.\ \tau$ by inversion using tpSubInd
$\Omega \vdash \boldsymbol{x} \in \boldsymbol{A}^{\#}$ by inversion
$\Omega \vdash v \in \tau[\mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x}]$ by inversion
$\Omega \vdash c$
$\quad \to \epsilon u{\in}\tau[\mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x}].\ (\boldsymbol{x},\ u) \mapsto f'$ by redEps with $\boldsymbol{M}$
Note that $f' = f[\uparrow_u (\mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x}), u/u]$
$\quad \to (\boldsymbol{x},\ v) \mapsto f''$ by redEps with v
Note that $f'' = f[\mathrm{id}_\Omega, \boldsymbol{x}/\boldsymbol{x}, v/u]$

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla \boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)\ \mathsf{wff} \\ c = \epsilon u_1{\in}(\nabla \boldsymbol{\Gamma}.\ \tau_1).\ \epsilon u_2{\in}(\nabla \boldsymbol{\Gamma}.\ \tau_2).\ (\nu \boldsymbol{\Gamma}.\ (u_1 \backslash \boldsymbol{\Gamma},\ u_2 \backslash \boldsymbol{\Gamma})) \mapsto f\end{array}}{\Omega \vdash c\ \text{covers}\ (\nabla \boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)) \supset \sigma}\ \mathsf{coverPairMeta}$

$\Omega \vdash \mathrm{id}_\Omega, (\nu \boldsymbol{\Gamma}.\ (v_1,\ v_2))/u : \Omega, u{\in}(\nabla \boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2))$
by assumption and inversion
$\Omega \vdash \nu \boldsymbol{\Gamma}.\ (v_1,\ v_2) \in \nabla \boldsymbol{\Gamma}.\ (\tau_1 \star \tau_2)$ by inversion using tpSubInd
$\Omega, \boldsymbol{\Gamma} \vdash v_1 \in \tau_1$ by inversion
$\Omega, \boldsymbol{\Gamma} \vdash v_2 \in \tau_2$ by inversion
$\Omega \vdash \nu \boldsymbol{\Gamma}.\ v_1 \in \nabla \boldsymbol{\Gamma}.\ \tau_1$ by new
$\Omega \vdash \nu \boldsymbol{\Gamma}.\ v_2 \in \nabla \boldsymbol{\Gamma}.\ \tau_2$ by new
$\Omega \vdash c$
$\quad \xrightarrow{*} \epsilon u_2{\in}(\nabla \boldsymbol{\Gamma}.\ \tau_2).\ (\nu \boldsymbol{\Gamma}.\ (v_1,\ u_2 \backslash \boldsymbol{\Gamma})) \mapsto f'$
by redEps with $\nu \boldsymbol{\Gamma}.\ v_1$
and redCasePattern noting that $(\nu \boldsymbol{\Gamma}.\ v_1) \backslash \boldsymbol{\Gamma} \xrightarrow{*} v_1$
Note that $f' = f[\uparrow_{u_2} (\mathrm{id}_\Omega, (\nu \boldsymbol{\Gamma}.\ v_1)/u_1), u_2/u_2]$
$\quad \xrightarrow{*} \nu \boldsymbol{\Gamma}.\ (v_1,\ v_2) \mapsto f''$
by redEps with $\nu \boldsymbol{\Gamma}.\ v_2$
and redCasePattern noting that $(\nu \boldsymbol{\Gamma}.\ v_2) \backslash \boldsymbol{\Gamma} \xrightarrow{*} v_2$
Note that $f'' = f[\mathrm{id}_\Omega, (\nu \boldsymbol{\Gamma}.\ v_1)/u_1, (\nu \boldsymbol{\Gamma}.\ v_2)/u_2]$

Case:

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega_1 \vdash \overline{c} \text{ covers } \nabla \boldsymbol{x'} {\in} \boldsymbol{A}^{\#}.\ (\nabla \boldsymbol{\Gamma}.\ \forall \overline{\alpha {\in} \delta}.\ \tau) \qquad (\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash \overline{c} \backslash \boldsymbol{x} \text{ covers } (\nabla \boldsymbol{\Gamma}.\ \forall \overline{\alpha {\in} \delta}.\ \tau)[\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{coverPop}$$

Ω_1 does not contain any decs. of $\alpha^* {\in} \delta^*$ by assumption

Ω_2 does not contain any decs. of $\alpha^* {\in} \delta^*$ by assumption

Let $[\boldsymbol{x}/\boldsymbol{x'}]$ be shorthand for $\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}$ for simplicity

Let $[\boldsymbol{x}/\boldsymbol{x'}]_+$ be shorthand for $(\uparrow_{\boldsymbol{x}} \mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}) + \boldsymbol{\Gamma}$ for simplicity

Let $[\boldsymbol{x'}/\boldsymbol{x}]_+$ be shorthand for $(\uparrow_{\boldsymbol{x'}} \mathrm{id}_{\Omega_1}, \boldsymbol{x'}/\boldsymbol{x}) + \boldsymbol{\Gamma}$ for simplicity

$(\nabla \boldsymbol{\Gamma}.\ \forall \overline{\alpha {\in} \delta}.\ \tau)[\boldsymbol{x}/\boldsymbol{x'}] = \nabla \boldsymbol{\Gamma}[\boldsymbol{x}/\boldsymbol{x'}].\ \forall \overline{\alpha {\in} \delta}.\ [\boldsymbol{x}/\boldsymbol{x'}]_+$

by Substitution Application

$(\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}) \leq \Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2$ by assumption

$\Omega_2 = \cdot$ by above and Definition of $\leq$

The fact that Ω_2 is forced to be empty, is key to enforcing that coverage holds.

$\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \boldsymbol{\Gamma}[\boldsymbol{x}/\boldsymbol{x'}] \vdash \mathrm{id}, \overline{v}/\overline{\alpha} : \Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \boldsymbol{\Gamma}[\boldsymbol{x}/\boldsymbol{x'}], \overline{\alpha {\in} \delta}[\boldsymbol{x}/\boldsymbol{x'}]_+$

by assumption and Substitution Application

$\Omega_1, \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \boldsymbol{\Gamma} \vdash \mathrm{id}, (\overline{v}[\boldsymbol{x'}/\boldsymbol{x}]_+)/\overline{\alpha} : \Omega_1, \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \boldsymbol{\Gamma}, \overline{\alpha {\in} \delta}$ by Substitution

There exists a c_i in $\overline{c}$ such that $\Omega_1 \vdash c_i \xrightarrow{*} \nu \boldsymbol{x'} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}.\ \nu \boldsymbol{\Gamma}.\ (\overline{v}[\boldsymbol{x'}/\boldsymbol{x}]_+ \mapsto f)$

by i.h. on $\mathcal{E}_1$

$\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#} \vdash c_i \backslash \boldsymbol{x} \xrightarrow{*} (\nu \boldsymbol{\Gamma}.\ (\overline{v}[\boldsymbol{x'}/\boldsymbol{x}]_+ \mapsto f))[\boldsymbol{x}/\boldsymbol{x'}]$

by $\mathsf{redCasePop}$(0 or more) and $\mathsf{redCasePopElim}$

$\Omega_1, \boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}, \Omega_2 \vdash c_i \backslash \boldsymbol{x} \xrightarrow{*} \nu \boldsymbol{\Gamma}[\boldsymbol{x}/\boldsymbol{x'}].\ (\overline{v} \mapsto f[\boldsymbol{x}/\boldsymbol{x'}]_+)$

by above and Substitution

$$\textbf{Case: } \mathcal{E} = \frac{\begin{array}{l}(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x})\ \mathsf{ctx}\\ \mathcal{F}_1 :: \boldsymbol{\Sigma} \gg \overline{c}\ \text{covers}\ \forall(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau\\ \mathcal{F}_2 :: \boldsymbol{\Gamma_x} \gg \overline{c}\ \text{covers}\ \forall(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau\\ \mathcal{F}_3 :: \Omega \gg^{\text{world}} \overline{c}\ \text{covers}\ \forall(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau\end{array}}{\Omega \vdash \overline{c}\ \text{covers}\ \forall(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau}\ \mathsf{coverLF}$$

$\Omega \vdash \mathrm{id}_\Omega, \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x} : \Omega, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}$ by assumption
$(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x})\ \mathsf{ctx}$ by assumption
$\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot, \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x}) : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in(\boldsymbol{\Pi\Gamma_x.\ B_x})$ by Lemma B.8.3
By the canonical form property of LF, we can do case analysis on $\boldsymbol{M}$ and get three cases:

Signature Case: $\boldsymbol{c}{:}(\boldsymbol{\Pi\Gamma_c.\ B_c})$ in $\boldsymbol{\Sigma}$
and $\boldsymbol{M} = \boldsymbol{\lambda\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{c\ M_1\ \ldots\ M_m}$

$\boldsymbol{\Gamma_c} = \boldsymbol{x_1}{:}\boldsymbol{A_1}, \ldots, \boldsymbol{x_m}{:}\boldsymbol{A_m}$ by assumption
$\Omega \vdash \mathrm{id}_\Omega, \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x} : \Omega, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in(\boldsymbol{\Pi\Gamma_x.\ B_x})$ by assumption
$\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x} : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in(\boldsymbol{\Pi\Gamma_x.\ B_x})$ by assumption
$\|\Omega\|^\nabla \vdash \boldsymbol{M} = \boldsymbol{\lambda\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{c\ M_1\ \ldots\ M_m}$
$\in (\boldsymbol{\Pi\Gamma_x.\ B_x})[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}]$ by assumption
Let $\sigma = [(((\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}) + \boldsymbol{\Gamma_x}), \boldsymbol{M_1}/\boldsymbol{x_1}, \ldots, \boldsymbol{M_m}/\boldsymbol{x_m}]$
$\|\Omega\|^\nabla, \boldsymbol{\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}] \vdash \sigma : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_c}$ by Substitution
$\|\Omega\|^\nabla, \boldsymbol{\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}] \vdash \boldsymbol{B_c}[\sigma] = \boldsymbol{B_x}[\sigma]$
by Typing Rules with Weakening

By inversion on $\mathcal{F}_1$ either scUnify or scSkip must have applied.
However, scSkip could not have applied by above
$\Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \text{mgu}\ (\boldsymbol{B_x} \approx \boldsymbol{B_c}) : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_c}$
and $\omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_c$ by scUnify
There exists a g in $\overline{c}$ such that
$g = \epsilon\Omega'.\ \overline{\boldsymbol{x'}}[\omega]; (\boldsymbol{\lambda\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A].\ (\boldsymbol{c\ x_1\ \ldots\ x_m})[\omega]) \mapsto f$ by scUnify
There exists a σ' such that $\sigma = \omega \circ \sigma'$ by mgu (Def. A.2.7)
and $\|\Omega\|^\nabla, \boldsymbol{\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}] \vdash \sigma' : \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \mathrm{id}_\Omega), \omega_A]$
and $\sigma' = ((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma_x}$ By Lemma B.20.3

We now show that σ'' gives us the assignments of $\epsilon\Omega'$
so that the case g matches.

The key idea is that we show it is equivalent to applying the object to σ.

Keep in mind that $(\uparrow_{\|\Omega\|^\nabla} \cdot) \leq \mathrm{id}_\Omega$.

Main Case:
$(\boldsymbol{\lambda \Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A]\boldsymbol{.\ (c\ x_1\ \ldots\ x_m)}[\omega])[\mathrm{id}_\Omega, \sigma'']$
$= (\boldsymbol{\lambda \Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A]\boldsymbol{.\ (c\ x_1\ \ldots\ x_m)}[\omega])[(\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'']$
$= \boldsymbol{\lambda}(\boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A][(\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'])\boldsymbol{.\ (c\ x_1\ \ldots\ x_m)}[\omega \circ ((\mathrm{id}_\Omega, \sigma'') + \boldsymbol{\Gamma_x})]$
$= \boldsymbol{\lambda}(\boldsymbol{\Gamma_x}[\omega \circ (((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma_x})])\boldsymbol{.\ (c\ x_1\ \ldots\ x_m)}[\omega \circ ((\mathrm{id}_\Omega, \sigma'') + \boldsymbol{\Gamma_x})]$
$= \boldsymbol{\lambda}(\boldsymbol{\Gamma_x}[\omega \circ \sigma'])\boldsymbol{.\ (c\ x_1\ \ldots\ x_m)}[\omega \circ \sigma']$
$= \boldsymbol{\lambda}(\boldsymbol{\Gamma_x}[\sigma])\boldsymbol{.\ (c\ x_1\ \ldots\ x_m)}[\sigma]$
$= \boldsymbol{\lambda \Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}]\boldsymbol{.\ c\ M_1\ \ldots\ M_m}$
$= \boldsymbol{M}$

Case for $\boldsymbol{x'_i} \in \overline{\boldsymbol{x'} \in \delta}$
$\boldsymbol{x'_i}[\omega][\mathrm{id}_\Omega, \sigma'']$
$= \boldsymbol{x'_i}[\omega][(\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'']$
$= \boldsymbol{x'_i}[\omega][((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma_x}]$
$= \boldsymbol{x'_i}[\omega \circ (((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma_x})]$
$= \boldsymbol{x'_i}[\sigma]$
$= v_i$ by Definition of σ

$\Omega \vdash g \xrightarrow{*} \overline{v}; \boldsymbol{M} \mapsto f[\mathrm{id}_\Omega, \sigma'']$ by redEps with σ'' (Lemma B.20.1)

Local Parameter Case: $\boldsymbol{c}{:}(\boldsymbol{\Pi\Gamma_c}.\ \boldsymbol{B_c})$ in $\boldsymbol{\Gamma_x}$
and $\boldsymbol{M} = \boldsymbol{\lambda\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{c}\ \boldsymbol{M_1}\ \ldots\ \boldsymbol{M_m}$

This follows the same reasoning as the previous case
using lcUnify, lcSkip, and $\mathcal{F}_2$ in lieu of scUnify, scSkip, and $\mathcal{F}_1$.

Global Parameter Case: $\boldsymbol{c}\overset{\nabla}{\in}(\boldsymbol{\Pi\Gamma_c}.\ \boldsymbol{B_c})$ in Ω
and $\boldsymbol{M} = \boldsymbol{\lambda\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{c}\ \boldsymbol{M_1}\ \ldots\ \boldsymbol{M_m}$

$\boldsymbol{\Gamma_c} = \boldsymbol{x_1}{:}\boldsymbol{A_1}, \ldots, \boldsymbol{x_m}{:}\boldsymbol{A_m}$ by assumption
$\Omega \vdash \mathrm{id}_\Omega, \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x} : \Omega, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in(\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})$ by assumption
$\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x} : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in(\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})$ by assumption
$\|\Omega\|^\nabla \vdash \boldsymbol{M} = \boldsymbol{\lambda\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{c}\ \boldsymbol{M_1}\ \ldots\ \boldsymbol{M_m}$
$\in (\boldsymbol{\Pi\Gamma_x}.\ \boldsymbol{B_x})[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}]$ by assumption
Let $\sigma_0 = [(((\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}) + \boldsymbol{\Gamma_x}), \boldsymbol{M_1}/\boldsymbol{x_1}, \ldots, \boldsymbol{M_m}/\boldsymbol{x_m}]$
$\|\Omega\|^\nabla, \boldsymbol{\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}] \vdash \sigma_0 : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_c}$ by Substitution
$\|\Omega\|^\nabla, \boldsymbol{\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}] \vdash \boldsymbol{B_c}[\sigma] = \boldsymbol{B_x}[\sigma]$
by Typing Rules with Weakening

By inversion on $\mathcal{F}_3$ either gcSkipMismatch or gcSkipWorld
must have applied.

Subcase: gcSkipMismatch applied

Impossible case as $\boldsymbol{B_x} \approx \boldsymbol{B_c}$ is possible as evidenced by above

Subcase: gcSkipWorld applied

$\boldsymbol{\Gamma_c} = \boldsymbol{x_1}{:}\boldsymbol{A_1}, \ldots, \boldsymbol{x_m}{:}\boldsymbol{A_m}$ Repeated from above

$\Omega \vdash \mathrm{id}_\Omega, \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x} : \Omega, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in(\boldsymbol{\Pi\Gamma_x.\ B_x})$ Repeated

$\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{M}/\boldsymbol{x} : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in(\boldsymbol{\Pi\Gamma_x.\ B_x})$ Repeated

$\|\Omega\|^\nabla \vdash \boldsymbol{M} = \boldsymbol{\lambda\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{c\ M_1\ \ldots\ M_m}$
$\in (\boldsymbol{\Pi\Gamma_x.\ B_x})[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}]$ Repeated

$\mathcal{W}$ world by inversion on gcSkipWorld

$\mathcal{W} \gg \overline{c}$ covers $\forall(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{\Pi\Gamma_x.\ B_x}).\ \tau$ by gcSkipWorld

$(\Omega_g, \boldsymbol{\Pi\Gamma_c.\ B_c}) \in \mathcal{W}$ and $\Omega_g \leq_* \Omega$ by gcSkipWorld

$(\Omega_g, \boldsymbol{\Pi\Gamma_c.\ B_c}) \in_{\mathrm{alt}} \mathcal{W}$ by Lemma B.9.2

There exists an $(\Omega_b, (\boldsymbol{\Pi\Gamma_b.\ B_b}))$ in $\mathcal{W}$ such that
Ω_b only contains decs. of type $\boldsymbol{A}^*$ or $\boldsymbol{A}^{*\#}$ and
$\|\Omega_g\|^\nabla \vdash \omega_0' : \Omega_b$ and
$(\boldsymbol{\Pi\Gamma_b.\ B_b})[\omega_0'] = \boldsymbol{\Pi\Gamma_c.\ B_c}$ by inversion and includesAltYes
Note that includesAltAny could not apply as then $\mathcal{W}$ would not cover.

There exists an ω_0 such that
$\|\Omega\|^\nabla \vdash (\uparrow_{\|\Omega\|^\nabla} \cdot), \omega_0 : \Omega_b$ and
$(\boldsymbol{\Pi\Gamma_b.\ B_b})[(\uparrow_{\|\Omega\|^\nabla} \cdot), \omega_0] = \boldsymbol{\Pi\Gamma_c.\ B_c}$
by Lemma B.8.5 and Weakening $(\Omega_g \leq_* \Omega)$

Let $\sigma = [(((\uparrow_{\|\Omega\|^\nabla} \cdot), \omega_0, \overline{v}/\overline{\boldsymbol{x'}}) + \boldsymbol{\Gamma_x}), \boldsymbol{M_1}/\boldsymbol{x_1}, \ldots, \boldsymbol{M_m}/\boldsymbol{x_m}]$
$\|\Omega\|^\nabla, \boldsymbol{\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla} \cdot), \overline{v}/\overline{\boldsymbol{x'}}] \vdash \sigma : \Omega_b, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_c}$
by Substitution and Weakening

Looking at $(\Omega_b, (\boldsymbol{\Pi\Gamma_b.\ B_b}))$, by inversion on $\mathcal{F}_3$,
either wcUnify or wcSkip must have applied.

$\boldsymbol{B_b} \approx \boldsymbol{B_c}$ by ω_0

$\boldsymbol{B_x} \approx \boldsymbol{B_c}$ by above

Therefore, $\boldsymbol{B_b} \approx \boldsymbol{B_x}$ and wcSkip could not have applied.

$\Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \cdot), \omega_A] \vdash \omega = \mathrm{mgu}\ (\boldsymbol{B_x} \approx \boldsymbol{B_b}) : \Omega_b, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{\Gamma_x}, \boldsymbol{\Gamma_b}$
and $\omega = ((\uparrow_{\Omega'} \cdot), \omega_A + \boldsymbol{\Gamma_x}), \omega_b$ by wcUnify

There exists a g in $\overline{c}$ such that
$g = \epsilon\Omega'.\ \epsilon\boldsymbol{y_b}\in(\boldsymbol{\Pi\Gamma_b.\ B_b})[\omega]^\#.$
$\overline{\boldsymbol{x'}}[\omega]; (\boldsymbol{\lambda\Gamma_x}[(\uparrow_{\Omega'} \mathrm{id}_\Omega), \omega_A].\ (\boldsymbol{y_b\ x_1\ \ldots\ x_m})[\omega]) \mapsto f$ by wcUnify

There exists a σ' such that $\sigma = \omega \circ \sigma'$ by mgu (Def. A.2.7)
and $\|\Omega\|^\nabla, \boldsymbol{\Gamma_x}[\cdot, \overline{v}/\overline{\boldsymbol{x'}}] \vdash \sigma' : \Omega', \boldsymbol{\Gamma_x}[(\uparrow_{\Omega'} \mathrm{id}_\Omega), \omega_A]$
and $\sigma' = ((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma_x}$ by Lemma B.20.3

We now show that σ'' gives us the assignments of $\epsilon\Omega'$ so that the case g matches. (Similar to the signature case above)

Main Case:

$(\boldsymbol{\lambda\Gamma_x}[(\uparrow_{\Omega'}\cdot), \omega_A].\ (\boldsymbol{y_b\ x_1\ \ldots\ x_m})[\omega])[\mathrm{id}_\Omega, \sigma'']$

$\quad (\boldsymbol{\lambda\Gamma_x}[(\uparrow_{\Omega'}\cdot), \omega_A].\ (\boldsymbol{y_b\ x_1\ \ldots\ x_m})[\omega])[(\uparrow_{\|\Omega\|^\nabla}\cdot), \sigma'']$

$\quad = \boldsymbol{\lambda}(\boldsymbol{\Gamma_x}[(\uparrow_{\Omega'}\cdot), \omega_A][(\uparrow_{\|\Omega\|^\nabla}\cdot), \sigma'']).$

$\qquad\qquad (\boldsymbol{y_b\ x_1\ \ldots\ x_m})[\omega \circ (((\uparrow_{\|\Omega\|^\nabla}\cdot), \sigma'') + \boldsymbol{\Gamma_x})]$

$\quad = \boldsymbol{\lambda}(\boldsymbol{\Gamma_x}[\omega \circ \sigma'']).\ (\boldsymbol{y_b\ x_1\ \ldots\ x_m})[\omega \circ \sigma']$

$\quad = \boldsymbol{\lambda}(\boldsymbol{\Gamma_x}[\sigma]).\ (\boldsymbol{y_b\ x_1\ \ldots\ x_m})[\sigma]$

$\quad = \boldsymbol{\lambda\Gamma_x}[(\uparrow_{\|\Omega\|^\nabla}\cdot), \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{y_b\ M_1\ \ldots\ M_m}$

$\quad = \boldsymbol{M}$

Type Equality:

$(\boldsymbol{\Pi\Gamma_b.\ B_b})[\omega][\mathrm{id}_\Omega, \sigma'']$

$\quad = (\boldsymbol{\Pi\Gamma_b.\ B_b})[\omega \circ ((\uparrow_{\|\Omega\|^\nabla}\cdot), \sigma'' + \boldsymbol{\Gamma_x})]$

$\quad = (\boldsymbol{\Pi\Gamma_b.\ B_b})[\sigma]$

$\quad = \boldsymbol{\Pi\Gamma_c.\ B_c}$ noting $((\uparrow_{\|\Omega\|^\nabla}\cdot), \omega_0) \leq \sigma$

Case for $\boldsymbol{x'_i} \in \overline{\boldsymbol{x'} \in \delta}$

$\boldsymbol{x'_i}[\omega][(\uparrow_{\|\Omega\|^\nabla}\cdot), \sigma'']$

$\quad = \boldsymbol{x'_i}[\omega][((\uparrow_{\|\Omega\|^\nabla}\cdot), \sigma'') + \boldsymbol{\Gamma_x}]$

$\quad = \boldsymbol{x'_i}[\omega \circ (((\uparrow_{\|\Omega\|^\nabla}\cdot), \sigma'') + \boldsymbol{\Gamma_x})]$

$\quad = \boldsymbol{x'_i}[\sigma]$

$\quad = v_i$ by Definition of σ

$\Omega \vdash g$

$\quad \xrightarrow{*} \epsilon \boldsymbol{y_b} \in \boldsymbol{\Pi\Gamma_c.\ B_c}^{\#}.\ \overline{v}; (\boldsymbol{\lambda\Gamma_x}[\mathrm{id}_\Omega, \overline{v}/\overline{\boldsymbol{x'}}].\ \boldsymbol{y_b\ M_1\ \ldots\ M_m}) \mapsto f'$

by Generalized redEps using σ'' (Lemma B.20.1)

$\quad \to \overline{v}; \boldsymbol{M} \mapsto f''$ by redEps (using c)

$(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{A}^{\#})$ ctx
All elements of $\overline{\boldsymbol{x'}\in\delta}$ occur free in $\boldsymbol{A}$
$g = \epsilon\overline{\boldsymbol{x'}\in\delta}.\ \epsilon\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ \nu\boldsymbol{\Gamma}_{\boldsymbol{g}}.\ (\overline{\boldsymbol{x'}}; \boldsymbol{x} \mapsto f)$
g in $\overline{\mathrm{c}}$
$\mathcal{F} :: \boldsymbol{\Gamma} \gg^{\nabla} \overline{c}$ covers $\nabla\boldsymbol{\Gamma}.\ (\forall\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{A}^{\#}.\ \tau)$

Case: $\mathcal{E} = \dfrac{}{\Omega \vdash \overline{c}$ covers $\nabla\boldsymbol{\Gamma}.\ (\forall(\overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverNewLF}^{\#}$

$\Omega, \boldsymbol{\Gamma} \vdash \mathrm{id}_{(\Omega,\boldsymbol{\Gamma})}, \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{y}/\boldsymbol{x} : \Omega, \boldsymbol{\Gamma}, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{A}^{\#}$ by assumption
$\|\Omega\|^{\nabla}, \boldsymbol{\Gamma} \vdash (\uparrow_{\|(\Omega,\boldsymbol{\Gamma})\|^{\nabla}} \cdot), \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{y}/\boldsymbol{x} : \Gamma, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{A}^{\#}$ by Lemma B.8.3
$\|\Omega\|^{\nabla}, \boldsymbol{\Gamma} \vdash ((\uparrow_{\|\Omega\|^{\nabla}} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{y}/\boldsymbol{x} : \Gamma, \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{A}^{\#}$
by Substitution Rules (Proof by induction on the list)
$\|\Omega\|^{\nabla}, \boldsymbol{\Gamma} \vdash \boldsymbol{y} \in \boldsymbol{A}^{\#}[((\uparrow_{\|\Omega\|^{\nabla}} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}]$ by inversion
We now consider cases on $\boldsymbol{y}$. It is either in $\|\Omega\|^{\nabla}$ or in $\boldsymbol{\Gamma}$.

Case: $\|\Omega\|^{\nabla} \vdash \boldsymbol{y} \in \boldsymbol{A}^{\#}[((\uparrow_{\|\Omega\|^{\nabla}} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}]$

All $\overline{\boldsymbol{x'}}$ occur free in $\boldsymbol{A}$ by assumption
$\|\Omega\|^{\nabla} \vdash (\uparrow_{\|\Omega\|^{\nabla}} \cdot), \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{y}/\boldsymbol{x} : \overline{\boldsymbol{x'}\in\delta}, \boldsymbol{x}\in\boldsymbol{A}^{\#}$
by weakening since all occur free
$\boldsymbol{A}^{\#}[((\uparrow_{\|\Omega\|^{\nabla}} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}] = \boldsymbol{A}^{\#}[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), \overline{v}/\overline{\boldsymbol{x'}}]$ by above

We now show that $\overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{y}/\boldsymbol{x}$ gives us the assignments of ϵ's in g

for all $\boldsymbol{x'_i} \in \overline{\boldsymbol{x'}\in\delta}$
$\boldsymbol{x'_i}[(\mathrm{id}_{\Omega}, \overline{v}/\overline{x'}) + \boldsymbol{\Gamma}]$
$= \boldsymbol{x'_i}[((\uparrow_{\|\Omega\|^{\nabla}} \cdot), \overline{v}/\overline{x'}) + \boldsymbol{\Gamma}]$ by Weakening
$= v_i$

$\boldsymbol{A}^{\#}[(\uparrow_{\|\Omega\|^{\nabla}} \cdot), \overline{v}/\overline{x'}] = \boldsymbol{A}^{\#}[\mathrm{id}_{\Omega}, \overline{v}/\overline{x'}]$ by Weakening

Therefore,
$\Omega \vdash g$
$\xrightarrow{*} \epsilon\boldsymbol{x}\in\boldsymbol{A}^{\#}[\mathrm{id}_{\Omega}, \overline{v}/\overline{\boldsymbol{x'}}].\ \nu\boldsymbol{\Gamma}.\ (\overline{v}; \boldsymbol{x} \mapsto f')$
by generalized redEps using $\overline{v}$ (Lemma B.20.1)
$\to \nu\boldsymbol{\Gamma}.\ (\overline{v}; \boldsymbol{y} \mapsto f'')$ by redEps with $\boldsymbol{y}$

Case: $\boldsymbol{y}{:}\boldsymbol{A_y}$ in $\boldsymbol{\Gamma}$

Either pfUnify or pfSkip applied. — by inversion on $\mathcal{F}$
$\boldsymbol{A_y} = \boldsymbol{A}[((\uparrow_{\|\Omega\|^\nabla} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}]$ — by inversion
pfSkip could not have applied — by above
$\Omega', \boldsymbol{\Gamma} \vdash \omega' = \text{mgu } (\boldsymbol{A_y} \approx \boldsymbol{A}) : \Gamma, \overline{\boldsymbol{x'} \in \delta}$ — by inversion using pfUnify
$\omega' = ((\uparrow_{\Omega'} \cdot) + \boldsymbol{\Gamma}), \omega_A$ — by inversion using pfUnify
There exists a g' in $\overline{c}$ such that
$g' = \epsilon\Omega'.\ \nu\boldsymbol{\Gamma}.\ (\overline{\boldsymbol{x'}}[\omega']; \boldsymbol{y} \mapsto f)$ — by inversion using pfUnify

$\|\Omega\|^\nabla, \boldsymbol{\Gamma} \vdash ((\uparrow_{\|\Omega\|^\nabla} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}, \boldsymbol{y}/\boldsymbol{x} : \Gamma, \overline{\boldsymbol{x'} \in \delta}, \boldsymbol{x} \in \boldsymbol{A}^\#$
Repeated from above
$\|\Omega\|^\nabla, \boldsymbol{\Gamma} \vdash ((\uparrow_{\|\Omega\|^\nabla} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}} : \Gamma, \overline{\boldsymbol{x'} \in \delta}$ — by inversion
There exists a σ' such that
$(((\uparrow_{\|\Omega\|^\nabla} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}) = \omega' \circ \sigma'$ — by mgu (Def. A.2.7)
and $\|\Omega\|^\nabla, \boldsymbol{\Gamma} \vdash \sigma' : \Omega', \boldsymbol{\Gamma}$
and $\sigma' = ((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma}$ — by Lemma B.20.3

We now show that σ'' gives us the assignments of $\epsilon\Omega'$
so that g' matches

for all $\boldsymbol{x'_i} \in \overline{\boldsymbol{x'} \in \delta}$
$$\begin{aligned}
&\boldsymbol{x'_i}[\omega'][(\text{id}_\Omega, \sigma'') + \boldsymbol{\Gamma}] \\
&\quad = \boldsymbol{x'_i}[\omega'][((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma}] \\
&\quad = \boldsymbol{x'_i}[\omega' \circ (((\uparrow_{\|\Omega\|^\nabla} \cdot), \sigma'') + \boldsymbol{\Gamma})] \\
&\quad = \boldsymbol{x'_i}[((\uparrow_{\|\Omega\|^\nabla} \cdot) + \boldsymbol{\Gamma}), \overline{v}/\overline{\boldsymbol{x'}}] \\
&\quad = v_i
\end{aligned}$$
Therefore,
$\Omega \vdash g' \xrightarrow{*} \nu\boldsymbol{\Gamma}.\ (\overline{v}; \boldsymbol{y} \mapsto f')$
by generalized redEps (Lemma B.20.1) with ω''
Note: $f' = f[(\text{id}_\Omega, \sigma'') + \boldsymbol{\Gamma}]$

Case: $\mathcal{E} = \dfrac{\cdot \leq. \Omega}{\Omega \vdash \mathit{nil} \text{ covers } (\forall(\overline{\alpha \in \delta}, \boldsymbol{x} \in \boldsymbol{A}^{\#}).\ \tau)}\ \mathsf{coverEmpty}^{\#}$

Ω does not contain any decs. of $\alpha^* \in \delta^*$ by assumption
$\Omega \vdash \mathrm{id}_\Omega, \overline{v}/\overline{\alpha}, \boldsymbol{y}/\boldsymbol{x} : \Omega, \overline{\alpha \in \delta}, \boldsymbol{x} \in \boldsymbol{A}^{\#}$ by assumption
Ω does not contain any decs. of $\boldsymbol{x} \in \boldsymbol{A}^{\#}$ by above
$\cdot \leq. \Omega$ by assumption
Ω does not contain any decs. $\boldsymbol{x} \overset{\triangledown}{\in} \boldsymbol{A}^{\#}$
by induction (noting that $\mathsf{leWorldAddNew}$ is impossible)
$\Omega \vdash \boldsymbol{y} \in \boldsymbol{A}^{\#}[\mathrm{id}_\Omega, \overline{v}/\overline{\alpha}]$ by inversion
Contradiction as Ω cannot contain the $\boldsymbol{y}$, so this case does not apply!

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\boldsymbol{\Gamma} \vdash^{\mathsf{lf}} \boldsymbol{A} : \boldsymbol{type} \\ c = \epsilon \boldsymbol{x} \in (\boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}).\ \nu\boldsymbol{\Gamma}.\ (\boldsymbol{(x\ \Gamma)} \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\forall \boldsymbol{x} \in \boldsymbol{A}.\ \tau)}\ \mathsf{coverNewLF}$

$\Omega, \boldsymbol{\Gamma} \vdash \mathrm{id}_{(\Omega,\boldsymbol{\Gamma})}, \boldsymbol{M}/\boldsymbol{x} : \Omega, \boldsymbol{\Gamma}, \boldsymbol{x} \in \boldsymbol{A}$ by assumption
$\Omega, \boldsymbol{\Gamma} \vdash \boldsymbol{M} \in \boldsymbol{A}$ by inversion
$\Omega \vdash \boldsymbol{\lambda\Gamma.}\ \boldsymbol{M} \in \boldsymbol{\Pi\Gamma.}\ \boldsymbol{A}$ by **LF_Lam** and isLF
$\Omega \vdash c \to \nu\boldsymbol{\Gamma}.\ (\boldsymbol{M} \mapsto f[(\mathrm{id}_\Omega, \boldsymbol{(\lambda\Gamma.\ M)}/\boldsymbol{x}) + \boldsymbol{\Gamma}])$ by redEps with $\boldsymbol{\lambda\Gamma.}\ \boldsymbol{M}$

Case: $\mathcal{E} = \dfrac{\begin{array}{l}\cdot \vdash \nabla\boldsymbol{\Gamma}.\ \sigma\ \mathsf{wff} \\ c = \epsilon u \in (\nabla\boldsymbol{\Gamma}.\ \sigma).\ \nu\boldsymbol{\Gamma}.\ ((u \backslash \boldsymbol{\Gamma}) \mapsto f)\end{array}}{\Omega \vdash c \text{ covers } \nabla\boldsymbol{\Gamma}.\ (\sigma \supset \tau)}\ \mathsf{coverNewMeta}$

$\Omega, \boldsymbol{\Gamma} \vdash \mathrm{id}_{(\Omega,\boldsymbol{\Gamma})}, v/u : \Omega, \boldsymbol{\Gamma}, u \in \sigma$ by assumption
$\Omega, \boldsymbol{\Gamma} \vdash v \in \sigma$ by inversion
$\Omega \vdash \nu\boldsymbol{\Gamma}.\ v \in \nabla\boldsymbol{\Gamma}.\ \sigma$ by new
$\Omega \vdash c \xrightarrow{*} \nu\boldsymbol{\Gamma}.\ (v \mapsto f[(\mathrm{id}_\Omega, (\nu\boldsymbol{\Gamma}.\ v)/\boldsymbol{x}) + \boldsymbol{\Gamma}])$ by redEps with $\nu\boldsymbol{\Gamma}.\ v$
and $\mathsf{redCaseNew}$ with $\mathsf{redCasePattern}$ noting that $(\nu\boldsymbol{\Gamma}.\ v) \backslash \boldsymbol{\Gamma} \xrightarrow{*} v$

□

B.21 Meta-Theory: Progress

Theorem B.21.1 (Progress)**.**

- *If* Ω *does not contain any decs. of* $\alpha{\in}\delta$ *and* $\Omega \vdash e \in \delta$ *and* e *is not a value, then* $\Omega \vdash e \to f$*.*
- *If* Ω *does not contain any decs. of* $\alpha{\in}\delta$ *and* $\Omega \vdash \mathrm{id}_\Omega, \overline{e}/\overline{\alpha} : \Omega, \overline{\alpha{\in}\delta}$*, then for all* e_i *in* $\overline{e}$ *such that* e_i *is not a value,* $\Omega \vdash e_i \to e_i'$*.*

Proof. By induction on e and $\overline{f}$.

Case: ()

() is a value

Case: u and $\mathcal{E} = \dfrac{(\Omega_1, u{\in}\tau, \Omega_2)\ \mathsf{ctx} \quad (\Omega_1, u{\in}\tau) \leq (\Omega_1, u{\in}\tau, \Omega_2)}{\Omega_1, u{\in}\tau, \Omega_2 \vdash u \in \tau}\,\tau\mathsf{var}$

The context cannot contain any decs. of $\alpha{\in}\delta$. by assumption
This is contradicted here by the existence of u.

Case: $\boldsymbol{x}$

$\boldsymbol{x}$ is a value (as it is an $\boldsymbol{M}$)

Case: $\boldsymbol{M}$

$\boldsymbol{M}$ is a value

Case: (fn $\overline{c}$)

fn $\overline{c}$ is a value

Case: $(e\ \overline{f})$ and $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash e \in \forall\overline{\alpha\in\delta}.\ \tau & \Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha\in\delta}\end{array}}{\Omega \vdash e\ \overline{f} \in \tau[\mathrm{id}_\Omega, \overline{f}/\overline{\alpha}]}\ \mathsf{impE}$

Ω does not contain any decs. of $\alpha\in\delta$ by assumption

Subcase: e is a value and $\overline{f}$ is all values

$e = \mathrm{fn}\ \overline{c}$ by inversion using impl
$\Omega \vdash \overline{c}$ covers $\forall\overline{\alpha\in\delta}.\ \tau$ by inversion using impl
$\Omega \vdash \mathrm{id}_\Omega, \overline{f}/\overline{\alpha} : \Omega, \overline{\alpha\in\delta}$ by assumption
There exists a c_i in $\overline{c}$ such that $\Omega \vdash c_i \xrightarrow{*} (\overline{f} \mapsto e')$ by Lemma B.20.4
$\Omega \vdash e\ \overline{f} \to e'$ by redexLam

Subcase: e is a value and $\overline{f}$ is not all values

There exists an f_i in $\overline{f}$ such that
$\overline{f} = \overline{f_1}; f_i; \overline{f_n}$
and $\Omega \vdash f_i \to f_i'$ by i.h. on $\overline{f}$ with $\mathcal{E}_2$
$\Omega \vdash e\ \overline{f} \to e\ (\overline{f_1}; f_i'; \overline{f_n})$ by redAppR

Subcase: e is not a value

$\Omega \vdash e \to e'$ by i.h. on e with $\mathcal{E}_1$
$\Omega \vdash e\ \overline{f} \to e'\ \overline{f}$ by redAppL

Case: $(\nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e)$ and $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Omega, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash e \in \tau\end{array}}{\Omega \vdash \nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e \in \nabla\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ \tau}\ \mathsf{new}$

Ω does not contain any decs. of $\alpha\in\delta$ by assumption
$\nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e$ is not a value by assumption
e is not a value by Definition of values
$\Omega, \boldsymbol{x}\overset{\nabla}{\in}\boldsymbol{A}^{\#} \vdash e \to f$ by i.h. on e with $\mathcal{E}_1$
$\Omega \vdash \nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e \to \nu\boldsymbol{x}\in\boldsymbol{A}^{\#}.\ e$ by redNew

Case: $(\nu u{\in}\mathcal{W}.\ e)$ and $\mathcal{E} = \dfrac{\mathcal{E}_1}{\Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash e \in \tau}\ \mathsf{newW}$

Ω does not contain any decs. of $\alpha{\in}\delta$	by assumption
$\nu u\overset{\triangledown}{\in}\mathcal{W}.\ e$ is not a value	by assumption
e is not a value	by Definition of values
$\Omega, u\overset{\triangledown}{\in}\mathcal{W} \vdash e \to f$	by i.h. on e with $\mathcal{E}_1$
$\Omega \vdash \nu u\overset{\triangledown}{\in}\mathcal{W}.\ e \to \nu u\overset{\triangledown}{\in}\mathcal{W}.\ e$	by redNewW

Case: $(e\backslash\boldsymbol{x})$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\boldsymbol{x'}{\in}\boldsymbol{A}^{\#}.\ \tau \\ (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}) \leq (\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)\end{array}}{\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \in \tau[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]}\ \mathsf{pop}$

$(\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2)$ does not contain any decs. of $\alpha{\in}\delta$ by assumption

Subcase: e is not a value

$\Omega_1 \vdash e \to f$	by i.h. on e with $\mathcal{E}_1$
$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash e\backslash\boldsymbol{x} \to f\backslash\boldsymbol{x}$	by redPop

Subcase: e is a value of the form $\nu\boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}.\ v$

$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash (\nu\boldsymbol{x'}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}.\ v)\backslash\boldsymbol{x} \to v[\uparrow_{\boldsymbol{x}}\mathrm{id}_{\Omega_1}, \boldsymbol{x}/\boldsymbol{x'}]$ by redPopElim

Subcase: e is a value of the form $\mathrm{fn}\ \overline{c}$

$\Omega_1, \boldsymbol{x}\overset{\triangledown}{\in}\boldsymbol{A}^{\#}, \Omega_2 \vdash (\mathrm{fn}\ \overline{c})\backslash\boldsymbol{x} \to \mathrm{fn}\ (\overline{c}\backslash\boldsymbol{x})$ by redPopFn

Case: $(e\backslash u)$ and $\mathcal{E} = \dfrac{\begin{array}{l}(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)\ \mathsf{ctx} \\ \mathcal{E}_1 :: \Omega_1 \vdash e \in \nabla\mathcal{W}_2.\ \tau \\ \mathcal{W} \leq \mathcal{W}_2 \\ (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}) \leq_{\mathcal{W}_2} (\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)\end{array}}{\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \in \tau}\ \mathsf{popW}$

$(\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2)$ does not contain any decs. of $\alpha{\in}\delta$ by assumption

Subcase: e is not a value

$\Omega_1 \vdash e \to f$ by i.h. on e with $\mathcal{E}_1$
$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash e\backslash u \to f\backslash u$ by $\mathsf{redPopW}$

Subcase: e is a value of the form $\nu u'\overset{\triangledown}{\in}\mathcal{W}_2.\ v$

$\Omega_1, u\overset{\triangledown}{\in}\mathcal{W}, \Omega_2 \vdash (\nu u'\overset{\triangledown}{\in}\mathcal{W}_2.\ v)\backslash u \to v[\uparrow_u \mathrm{id}_{\Omega_1}, u/u']$ by $\mathsf{redPopElimW}$

Case: $(e,\ f)$ and $\mathcal{E} = \dfrac{\Omega \vdash (\exists\alpha{\in}\delta.\ \tau)\ \mathsf{wff} \qquad \overset{\mathcal{E}_1}{\Omega \vdash e \in \delta} \qquad \overset{\mathcal{E}_2}{\Omega \vdash f \in \tau[\mathrm{id}_\Omega, e/\alpha]}}{\Omega \vdash (e,\ f) \in \exists\alpha{\in}\delta.\ \tau}\ \mathsf{pairI}$

Ω does not contain any decs. of $\alpha{\in}\delta$ by assumption
$(e,\ f)$ is not a value by assumption

Subcase: e is not a value

$\Omega \vdash e \to e'$ by i.h. on e with $\mathcal{E}_1$
$\Omega \vdash (e,\ f) \to (e',\ f)$ by $\mathsf{redPairL}$

Subcase: f is not a value

$\Omega \vdash f \to f'$ by i.h. on f with $\mathcal{E}_2$
$\Omega \vdash (e,\ f) \to (e,\ f')$ by $\mathsf{redPairR}$

Case: $(\mu u{\in}\tau.\ e)$

$\Omega \vdash \mu u{\in}\tau.\ e \to e[\mathrm{id}_\Omega, \mu u{\in}\tau.\ e/u]$ by redFix

. . . . . .

Case: *nil*

for all e_i in *nil* such that e_i is not a value,
$\Omega \vdash e_i \to e_i'$ Since *nil* has no elements

Case: $(\overline{f'}; f)$ and $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'} : \Omega, \overline{\alpha' \in \delta'} & \Omega \vdash f \in \delta[\mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}]\end{array}}{\Omega \vdash \mathrm{id}_\Omega, \overline{f'}/\overline{\alpha'}, f/\alpha : \Omega, \overline{\alpha' \in \delta'}, \alpha \in \delta}\ \mathsf{tpSubInd}$

Ω does not contain any decs. of $\alpha \in \delta$ by assumption
for all e_i in $\overline{f'}$ such that e_i is not a value,
$\Omega \vdash e_i \to e_i'$ by i.h. on $\overline{f'}$ with $\mathcal{E}_1$

Subcase: f is a value

for all e_i in $(\overline{f'}; f)$ such that e_i is not a value,
$\Omega \vdash e_i \to e_i'$ by above

Subcase: f is not a value

$\Omega \vdash f \to f_2$ by i.h. on f with $\mathcal{E}_2$
for all e_i in $(\overline{f'}; f)$ such that e_i is not a value,
$\Omega \vdash e_i \to e_i'$ by above

□

B.22 Meta-Theory: Type Safety

Theorem B.22.1 (Type Safety)**.**
Delphin is a type safe language, i.e. if Ω *does not contain any decs. of* $\alpha{\in}\delta$ *and* $\Omega \vdash e \in \delta$ *and* e *is not a value, then there exists an* f *such that* $\Omega \vdash e \longrightarrow f$ *and* $\Omega \vdash f \in \delta$*.*

Proof. Follows directly from type preservation (Theorem B.19.4) and progress (Theorem B.21.1). □