[DRAFT] The Calculus of Dependent Lambda Eliminations*

Aaron Stump
Computer Science, The University of Iowa, Iowa City, Iowa, USA
astump@acm.org

Abstract

Modern constructive type theory is based on pure dependently typed lambda calculus, augmented with user-defined datatypes. This paper presents an alternative called the Calculus of Dependent Lambda Eliminations, based on pure lambda encodings with no auxiliary datatype system. New typing constructs are defined which enable induction, as well as large eliminations with lambda encodings. These constructs are constructor-constrained recursive types, and a lifting operation to lift simply typed terms to the type level. Using a lattice-theoretic denotational semantics for types, the language is proved logically consistent. The power of CDLE is demonstrated through several examples, which have been checked with a prototype implementation called Cedille.

1 Introduction

Lambda encodings are schemes for representing datatypes and related operations as pure lambda terms. The Church encoding, where data are encoded as their own fold functions, is the best known (Church, 1941), and is typable in System F (Böhm & Berarducci, 1985). Lambda encodings were abandoned as a basis for constructive type theory almost thirty years ago, due to the following problems:

1. Accessors (like predecessor for numerals, or head and tail for lists) are provably asymptotically inefficient with the Church encoding (Parigot, 1989).
2. Induction principles are not derivable for lambda encodings (Geuvers, 2001).
3. Large eliminations, which compute types from data, are not possible with lambda-encoded data, at least not in normalizing type theories. This is because such theories distinguish different levels of the language, such as terms, types, kinds, etc., and one cannot apply a function at one level to compute a term at a higher level. Also, using impredicative quantification \( \forall X : \ast \) one level up leads to failure of normalization and hence logical consistency (Coquand, 1986).
4. Without large eliminations, it is not possible to prove basic negative facts about lambda-encoded data, like \( 0 \neq 1 \) (Werner, 1992).

* This paper is under review for a special issue of the Journal of Functional Programming as of January 22, 2016. The version you are reading is almost certainly not the final version of this paper.
On the positive side, there is one powerful benefit of typed lambda-encodings, not available with primitive datatypes:

- Higher-order encodings – where datatypes contain embedded functions whose types have negative occurrences of the datatype symbol – are permitted without violating normalization. With primitive datatypes as in Coq or Agda, negative occurrences of the datatype in the datatype definition very easily lead to failure of normalization.

Parigot solved the first problem with an encoding where data are represented not as iterators but as recursors: every call to the iterated function is presented with the predecessor number as well as the result of iteration on that number. So 2 is encoded as $\lambda s.\lambda z.s\,1\,(s\,0\,z)$. While in theory the space required for normal forms is exponential, in practice closure-based implementations of lambda calculus compute efficiently with Parigot encodings (Koopman et al., 2014). And Parigot encodings can be typed in a normalizing extension of System F with positive-recursive types (cf. (Abel & Matthes, 2004; Mendler, 1988)).

The present paper proposes new type constructs to solve the problems of induction and large eliminations. Earlier work by Fu and Stump on System S proposes a solution to the problem of induction using a typing construct called self types to allow the type to refer to the subject of the typing via bound variable $x$ in $tx.T$ (Fu & Stump, 2014). To prove consistency, they rely on a dependency-eliminating translation to System $F_\omega$ plus positive-recursive types. This method is not applicable to analyze a system with large eliminations, where dependence of types on terms is fundamental. In the present paper, a deeper analysis of intrinsically inductive lambda encodings is undertaken, with a direct lattice-theoretic semantics which can account for large eliminations. In the rest of this section, the two new features that enable intrinsically inductive lambda-encoding and large eliminations with lambda-encodings, respectively, are surveyed. Then we turn to the definition (Sections 2 and 3) and analysis (Sections 5 and 6) of the new type theory incorporating these features, called the Calculus of Dependent Lambda Eliminations (CDLE). This system is a type assignment system, not suitable for implementation. An algorithmic approach to CDLE, which has been implemented in a prototype tool called Cedille, is then considered, together with examples (Sections 7 and 9).

1.1 Constructor-constrained recursive types

To define intrinsically inductive lambda-encodings, we begin with the dependent intersection types of Kopylov (Kopylov, 2003). We will denote these types with prefix notation $tx:T.T'$ instead of Kopylov’s $x:T \cap T'$. Let $S$ and $Z$ be meta-level abbreviations for $\lambda n.\lambda s.\lambda z.s\,(n\,s\,z)$ and $\lambda s.\lambda z.z$, respectively. Also, we will make use of a top type $\mathcal{U}$, inhabited by all closed $\lambda$-abstractions. Now at the meta-level, define a sequence of types by recursion on meta-level natural number $k$, with increasing support for dependent typing:

$$
\begin{align*}
\text{Nat}_0 & := \mathcal{U} \\
\text{Nat}_{k+1} & := tn:\text{Nat}_k \forall P:\text{Nat}_k \rightarrow \ast, \\
& \quad (\forall n:\text{Nat}_k, P\,n \rightarrow P\,(S\,n)) \rightarrow P\,Z \rightarrow P\,n
\end{align*}
$$

$\text{Nat}_{k+1}$ denotes the subset of $\text{Nat}_k$ for which induction holds, for predicates on $\text{Nat}_k$. We use intersection types, because the natural proof that $n$ is inductive may be identified, in a
type assignment system such as we will consider, with $n$ itself (cf. (Leivant, 1983), Section 9.5 of (Krivine, 2011)). We will see this in more detail below (Section 4).

Now the goal is to internalize the limit of this sequence of types as a single type $\mathbb{N}$, using a positive-recursive type. This is not possible with standard forms of recursive types, due to type dependency. For suppose we tried to define $\mathbb{N}$ as

$$
\mu \text{Nat}: \kappa. \nu n: \text{Nat}.
$$

To kind this type, we would have to kind $(\nu n: \text{Nat}. P n \rightarrow P (S n)) \rightarrow P Z \rightarrow P n$

Semantically, this will be interpreted as the greatest lower bound of the decreasing sequence of meanings for $\text{Nat}_k$, defined above. The key new idea is to include this set $\Theta$ (here, $S \in \text{Nat} \rightarrow \text{Nat}, Z \in \text{Nat} \rightarrow \text{Nat}$) of typings which hold for $\forall \kappa$ and are preserved as we pass further into the sequence. This is so that we can kind the body of the $\nu$-type. For the semantic analysis, it will turn out to be critical for $\Theta$ to hold not just for the decreasing sequence of meanings, but also for the greatest lower bound of that sequence. Without some restriction, this appears not to be guaranteed. Here, we require that each typing constraint in $\Theta$ be of the form $\Pi \eta_1 : T_1 \cdots \Pi \eta_n : T_n, T$, where the $\nu$-bound variable occurs only positively in $T_1, \ldots, T_n$, and only at the head of $T$ (i.e., $T$ is either $X$ or $X$ applied to some $X$-free expressions). $\text{Nat} \rightarrow \text{Nat}$ meets this requirement, as a simple example, but so do more complex types.

CDLE’s type system has a rule for folding and unfolding $\nu$-types. There is also a rule for typing of constructors: $\Gamma \vdash t : [N/X]T$ is derivable for all $t \in T$ in the constructor set $\Theta$, once a $\nu$-type $N = \nu X : \kappa \mid \Theta, T'$ has been kinded in context $\Gamma$.

1.2 Lifting terms to the type level

The basic idea for supporting large eliminations with lambda encodings is to lift expressions explicitly from the term level of the language to the type level. While it is well-known that one cannot lift the entire term language to the type level without losing normalization (Coquand, 1986), there is no problem with lifting simply typed terms. For example, the term $\lambda s. \lambda z : s \ z$ representing 2 in the Church encoding can be lifted to the type level as $\lambda s : \kappa \rightarrow \kappa. \lambda z : \kappa \rightarrow (s \ z)$, for any particular kind $\kappa$ (for example, $\star$, the kind which classifies types). Certainly the ability to do arithmetic with simply-typed lambda encodings is limited (cf. (Leivant, 1991)). But typically for large eliminations, one seeks...
just to do a single fold over the datatype to compute a type from the data. For example, for statically typed printf, as proposed by Augustsson (Augustsson, 1998), one wishes to compute the type of the rest of the arguments to printf from the format string. This requires just a single fold.

CDLE introduces a novel construct \( \uparrow_L t \), representing the type obtained by lifting a simply-typed term \( t \) to the type level. The type \( L \) is a lifting type, which serves to constrain the type of \( t \) to be simply typed, and also shows how that type should be lifted to a kind. For example, to lift Church-encoded 2 to the type level, one writes \( \uparrow_{(\ast \to \ast)} 2 \), where \( \ast \) is a primitive lifting type used to represent the kind \( \ast \). We are not lifting 2 at its polymorphic type \( \forall X : \ast. (X \to X) \to X \to X \), of course, as this type is not permitted at the kind level. Instead, we are lifting an instantiation \( (X \to X) \to X \to X \) of this type, where \( \ast \) indicates the instantiation points.

One technical issue that must be addressed with this idea is the presence of variables which occur free inside a lifting expression. For a simple example, suppose we have a free variable \( x \) of type \( \forall X : \ast. X \to X \), and consider this type, where \( x \) is being instantiated to \( \ast \):

\[
\uparrow_{\ast \to \ast} \lambda y. x
ty
\]

It is tempting always to push lifting across \( \lambda \)-abstractions, but if we do that here, we will get:

\[
\lambda y : \ast. \uparrow_{\ast} (x
ty)
\]

The body is not typable, because \( x \) (instantiated to have type \( \ast \to \ast \)) is being applied to a type, namely \( y \) of type \( \ast \).

One can imagine several solutions to this problem. Here, we opt not to push lifting across a series of \( \lambda \)-abstractions unless the body is of the form \( x \bar{t} \), where \( x \) is bound in that series. We will form type-level \( \beta \)-redexes for the arguments \( \bar{t} \), in case they are not headed by a variable in the series. So we will lift the successor \( \lambda s. \lambda z. s (n
ds \ z) \) of Church-encoded \( n \) to

\[
\lambda s : \ast \to \ast. \lambda z : \ast. s (\uparrow_{(\ast \to \ast)} (\ast \to \ast) \lambda s. \lambda z. n \ s \ z \ s \ z)
\]

Despite this trick, we will still need some additional conversion principles for lifting, which we will see below.

2 Syntax

Figure 1 gives the syntax of CDLE. We are separating clauses of the grammars with ||, to avoid confusion with the single vertical bar in the syntax for \( \nu \)-types. We use \( \forall \) consistently in the types for functions for which no argument is explicitly given when the function is called. So these are implicit products, as introduced by Miquel (Miquel, 2001). \( \Pi \) is used for explicit products, where an argument is required when applied. We do not use type-level implicit products.

The type \( \mathcal{U} \) is a universal type, inhabited by all closed \( \lambda \)-abstractions. In the construct \( \forall X : \kappa | \Theta, T \), the scope of bound variable \( X \) is \( \Theta \) and the body \( T \). We are using \( \forall \) instead of \( \mu \), because our semantics will make \( \forall X : \kappa | \Theta, T \) the greatest fixed-point of \( T \). Nevertheless, we will focus here on using this type for inductive datatypes, not coinductive ones (which are outside the scope of this paper). Several rules related to \( \forall \)-types will make use of a
variables \textit{x, X}
terms \textit{t} :::= x || \lambda x.t || t'
kinds \textit{\kappa} :::= \star || \Pi \textit{X}.T.\kappa || \Pi X.\kappa.\kappa'
types \textit{T} :::= X || \Pi x:T.\kappa || \forall X:T.\kappa || \forall x:T.\kappa' || \\
\lambda x:T.\kappa || \nu X:T.\kappa || \lambda x:T.\kappa' || \\
\lambda X:\kappa.\kappa || T \kappa || T \kappa' || \forall \kappa || \uparrow_L t
lifting types \textit{L} :::= \star || L \rightarrow L'
constructor sets \textit{\Theta} :::= \cdot || t \in T, \Theta
typing contexts \textit{\Gamma} :::= \cdot || \textit{\Gamma}, X: \kappa || \textit{\Gamma}, t:T || \textit{\Gamma}, t \in T

Fig. 1. Syntax of CDLE, and typing contexts

notation \textit{\Upsilon}_\kappa for a top type at kind \kappa. We define this by recursion on \kappa:
\begin{align*}
\Upsilon_* &= \Upsilon \\
\Upsilon_{\Pi X.T.\kappa} &= \lambda X:T.\Upsilon_{\kappa} \\
\Upsilon_{\Pi X.\kappa.\kappa'} &= \lambda X.\kappa.\Upsilon_{\kappa'}
\end{align*}

We usually elide the final ",", " from constructor sets \textit{\Theta} and typing contexts \textit{\Gamma}. We use other standard notations for typed lambda calculus, in particular \( T \rightarrow T' \) for \( \Pi x:T.\kappa || \forall x:T.\kappa' \) when \textit{x} is not free in \( T' \). The type \( t x:T.\kappa \) is a dependent intersection type, as introduced by Kopylov (Kopylov, 2003). We use \textit{t} \(\in T \) to denote a constraint that term \textit{t} has type \textit{T}, as opposed to a declaration of a variable \textit{x} to have type \textit{T} (written \( x:T \)). Here we see one unusual feature of the type system, which is that the context may contain hypotheses that a term has a given type \( (t \in T) \). This feature comes in with the constructor-constrained recursive types \( \forall X: \kappa \Theta.T \). We will see how to avoid it when we turn to the Cedille implementation of CDLE (Section 7). We implicitly assume that \textit{\Gamma} does not declare any variable \textit{x} or \textit{X} twice, and that bound variables are renamed to enforce this. If the set \textit{\Theta} is empty, we may write \( \forall X: \kappa.T \) instead of \( \forall X: \kappa \Theta.T \).

### 3 Type Assignment

We consider now the type assignment rules for CDLE. These include a number of features that would make them unsuitable for direct use in a practical implementation. By accepting some nonalgorithmic features, we can more easily establish, in CDLE, a firm theoretical foundation for the practical implementation of dependent typing based on pure lambda encodings. We will see how this works out when we turn to the Cedille implementation (Section 7).

The typing rules for terms and constructor sets are in Figure 2. We also use kinding rules for types, in Figure 3. Figure 4 gives kinding rules for constructor sets, and superkinding rules. Figure 6 defines judgements imposing the restriction mentioned above on the form of types in constructor sets \textit{\Theta}. To express our positivity requirement for kinding \( \nu \)-types, we use a judgment \( X \in_p T \) for \( p \in \{+, -\} \). The definition is unsurprising, so we omit it. Note, however, that a more flexible approach is proposed in (Abel & Matthes, 2004), using kind-level variance annotations. Adding these to CDLE should be straightforward future work. We also write \( FV(T) \) for the set of free variables (term and type) in \( T \), and \( \text{decl}(\Gamma) \) for the set of variables (term and type) declared in \( \Gamma \) via \( x:T \) or \( X: \kappa \). We write \textit{terms}(\textit{\Theta}) for the set of terms \textit{t} with constraint \( t \in T \) listed in \Theta for some \textit{T}.
\[
\begin{align*}
\frac{\langle x : T \rangle \in \Gamma}{\Gamma \vdash x : T} & \quad \frac{FV(\lambda x.t) \subseteq \text{decl}(\Gamma)}{\Gamma \vdash \lambda x.t : \square} \\
\frac{\Gamma \vdash t : T' \quad \Gamma \vdash T \triangleright T' \quad \Gamma \vdash T : \star}{\Gamma \vdash t : T} & \quad \frac{\Gamma \vdash t : T' \quad \Gamma \vdash T \triangleright T}{\Gamma \vdash t : T} \quad \frac{\Gamma \vdash t' : T \quad t = \beta t'}{\Gamma \vdash t : T} \\
\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash (\lambda x.t) : T'}{\Gamma \vdash \lambda x.t : \Pi x : T.T'} & \quad \frac{\Gamma \vdash t : \Pi x : T_1, T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t : [t'/x]T_2} \\
\frac{\Gamma \vdash \lambda x.t : \Pi x : T.T'}{\Gamma \vdash \lambda x.t : \Pi x : T.T'} & \quad \frac{(t \in T) \in \Gamma}{\Gamma \vdash t : t} \\
\frac{\Gamma \vdash t : T \quad \Gamma \vdash \forall X : \kappa t}{\Gamma \vdash t : \forall x : \kappa, T} & \quad \frac{\Gamma \vdash t : \forall x : T. T'}{\Gamma \vdash t : [T'/X]T} \quad \frac{\Gamma \vdash t : [t'/x]T'}{\Gamma \vdash t : [t/x]T'} \\
\frac{\Gamma \vdash t : T \quad \Gamma \vdash t : [t'/x]T'}{\Gamma \vdash t : T} & \quad \frac{\Gamma \vdash t : t : T \quad \Gamma \vdash t : T}{\Gamma \vdash t : t : T \quad \Gamma \vdash t : t : T} \\
\frac{\Gamma \vdash t : T \quad \Gamma \vdash t : [t'/x]T'}{\Gamma \vdash t : [x/T']T} & \quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash t : T}{\Gamma \vdash t : t : T} \\
N = \forall X : \kappa \mid \Theta_1, t \in T, \Theta_2, T' \quad \Gamma \vdash N : \kappa \\
\frac{\Gamma \vdash t : [N/X]T}{\Gamma \vdash t : t}.
\end{align*}
\]

Fig. 2. Typing of terms and constructor sets

\[
\begin{align*}
\frac{\Gamma \vdash T_1 : \star \quad \Gamma, x : T_1 \vdash T_2 : \star}{\Gamma \vdash \forall x : T_1, T_2 : \star} & \quad \frac{\Gamma \vdash \kappa : \square \quad \Gamma, X : \kappa \vdash T : \star}{\Gamma \vdash \Pi x : T_1, T_2 : \star} \\
\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash t : T, T' : \star} & \quad \frac{\Gamma \vdash t : \Pi x : T_1, T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t : [t'/x]T_2} \\
\frac{\Gamma \vdash T : \Pi x : T_1, T_2 : \kappa \quad \Gamma \vdash t : t'}{\Gamma \vdash t : \Pi x : T_1, T_2 : \kappa} & \quad \frac{\Gamma \vdash \Pi x : T_1, T_2 : \kappa \quad \Gamma \vdash T' : \kappa}{\Gamma \vdash T \vdash \lambda x : T, T'. \Pi x : T, T' : \kappa} \\
\frac{\Gamma \vdash \Pi x : T_1, T_2 : \kappa \quad \Gamma \vdash t : t'}{\Gamma \vdash T : T \vdash \Pi x : T_1, T_2 : \kappa} & \quad \frac{\Gamma \vdash t : T \vdash [t'/x]T}{\Gamma \vdash t : T} \\
\frac{\Gamma \vdash \forall X : \kappa \mid \Theta_1, t \in T, \Theta_2, T' \quad \Gamma \vdash N : \kappa}{\Gamma \vdash t : t} \\
\frac{\Gamma \vdash \forall X : \kappa \mid \Theta_1, t \in T, \Theta_2, T' \quad \Gamma \vdash N : \kappa}{\Gamma \vdash t : t}.
\end{align*}
\]

Fig. 3. Kinding of types

\[
\begin{align*}
\frac{\Gamma \vdash \cdots : \star}{\Gamma \vdash t : t \in T, \Theta : \star} & \quad \frac{\Gamma \vdash \cdots : \star}{\Gamma \vdash t : t} \\
\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash \kappa : \square \quad \Gamma \vdash [x/X]T}{\Gamma \vdash \Pi x : T \vdash [x/X]T \vdash \Pi x : T, \kappa : \square} & \quad \frac{\Gamma \vdash \Pi x : T, \kappa : \square \quad \Gamma, x : T \vdash \kappa : \square \quad \Gamma \vdash [x/X]T}{\Gamma \vdash \Pi x : T, \kappa : \square} \\
\frac{\Gamma \vdash \cdots : \star}{\Gamma \vdash t : t} & \quad \frac{\Gamma \vdash \cdots : \star}{\Gamma \vdash t : t}.
\end{align*}
\]

Fig. 4. Kinding of constructor sets, and superkinding
Our system has a direct-computation typing rule, as in Nuprl (Constable et al., 1986). This rule uses a relation $\equiv_\beta$, which is just standard $\beta$-equivalence of pure untyped lambda calculus. Direct computation allows us to use a more general typing rule for $\lambda$-abstractions: in the premise, we apply the $\lambda$-abstraction, rather than typing its body. Note that the rule also implies type preservation under $\beta$-reduction; the soundness of this will be established with our semantics. CDLE has forward and backward conversion rules for typing, using a directed conversion relation $\triangleright$. The computation rules (central axioms) for $\triangleright$ are given in Figure 5. Additional rules including transitivity, reflexivity, and rules making the relation a congruence are straightforward, and omitted for space reasons.\footnote{See supplementary document.}

\begin{figure}[h]
\centering
\begin{align*}
N &= vX : X \Theta, T \\
\Gamma \vdash N : [N/X]T &\quad \Gamma \vdash (\lambda x : T.T') t \triangleright [t/x]T' \\
\Gamma \vdash \Pi x : T.T' &\quad \Gamma \vdash (\lambda X : X.T) T' \triangleright [T'/X]T \\
\frac{t \rightarrow^\ast t'}{\Gamma \vdash t \triangleright T t'} &\quad \frac{\Gamma \vdash t : T \quad X \notin FV(T')}{\text{lift}_L(t) = T} \\
\frac{\Gamma \vdash \forall x : T.T' \triangleright T'}{\Gamma \vdash \forall x : T.T' \triangleright T'} &\quad \frac{\Gamma \vdash \Lambda x : T. t \triangleright T}{\text{lift}_L(\lambda x : T. t) \triangleright \Lambda x : T. (\text{lift}_L(t))}
\end{align*}
\caption{Computation rules for conversion}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{CtorTp}_X T &\quad \text{Ctors}_X \Theta &\quad t \notin \text{terms}(\Theta) &\quad X \in^+ T_1 \quad \text{CtorTp}_X T_2 \\
\text{Ctors}_X (t \in T, \Theta) &\quad \text{HeadOnly}_X T &\quad \text{X} \notin FV(T) &\quad \text{HeadOnly}_X T
\end{align*}
\caption{Definition of helper judgements for constructor sets}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
lift(\ast) &= \ast \\
lift(L \rightarrow L') &= lift(L) \rightarrow lift(L') \\
|\cdot|_X &\quad |S|_X &\quad |S'|_X \\
\text{lift}_{L_1 \rightarrow L_2}.(\lambda x.t) &= \lambda x \cdot \text{lift}_{L_1}.(\text{lift}_{L_2}.(\lambda x.t)) \\
\text{lift}_{L_1}(x \bar{t}) &= x \text{ liftargs}_{L_1}(\bar{t}), \text{ if } (x \mapsto L') \in v \\
\text{liftargs}_{L_1 \rightarrow L_2,v}(t, \bar{t}) &= (((\lambda x \mapsto L_1) \lambda x.t) v), \text{ liftargs}_{L_1,v}(\bar{t}) \\
\text{liftargs}_{L_1}(\cdot) &= .
\end{align*}
\caption{Meta-level functions related to lifting}
\end{figure}
(nonsymmetric) relation just means that it may be necessary to perform a sequence of forward and backward conversions; the key point is that the backward conversions require an additional kinding derivation. One could also consider a conversion rule for kinding, but simple situations that would require this can be solved by type-level \( \eta \)-expansion, and including kind-level conversion complicates inversion on kinding. So to avoid such distractions this is omitted from CDLE. We will consider the nature of CDLE conversion further in Section 5.1 below. Several of the rules deal with lifting. We will see more about how they work below (Section 10.2).

The kinding rule for types \( \Gamma \vdash t \) (in Figure 3), uses a meta-level function \( \text{lift}(\cdot) \), defined in Figure 7, which maps lifting types to kinds as follows. The idea is to lift a type like \( * \rightarrow * \) to the kind \( * \rightarrow * \). We could also allow a lifting type \( \Pi x : T . L \) to enable lifting the bodies of abstractions without lifting the classifier for the bound variable, for quantifications over terms. We omit this here for simplicity, and because it is not required for our examples. We cannot lift implicit products, because CDLE does not have these at the type level, and adding them introduces semantic complications. We also use a meta-level function \( |L|_\chi \), also defined in Figure 7, to turn a lifting type into a type, replacing \( \star \) with \( \chi \).

Figure 7 defines a third function \( \text{liftargs}\_\_\_\_(-) \), which attempts to lift a term to a type (but may be undefined). We use vector notation \( \vec{t} \) for a possibly empty sequence \( t_1, \ldots, t_n \) of terms, where \( \cdot \) denotes the empty sequence. We write \( |\vec{t}| \) for the length of the sequence. The notation \( x \vec{t} \) means that \( x \) is applied in a left-nested fashion to the terms \( \vec{t} \). This \( \text{liftargs}\_\_\_\_(-) \) function attempts to push the lifting operator \( (\uparrow) \) down into a \( \lambda \)-abstraction. Roughly speaking, it tries to turn \( \uparrow (\lambda \vec{x} . x_i \vec{t}) \) into \( \lambda \vec{x} : \kappa . x_i T' \), where the kinds \( \kappa \) are derived from the lifting type given as the first argument to \( \text{liftargs}\_\_\_\_(-) \), and the types \( T' \) are new lifting types derived from the arguments \( \vec{t} \).

In describing these syntactic operations, we use some special notational conventions in Figure 7 with the meta-variable \( \nu \), which ranges over sequences of bindings \( x \mapsto L \) (where \( L \) is a lifting type). In the first equation for the \( \text{liftargs}\_\_\_\_(-) \) helper function, we write \( \lambda \nu . t \) to mean that all the variables listed in \( t \) should be \( \lambda \)-bound around \( t \), in the order they appear in \( \nu \). Also, we write \( v \mapsto L \) to mean that the lifting types in \( v \) should be added as domain types, in order, for an arrow type around \( L \). And we write \( t \mapsto v \) to mean that the variables in \( v \) should be given as arguments, in order, for an application of \( t \). These notations are used to implement the idea discussed in Section 1.2, of creating type-level \( \beta \)-redexes when pushing lifting to arguments.

### 4 Church-encoded natural numbers

As discussed in Section 1.1, we use the following definition for the type \( \mathbb{N} \) of the natural numbers, where \( S \) and \( Z \) are meta-level abbreviations for \( \lambda n . \lambda s . \lambda z . s \ (n \ s \ z) \) and \( \lambda s . \lambda z . z \):

\[
\nu \mathsf{Nat} : * \mid S \in \mathsf{Nat} \rightarrow \mathsf{Nat}, \ Z \in \mathsf{Nat}.
\]

\[
t : \mathsf{Nat} \ni P : \mathsf{Nat} \rightarrow *.
\]

\[
(\forall n : \mathsf{Nat} . P \ n \rightarrow P \ (S \ n)) \rightarrow P \ Z \rightarrow P\ n
\]

These are Church-encoded numbers, because the type for the input \( s \) for successor, namely \( \forall n : \mathsf{Nat} . P \ n \rightarrow P \ (S \ n) \), uses an implicit product (\( \forall \)). For the Parigot encoding, one just changes this to an explicit product (\( \Pi \)). We will mostly focus on the Church encoding.
in this paper, since it is somewhat simpler and more familiar than the asymptotically more
time-efficient Parigot encoding.

Let us see now in detail how to kind this type, using the \( \nu \)-kinding rule:

\[
X \in {}^+ T \\
\Gamma \vdash \kappa : \square \quad \text{Ctors}_X \Theta \\
\Gamma, X : \kappa \vdash \Theta : \star \\
\Gamma \vdash \left[ \frac{\nu \kappa}{X} \right] \Theta \\
\Gamma, X : \kappa, \Theta \vdash \left[ T / X \right] \Theta \\
\Gamma, X : \kappa, \Theta \vdash T : \kappa
\]

The first premise is obvious, though note that \( \text{Nat} \) occurs positively but not
strictly positively; the occurrences in the body of the type are in the domain parts of an even number of
abstractions. The second premise is trivial. For the third premise, we can confirm easily that
the constructor set for this example satisfies \( \text{Ctors}_\text{Nat} \), as required. For the fourth premise:
with \( \text{Nat} : \star \) in the context, we can kind the constructor set
\( S \in \text{Nat} \to \text{Nat}, Z \in \text{Nat} \). For the
fifth, we can assign \( \Upsilon \to \Upsilon \) to \( S \), using our direct-computation rule:

\[
\Gamma \vdash n : \Upsilon \vdash \lambda s. \lambda z.s \left( n s z \right) / \Upsilon \\
S n =_\beta \lambda s. \lambda z.s \left( n s z \right) \\
\Gamma, n : \Upsilon \vdash S n : \Upsilon \\
\Gamma \vdash S : \Upsilon \to \Upsilon
\]

We can also assign \( \Upsilon \) to \( Z \).

For the sixth premise, we must show that our constructor set is preserved by the body of
the \( \nu \)-type. So in the context (call it \( \Gamma \)) \( \text{Nat} : \star, S \in \text{Nat} \to \text{Nat}, Z \in \text{Nat} \), we must show the
following typings, where we write \( \text{NAT} \) to abbreviate the body of the \( \nu \)-type:

- \( \Gamma \vdash S : \text{NAT} \to \text{NAT} \)
- \( \Gamma \vdash Z : \text{NAT} \)

Let us just consider the second (the first also holds). Expanding \( \text{NAT} \), we see we must show

\[
\Gamma \vdash Z : \text{Nat} \forall P : \text{Nat} \to \star, \\
\left( \forall n : \text{Nat}. P n \to P \left( \lambda s. \lambda z.s \left( n s z \right) \right) \right) \to \\
P \left( \lambda s. \lambda z.z \right) \to P n
\]

From our constraints in \( \Gamma \), we have that \( \Gamma \vdash Z : \text{Nat} \). So we can assign the first type in the
dependent intersection. It remains to assign the second type, where \( n \) is instantiated with
\( Z \). For this, we can apply some introduction rules (together with direct computation) to
reduce the problem to the following typing, where types like \( P Z \) are kindable, from the
constraints in \( \Gamma \):

\[
\Gamma, P : \text{Nat} \to \star, s : \forall n : \text{Nat}. P n \to P \left( S n \right), z : P Z \vdash z : P Z
\]

This holds by the variable typing rule.

For the seventh premise, we must be able to assign kind \( \star \) to the body of the \( \nu \)-type,
assuming \( \text{Nat} : \star \) and the constructor set have been added to the context. The interesting
observation for this is that the applications of \( P \) can be kinded. For example, to kind \( P \left( S n \right) \),
we use the constraint \( S \in \text{Nat} \to \text{Nat} \) to assign type \( \text{Nat} \) to \( S n \).

If we have a term of type \( \mathbb{N} \), then by unfolding the \( \nu \)-type and then taking the second
projection of the dependent intersection, we can use that term for dependently typed it-
erations; for example, inductive proofs. Of course, we can also use it for simply typed
It may be of interest to some readers to know that CDLE validates axiom K for equality types (Hofmann & Streicher, 1998). K, which is equivalent to uniqueness of identity proofs, is all one must add to Martin-Löf Type Theory (MLTT) to support dependent pattern-matching, and thus is desirable for practical programming with dependent types (Goguen et al., 2006). But K is incompatible with Homotopy Type Theory (HoTT) (Univalent Foundations Program, 2013), where distinguishing proofs of the same equality is essential to the approach. So CDLE is not appropriate, without significant modification, for HoTT.

CDLE allows one to define an equality type with both J- and K-style elimination. The definition is in Figure 8, where we are writing \( T \land T' \) for \( \iota_x : T \to T' \) when \( x \notin \text{FV}(T') \). Note that here, the top type \( \forall_{A \to} \) that is used when kinding \( ^{JK} \) is defined (at the meta-level) to be \( \lambda x : A. \forall \). So we indeed have \( \lambda x.x \) in \( \forall_{A \to} a \), when checking that the constructor set is satisfied by the top type. We can easily prove, using similar reasoning as in Section 4 above, that \( \lambda x.x \) has type \( \forall A : \forall a : A. a =^A a \).

5 Semantics of Types

To define a semantics for types, we need a few preliminary definitions. We will work with set-theoretic partial functions for the semantics of higher-kinded types. An application of such a function is undefined if the argument is not in the domain of the partial function. (As standard in set theory, such functions are themselves sets.) We consider any meta-level expressions, including formulas, which contain undefined subexpressions to be undefined themselves. In lemmas and theorems, if we affirm formulas involving possibly undefined expressions, we are implicitly affirming all those expressions are defined. We write \( A \to B \) for the set of meta-level total functions from set \( A \) to set \( B \); that is, total functional subsets of \( A \times B \). We write \( (x \in A \to b) \) for the (meta-level) function mapping input \( x \) in the set \( A \) to \( b \).

For our semantics, we prove results about closed terms only, though for the semantics of the lifting operation we will have to consider open terms. Let \( \mathcal{L} \) be the set of closed lambda abstractions (i.e., terms of the form \( \lambda x.t \) with no free variables), and let \( \mathcal{N} \subseteq \mathcal{L} \) be the set of closed normal-form terms. We will write \( \to \) for (full) \( \beta \)-reduction. We also write \( \equiv_{\beta} \) for standard \( \beta \)-equivalence restricted to closed terms, and \( [t]_{\beta} \) for the set \( \{ t' \mid t \equiv_{\beta} t' \} \). The latter operation is extended to sets \( S \) of terms by writing \( [S]_{\beta} \) for \( \{ [t]_{\beta} \mid t \in S \} \). In a few
places we write $nf(t)$ for the (unique) normal form of term $t$; this is undefined if $t$ has no
normal form. We write $\Omega$ for an arbitrary term without normal form, like $\lambda x.x \ (\lambda x.x)$.

**Definition 1 (Reducibility candidates)**

\[ R := \{ [S]_c \mid S \subseteq L \} \]

A reducibility candidate (element of $R$) is a set of $\beta$-equivalence classes of $\lambda$-abstractions.
We will use this definition to develop as technically light a semantics as possible, while
still being sufficient to show logical consistency (Corollary 14 below). Further adaptation
would be necessary to show normalization, but this is not needed for our consistency proof.
One exception is that we will need to reason about normalization for proving soundness of
lifting. Throughout the development we will make use of a choice function $\zeta$. Given any
set $E$ of terms, $\zeta$ returns a $\lambda$-abstraction if $E$ contains one, and is undefined otherwise.

**Lemma 2**

If $E = [\lambda x.t]_c$, then $[\zeta(E)]_c = E$

**Lemma 3 (R is a complete lattice)**

The set $R$ ordered by subset forms a complete lattice, with greatest element $[L]_c$ and
greatest lower bound of a nonempty set of elements given by intersection.

**Lemma 4**

$[N]_c \in R$, and $\emptyset \in R$.

Figure 9 defines our semantics for types and kinds, by mutual structural recursion. The
semantic functions take arguments $\sigma$ and $\rho$, in addition to the type or kind to interpret.
We require that $\sigma$ maps term variables to terms, and $\rho$ maps type variables to sets. The
interpretations of types and kinds are then also sets. We will get more precise descriptions
of the domains and codomains of the semantic functions later. The interpretation of $\nu$-types
uses the notation $F^n(a)$ for (meta-level) iteration of the function $F$ $n$ times on $a$: $F(F(...F(a)))$.
The operation $\cap_{\kappa,\sigma,\rho}$ used in the semantics of $\nu$-types, and the value
$\top_{\kappa,\sigma,\rho}$ used in the semantics of $\forall$, are defined in Figure 10. The meaning of a type can be
empty, and so in interpreting $\forall x : T. T'$ we must take the intersection using $\cap$, which returns
the top element of $R$ if the interpretation of $T$ is empty. The meaning of a kind cannot be
empty, however, so we do not need to worry about this situation when interpreting $\forall X : \kappa. T$.
For the semantics of $\Pi x : T. \kappa$, if $[T]_{\sigma,\rho} \notin R$, then the meaning of the $\Pi$-kind is undefined.

An important principle in the definition of this semantics is that if the meaning of a type
is defined, then it satisfies the semantic counterparts of the conversion rules in Figure 5. So
loosely, if $T \triangleright T'$ and $[T]$ is defined, then $[T] = [T']$ just based on the definition of $[T]$
(not any auxiliary information). This greatly simplifies the semantic connection between
conversion and typing for the proof of semantic soundness (Theorem 13 below).

Figure 11 defines a semantic lifting function to lift terms to semantic functions at the
(set-theoretic) level where they are in the interpretations of kinds. We do not need to carry
the valuations $\sigma$ and $\rho$ through the definition, since we have restricted lifting types to be
simple types over $\ast$. A different kind of valuation $\theta$ is used, which maps term variables to
pairs $(L,S)$ where $L$ is a lifting type and $S$ is a set. If we included types $\Pi x : T.L$ as lifting
types, then we would need to make use of $\sigma$ and $\rho$ in the definitions in Figure 11.
Fig. 9. Semantics for types and kinds (see also Figures 10 and 11)

\[ \forall X \subseteq_{\sigma, \rho} Y \Longleftrightarrow X \subseteq Y \]

\[ \forall X \subseteq_{\Pi \alpha: T \cdot K \cdot \sigma, \rho} Y \Longleftrightarrow \forall E \in [T]_{\sigma, \rho}. X(E) \subseteq_{\sigma, \rho} Y(E) \]

\[ \forall \beta, \lambda X: K \cdot \text{Type}, \tau \subseteq_{\Pi \alpha: K \cdot \alpha \cdot \sigma, \rho} \beta \Longleftrightarrow \forall \beta, \lambda X: K \cdot \text{Type}, \tau \subseteq_{\Pi \alpha: K \cdot \alpha \cdot \sigma, \rho} \beta \]

\[ \text{T}_\ast = \{ \beta \}_{\sigma, \rho} \]

\[ \text{T}_\ast \sigma, \rho = \text{T}_\ast \]

\[ \text{T}_{\Pi \alpha: T \cdot K \cdot \sigma, \rho} = \{ E \in [T]_{\sigma, \rho} \mapsto T_{\sigma, \rho} \} \]

\[ \text{T}_{\Pi \alpha: K \cdot \alpha \cdot \sigma, \rho} = \{ S \in [K]_{\sigma, \rho} \mapsto K_{\sigma, \rho} \} \]

\[ \cap_x X = \begin{cases} \cap_x X, & \text{if } X \neq \emptyset \\ \text{T}_\ast, & \text{otherwise} \end{cases} \]

\[ \cap_x \alpha \cdot \sigma, \rho X = \cap_x X \]

\[ \cap_{\Pi \alpha: T \cdot K \cdot \sigma, \rho} X = \begin{cases} \cap_{\Pi \alpha: T \cdot K \cdot \sigma, \rho} X, & \text{if } X \neq \emptyset \\ \text{T}_\ast, & \text{otherwise} \end{cases} \]

\[ \cap_{\Pi \alpha: K \cdot \alpha \cdot \sigma, \rho} X = \begin{cases} \cap_{\Pi \alpha: K \cdot \alpha \cdot \sigma, \rho} X, & \text{if } X \neq \emptyset \\ \text{T}_\ast, & \text{otherwise} \end{cases} \]

\[ \text{Fig. 10. Pointwise-extended lattice operations} \]
Journal of Functional Programming

\[
\begin{align*}
\langle\lambda x. t\rangle^L_\theta &\quad=\quad S \in \llbracket \text{lift}(L)\rrbracket_{\theta,0} \mapsto \langle\langle t\rangle\rangle^L_{\theta [\theta \mapsto (S,L)]} \\
\langle x \bar{t}\rangle^L_\theta &\quad=\quad S \langle\langle t\rangle\rangle^{L_0}_{\theta} \cdots \langle\langle (\bar{t}_n)\rangle\rangle^{L_0}_{\theta}, \\
&\quad\text{if } \theta(x) = (S,L \rightarrow L) \text{ with } |L| = n = |\bar{t}|.
\end{align*}
\]

Fig. 11. Semantic lifting \(\langle\langle \rangle\rangle\)

5.1 About the conversion relation

Most type theories are defined using a congruence relation on types which is then shown to be algorithmic by proving its confluence and normalization. For CDLE – and, it seems likely, any system combining dependent and recursive types – the situation is somewhat more complicated, as indicated by the following theorem:

Theorem 5
There is no recursively enumerable convertibility relation between types in context which is sound and complete with respect to equality of interpretations.

Proof
We can reduce extensional equivalence of primitive recursive numeric functions to this problem. That relation is not r. e., since if it were, it would be decidable (inequivalence is obviously r.e.), and it is known not to be so. Suppose \(f\) and \(g\) have type \(\mathbb{N} \rightarrow \mathbb{N}\), and consider the following two types, where \(S\) denotes successor for Church-encoded numerals as above:

\[
\begin{align*}
\forall P: \mathbb{N} \rightarrow \ast, \forall X: \mathbb{N} \rightarrow \ast. \forall n: \mathbb{N}. P(f\ n) \rightarrow X(S\ n) \\
\forall P: \mathbb{N} \rightarrow \ast, \forall X: \mathbb{N} \rightarrow \ast. \forall n: \mathbb{N}. P(g\ n) \rightarrow X(S\ n)
\end{align*}
\]

These types have the same interpretation (with empty functions for \(\sigma\) and \(\rho\)) iff \(f\) and \(g\) return the same values for all inputs \(n: \mathbb{N}\).

So CDLE must be defined using a particular incomplete conversion relation. Further use of the theory will be required to see if further (semantically justified) principles need to be added for practical use. Additional analysis of this relation, such as studying decidability or complete formulations for subrelations, must remain to future work.

5.2 Reasoning about lifting

To prove soundness of the conversion and kinding rules for lift types \(\uparrow_L t\), we need some intricate and interesting reasoning, summarized in the following lemmas. Several of these can be viewed as semantic lemmas about simple typing. To justify the main conversion axiom about lifting, we have this lemma:

Lemma 6
Suppose \(\text{lift}_{L_\theta}(t)\) is defined, and suppose that \(\theta(x) = (S,L)\) holds for some \(S \in \llbracket \text{lift}(L)\rrbracket_{\theta,0}\) iff \(\nu(x) = L\). Suppose also that \(nf(t)\) is defined, and \(FV(t) \subseteq \text{dom}(\theta)\). Then \(\langle\langle t\rangle\rangle^L_{\theta,0,\theta} = \llbracket \text{lift}_{L_\theta}(t)\rrbracket_{\theta,\rho},\) where \(\rho'(x) = S\) iff \(\theta(x) = (S,L')\) for some \(L'\).

To reviewers: detailed proofs for this and the other results listed below may be found in a supplementary document submitted along with this manuscript.
The main lemma needed to justify kinding of lift types is the following, where we first introduce a definition relating valuations \( \theta \) used in semantic lifting (Figure 11) and the valuations \( \sigma \) mapping term variables to terms.

**Definition 7 ((\( \theta, R \))-constrained)**

Suppose \( \theta \) is a given valuation of the sort used in Figure 11, and \( R \in \mathcal{R} \) is also given. Then \( \sigma \) is called \((\theta, R)\)-constrained iff the following holds: \( \sigma(x) \in \\llbracket L \rrbracket_{\theta,(X \rightarrow \rightarrow R)} \) iff \( \theta(x) = (S, L) \).

**Lemma 8 (Main Lifting Lemma)**

Let \( t \) be a possibly open term in normal form, and assume a valuation \( \theta \) with \( \text{dom}(\theta) \supseteq FV(t) \), and such that for all \( x \in \text{dom}(\theta) \), \( \theta(x) = (S, L) \) iff \( S \in \llbracket \text{lift}(L) \rrbracket_{\emptyset, \theta} \). Also, make the following main assumption about \( t \) and \( L \): for all nonempty \( R \in \mathcal{R} \), for all \((\theta, R)\)-constrained \( \sigma \), we have \( [\sigma]_{c,R} \in \llbracket L \rrbracket_{\emptyset,(X \rightarrow \rightarrow R)} \). Then \( \llbracket t \rrbracket_{\emptyset} \in \llbracket \text{lift}(L) \rrbracket_{\emptyset, \theta} \).

This main lemma uses what turns out to be a powerful semantic idea: since the kinding rule for \( \uparrow \) has premise \( \Gamma, X : \star \vdash t : L | \chi \), we know that we have \( \sigma_t \in \llbracket L \rrbracket_{\emptyset,(X \rightarrow \rightarrow R)} \), for any \( R \in \mathcal{R} \). This additional quantification over \( R \) is crucial for getting the proof to go through, and leads to other interesting consequences. First, we get normalization, because we can instantiate \( R \) with \( [\mathcal{V}]_{c,R} \) (the set of closed normalizing terms):

**Lemma 9**

Suppose that for all \( R \in \mathcal{R} \), we have \( [t]_{c,R} \in \llbracket L \rrbracket_{\emptyset,(X \rightarrow \rightarrow R)} \). Then \( t \) is normalizing.

Next we have to note two lemmas, easily proved by induction on the lifting type \( L \) in question.

**Lemma 10**

Let \( \rho = [X \rightarrow R] \), where \( R \in \mathcal{R} \) is nonempty. Then \( \llbracket L | \chi \rrbracket_{\emptyset, \rho} \) is nonempty.

**Lemma 11**

Suppose \( [t_1]_{c,R} \not\in \llbracket L_1 \rrbracket_{\emptyset, \rho} \) where \( \rho = [X \rightarrow R] \) for some nonempty \( R \in \mathcal{R} \). Then for any \( L_2 \) there exists a term of the form \( \lambda y . t_2 \) such that \( [\lambda y . t_2]_{c,R} \in \llbracket L_1 \rightarrow L_2 \rrbracket_{\emptyset, \rho} \) but \( [t_1 / y]_{c,R} \not\in \llbracket L_2 \rrbracket_{\emptyset, \rho} \).

With these, we can derive the following strong property about inclusion of interpretations, which is needed for Lemma 8. The proof is interesting enough that it is given here in full.

**Lemma 12 (Trivial semantic subtyping for simple types)**

Suppose that for all nonempty \( R \in \mathcal{R} \), \( \llbracket L | \chi \rrbracket_{\emptyset,(X \rightarrow \rightarrow R)} \subseteq \llbracket L' | \chi \rrbracket_{\emptyset,(X \rightarrow \rightarrow R)} \). Then \( L = L' \).

**Proof**

The proof is by induction on the structure of \( L' \), considering several cases. We will refer to the assumption of the theorem as our **semantic subtyping assumption**. Let \( L \) and \( L' \) be sequences of lifting types with \( |L| = |L'| = n \), for some \( n \).

**Case:** Suppose \( L \) is \( L \rightarrow \star \) and \( L' \) is \( L_1 \rightarrow L_2 \rightarrow L_3 \) for some \( L_1 \) and \( L_3 \). Then we can easily violate our semantic subtyping by instantiating \( R \) with \( [\mathcal{V}]_{c,R} \) and taking \( [\lambda z . \lambda y . \Omega]_{c,R} \) as an element in \( \llbracket L | \chi \rrbracket_{\emptyset,(X \rightarrow \rightarrow \mathcal{V}_c \rho)} \) but not in \( \llbracket L' | \chi \rrbracket_{\emptyset,(X \rightarrow \rightarrow \mathcal{V}_c \rho)} \).
Suppose we are left with the case where $Z \uparrow_0 0 - 0 - FPR$ paper 22 January 2016 20:57

Case: vacuous, by Lemma 10). (using the fact that the quantifications imposed by the semantics of function types are not theorem. In that definition, we write $J$ in our semantic subtyping assumption with $\{L, \rho \in J | L \uparrow \}$. This follows (using also Lemma 10 to instantiate the variables $\bar{L}$ such that $\bar{T} = \rho$). But then by Lemma 11 there is a term $\lambda \bar{x}.x \bar{x}$. We have $\lambda \bar{x}.x \bar{x} \bar{x} \in \bar{L} \uparrow \star \in [K]_{\bar{S}}$, by a simple application of the semantics of function types. But we do not have $\lambda \bar{x}.x \bar{x} \bar{x} \in \bar{L} \uparrow \star \in [K]_{\bar{S}}$. This follows (using also Lemma 10 to instantiate the variables $\bar{x}$) because $E \in \bar{L} \uparrow \star \in [K]_{\bar{S}}$, but we deduced $\bar{E} \not\in \bar{L} \uparrow \star \in [K]_{\bar{S}}$. So we have $\bar{L} \uparrow \star$ as a semantic subtype of $L \uparrow \star$, and we may then apply the IH to conclude that $L \uparrow \star = L \uparrow \star$. This contradicts the assumption we made that those types are different.

6 Soundness for Typing

Figure 12 defines a semantics for typing contexts, for purposes of the following main theorem. In that definition, we write $\sigma \sqcup \{x \mapsto r\} \in [\Gamma]$ to mean $\sigma[x \mapsto t]$ where $x \not\in dom(\sigma)$ (and similarly for $\rho \sqcup \{X \mapsto S\}$). Figure 13 defines $[k | x \Theta]_{\sigma, \rho}$ to be the set of those elements of $[k]_{\sigma, \rho}$ which satisfy the constraints given by $\Theta$ for type variable $X$. These two helper notions are used in stating the main theorem below.

**Theorem 13 (Soundness of typing and kindning)**
If $(\sigma, \rho) \in [\Gamma]$, then

1. If $\Gamma \vdash k : \top$, then $[k]_{\sigma, \rho}$ is defined.
2. If $\Gamma \vdash T : k$, then $[T]_{\sigma, \rho} \in [k]_{\sigma, \rho}$.
3. If $\Gamma \vdash t : T$, then $[\sigma t]_{\rho, \beta} \in [T]_{\sigma, \rho}$ and $[T]_{\sigma, \rho} \in \mathcal{R}$.
4. If $\Gamma \vdash \Theta : \ast$ and $\Theta = t_1, \ldots, t_n \in T_n$, then $[T_1]_{\sigma, \rho} \in \mathcal{R}, \ldots, [T_n]_{\sigma, \rho} \in \mathcal{R}$.
5. If $\Gamma \vdash \Theta$, then $[\Theta]_{\sigma, \rho}$.  

*Journal of Functional Programming*
Stump

6. If $\Gamma \vdash T \triangleright T'$ and $\llbracket T \rrbracket_{\sigma, \rho} \in \llbracket \kappa \rrbracket_{\sigma, \rho}$ for some kind $\kappa$, then $\llbracket T' \rrbracket_{\sigma, \rho} = \llbracket T \rrbracket_{\sigma, \rho}$.

7. Suppose $(X : \kappa) \in \Gamma$, and let $\sigma = \sigma_1 \sqcup \sigma_2$ and $\rho = \rho_1 \sqcup \rho_2 [X \rightarrow S]$. Suppose also that $S \subseteq \kappa, \sigma, \rho, S'$ and $A \subseteq \llbracket \kappa \rrbracket_{\sigma, \rho, 1}$, with $A \neq \emptyset$.

   a) If $\Gamma \vdash T : \kappa'$, $\llbracket \kappa' \rrbracket_{\sigma_1, \rho_1}$ is defined, and $X \in^+ T$, then
   
   i. $\llbracket T \rrbracket_{\sigma, \rho, [X \rightarrow S]} \subseteq \llbracket \kappa' \rrbracket_{\sigma_1, \rho_1} \llbracket T \rrbracket_{\sigma, \rho, [X \rightarrow S']}$
   
   ii. $\cap \cap \llbracket \kappa' \rrbracket_{\sigma_1, \rho_1} \llbracket T \rrbracket_{\sigma, \rho, [X \rightarrow S]} \mid S \in A \subseteq \llbracket \kappa' \rrbracket_{\sigma_1, \rho_1}$

   b) If $\Gamma \vdash T : \kappa'$ and $X \in^- T$, then $\llbracket T \rrbracket_{\sigma, \rho, [X \rightarrow S']} \subseteq \llbracket \kappa' \rrbracket_{\sigma_1, \rho_1} \llbracket T \rrbracket_{\sigma, \rho, [X \rightarrow S]}$.

   c) If $\Gamma \vdash \kappa' \cdot \Box$, $\llbracket \kappa' \rrbracket_{\sigma_1, \rho_1}$ is defined, and $X \in^+ \kappa'$, then
   
   i. $\llbracket \kappa' \rrbracket_{\sigma, \rho} \subseteq \llbracket \kappa' \rrbracket_{\sigma, \rho, [X \rightarrow S']}$
   
   ii. $\cap \cap \llbracket \kappa' \rrbracket_{\sigma_1, \rho_1} \llbracket T \rrbracket_{\sigma, \rho, [X \rightarrow S]} \mid S \in A \subseteq \llbracket \kappa' \rrbracket_{\sigma, \rho, [X \rightarrow S]}$

   d) If $\Gamma \vdash \kappa' \cdot \Box$ and $X \in^- \kappa'$, then $\llbracket \kappa' \rrbracket_{\sigma, \rho, [X \rightarrow S]} \subseteq \llbracket \kappa' \rrbracket_{\sigma, \rho}$.

8. If $\llbracket \kappa \rrbracket_{\sigma, \rho}$ is defined, $\Gamma, X : \kappa : \Theta : \ast, \llbracket \Theta \rrbracket_{\sigma, \rho, [X \rightarrow T \ast]}$, and $\text{Ctors}_X \Theta$, then $(\llbracket \kappa \rrbracket_{X \Theta})_{\sigma, \rho} \subseteq \llbracket \kappa, \sigma, \rho \cup \cap, \ast \rangle$ is a complete lattice.

These parts must be proved by mutual induction on the structure of the assumed derivation in each part. Parts (1), (2), and (3) of Theorem 13 are statements that the main judgements of CDLE – superkinding, kinding, and typing, respectively – are sound with respect to our semantics. Parts (4) and (5) express soundness of two helper judgements dealing with constructor sets $\Theta$. Parts (5) and (6) express soundness of directed conversion. Parts (7) and (8) are critical for reasoning about $\forall$-types. Parts (7ai) and (7ci) express monotonicity of the semantics for type variables occurring only positively, and parts (7b) and (7d) express antimonotonicity for type variables occurring only negatively. Parts (7aii) and (7cii) are expressing one part of continuity, which is used in establishing that the meaning of a $\forall$-type is indeed a fixed-point of the interpretation of its body; the other ends up following from monotonicity. Part (8) embodies one of the central insights of constructor-constrained recursive types: if a constructor set $\Theta$ satisfies $\text{Ctors}_X \Theta$, then it is preserved not just through the chain of iterates of the interpretation of the body, but also in the limit of that sequence, its greatest lower bound. Without preservation of $\Theta$ in the limit, we cannot show that the meaning of a $\forall$-type is the appropriate fixed point.

**Corollary 14 (Logical consistency)**

There is no derivation of $\vdash t : \forall X : \ast. X$, for any term $t$.

**Proof**

By Theorem 13 part (3) and the semantics of $\forall$-types, if $\vdash t : \forall X : \ast. X$ is derivable, then $t \in \cap \mathcal{P}$. But $\cap \mathcal{P}$ is empty since $\emptyset \in \mathcal{P}$. □

### 7 Cédille: an Implementation of CDLE

I have implemented a type theory based on CDLE, in a tool called Cédille. At first glance, it may seem hard to imagine how to implement CDLE, because of typing rules like direct computation and the introduction rule for dependent intersections, which do not fit well into usual approaches to algorithmic typing. But one insight emerges which helps
us resolve these difficulties. These troublesome features of CDLE are needed solely for
kinded recursive types. Once recursive types are kinded, then it is a relatively simple
matter to unfold them when their inhabitants are eliminated (i.e., applied to arguments). We
need never introduce them, if we are content to use the constructors of the type (from the
constructor sets) as the sole means of constructing inhabitants of recursive types. This rules
out defining alternative versions of operations on lambda-encoded data, such as Rosser’s
alternative definitions of multiplication and exponentiation (though supporting these would
require additional rules in CDLE, to allow typing of non-constructor terms with recursive
types). But this is an acceptable loss to gain the power of higher-order encodings. A final
issue is the need to add typings $t \in T$ to contexts, due to the fact that constructor sets
contain typings of arbitrary terms. This issue is resolved in Cedille by introducing names
for the constructors, which are used in place of those arbitrary terms.

Cedille supports top-level definition of recursive types with the following syntax:

\[ \text{rec } X \text{ params } : \text{ indices } \mid \text{ctors } = T \text{ with defs} \]

Here, \textit{params} and \textit{indices} are telescopes of bindings, the first for parameters fixed for the
whole type definition, and the second for indices, which are inputs to the type constructor
\(X\) which may change in the body \(T\) of the definition. The \textit{ctors} are declarations of con-
structors; this component of the definition is just like \(\Theta\), except that constraints are of the form
\(x : T\), where \(x\) is a constructor name, rather than \(t \in T\). The definitions of the constructors
named in \textit{ctors}, using whichever lambda encoding is being applied, are given in the \textit{defs}.

For example, Figure 14 gives definitions of three standard datatypes: \textit{Nat} is for Church-
encoded natural numbers, \textit{List} is for Parigot-encoded lists, and \textit{Vector} is for Parigot-
encoded vectors (lists indexed by their length). Cedille uses the notation \(\_t\) for an implicit
( erased) argument, and \(\Lambda\) as a term-level binder for implicit inputs. Applications of terms
or types to types are written with the \(\cdot\) operator for parsing reasons. In datatype definitions
only, the special variable \(\textit{self}\) may be used as an implicitly \(\iota\)-abstracted variable referring
to the subject of the typing.

Let us consider how Cedille kinds the definition of \textit{Nat} (Figure 14), for a representative
example. Cedille uses Unicode, so Cedille code largely matches the mathematical syntax
we have already considered. The constructor sets must first be typed, assuming the kinding
\(\textit{Nat } : \star\). Next, the body is kinded, assuming that \(\textit{self}\) has the recursive type (applied to
any indices). So in this case, \(\textit{self}\) is assumed to have type \textit{Nat} when kinding the body.

Finally, each constructor definition (the equations following the \textit{with} keyword) must be
typed. Cedille types a definition \(c = t\) by checking that \(t\) has type \(T\) under the assumption
that the recursively defined type is equal to its body, with the \(\textit{self}\) variable explicitly \(\iota\)-
abstracted. There a variety of other small checks to perform as well (the conditions imposed
by \(\textit{ctors}_X\), the starting condition for kinding using the top type \(\mathcal{U}_\kappa\), and a few others).

Cedille implements local type inference to cut down on the number of annotations
required in terms (Pierce & Turner, 2000). We are either checking a term against a type
or a type against a kind, or else trying to synthesize a type for a term or a kind for a
type. Cedille seeks to instantiate \(\iota\)-types introduced by recursive definitions either when
checking against an introduction form (an implicit or explicit \(\lambda\)-abstraction), or when a
type is synthesized for the head of an application. The former is intended just for typing
constructor definitions, while the latter is for use there as well as when terms of recursive
type are eliminated. This simple scheme appears sufficient so far to avoid any explicit reasoning about dependent intersections on the part of the user.

A final note is that Cedille implements an algorithmic conversion relation based on normalizing term and type expressions. While this is not strictly speaking justified by Theorem 13 above, I conjecture that the proof may, with some effort, be adapted to show not just consistency but normalization. I have not invested this effort so far, for the following reason. Consistency is the crucial property for a type theory, as it tells us that we may safely avoid reducing some terms, and still know that they would reduce to canonical values. Normalization is nice in theory, but in practice the enormous computational complexity of functions which can be written in type theory means that there are terms which will cause type checking to run so long as to be practically indistinguishable from nontermination. So any type theory that truly requires a bound on the time required to check terms will have to do more than just prove normalization (and such theories have, of course, been developed; e.g., (Hofmann, 2000)).
Cedille is implemented in around 2500 lines of Agda (not counting an automatically generated parser). Agda is a dependently typed programming language under development (in its Agda 2 form) for around a decade. The main implementation was done by Ulf Norell (Norell, 2007), with subsequent additions from other researchers. Agda has an active user community consisting primarily of researchers. The language provides many sophisticated features for dependently typed programming, including in particular quite good term- and type-inference, and an elegant approach to coding using recursive equations, similar to Haskell. While a formal soundness proof remains to be developed, Agda seeks to enforce structural termination of all functions, in order to achieve logical soundness under the Curry-Howard isomorphism. Users can disable this check for individual functions, as I found necessary for a few functions in the implementation of Cedille. Cedille is based on the Iowa Agda Library, an alternative standard library I am developing, currently at a little under 5000 lines of code (see my forthcoming book (Stump, 2016)). In this section, I would like to highlight a few dependently typed programming idioms I found quite helpful for implementing Cedille.

Certainly there are many theorems one might wish to prove about various aspects of the implementation of a type theory (cf. (Chapman, 2009)). I judged that for my “engineering budget”, attempting to prove deep theorems would not be a good use of effort. Occasionally, however, an easy theorem was worth proving. For example, the code includes some basic functions which disassemble and reassemble type-level abstractions (definitions elided):

\[
\text{decompose-tpapps : type } \to \text{ type } \times \text{ L tty} \\
\text{recompose-tpapps : type } \times \text{ L tty } \to \text{ type}
\]

It is a little tricky to make sure these match up correctly, and so it seemed worthwhile to prove this easy theorem about these operations:

\[
\text{dere-tpapps : } \forall (t : \text{ type}) \to \text{ recompose-tpapps (decompose-tpapps t) } \equiv t
\]

This is certainly a nice use of dependently typed programming with the Curry-Howard isomorphism in Agda. But I got even more benefit out of other dependently typed programming idioms in writing Cedille, as will be discussed next.

### 8.1 Simple polytypic programming with universes

Cedille’s parser is generated by a tool called gratr, which I have been developing for several years. The parsers gratr generates build abstract syntax trees which are inhabitants of simple inductive datatypes which gratr automatically defines in Agda. For Cedille, these include datatypes term, type, liftingType, and kind, for the corresponding syntactic categories of CDLE. In the course of implementing a tool like Cedille, there are several functions one must write which traverse through all these different syntax trees. For example, several places in the implementation require a test for freeness of a variable in one or the other variety of expression. Thanks to Agda’s rather amazing support for term inference, this can be done polytypically. First, one declares a very simple universe, with expression descriptors for the different abstract syntax types:
Stump

data exprd : Set where
  TERM : exprd
  TYPE : exprd
  KIND : exprd
  LIFTINGTYPE : exprd

Next, following the standard idiom for programming with universes, one defines a decoding function:

\[ \_ ] : \text{exprd} \rightarrow \text{Set} \]
\[ [ \text{TERM} ] = \text{term} \]
\[ [ \text{TYPE} ] = \text{type} \]
\[ [ \text{KIND} ] = \text{kind} \]
\[ [ \text{LIFTINGTYPE} ] = \text{liftingType} \]

Then a polytypic version of the \text{is-free-in} function can be defined, which works for any variety of expression (the \text{is-free-e} argument just tells the function whether to search for the given variable in erased positions in expressions, or to skip those positions):

\text{is-free-in} : \{ \text{ed : exprd} \} \rightarrow \text{is-free-e} \rightarrow \text{var} \rightarrow [ \text{ed} ] \rightarrow \text{B}

Remarkably, Agda’s term inference is able to deduce \text{ed} from the Agda type \[ [ \text{ed} ] \] of an argument to \text{is-free-in}. While it is true that the decoding function in this case is extremely simple, it is still impressive that Agda can do this. This allows the programmer to call \text{is-free-in} elsewhere in the code, without specifying the (expression descriptor for the) Agda type of the expression in which \text{is-free-in} is to search for the variable. So instead of writing \text{is-free-in-term} \text{e x t} when \text{t} is a term, and \text{is-free-in-type} \text{e x t} when \text{t} is a type, one just writes \text{is-free-in} \text{e x t} in both cases. This reduces clutter in code that calls \text{is-free-in}. It also creates opportunities for more abstract code, because sometimes a function which calls \text{is-free-in} can be written generically in the expression descriptor. This use of universes for simple polytypic programming is quite rudimentary compared to others in the literature (e.g., (Dagand & McBride, 2014)), but it is also very easy to apply in Agda.

8.2 Type-level computation and local type inference

As mentioned above, Cedille implements a form of local type inference, to cut down on annotations without implementing a full-fledged constraint-based type-inference algorithm. Without dependent types, one would likely write two separate functions: one to check a term against a type, and another to synthesize a type from a term (and similarly for types and kinds). But with dependent types in Agda, we can combine these two functions into one, by computing the return type from the input for the type. If there is no input (nothing, in a maybe type), then check-term should return (maybe) a type. If there is a type (just tp), then check-term will not return anything. This can be expressed with the following simple type-level function in Agda:

\text{check-ret} : \forall \{ \text{A : Set} \} \rightarrow \text{maybe A} \rightarrow \text{Set}
check-ret(A) nothing = maybe A
check-ret (just _) = ⊤
Journal of Functional Programming

Eq ≜ Π A : ⋆ . A → A → ⋆ =

refl ≜ ∀ A : ⋆ . ∀ a : A . Eq · A a a =
    λ A . λ a . λ P . λ u . u .

Fig. 15. Leibniz equality

This then enables the typing of a single function which can be used both for checking terms against types or synthesizing types from terms (\(\text{spanM}\) is a monad used for producing information about the typing, including error messages):

\[
\text{check-term} : \text{ctxt} → \text{term} → (m : \text{maybe type}) → \text{spanM} (\text{check-ret m})
\]

The advantage of this combination is to eliminate some code duplication for operations that take place regardless of whether one is checking or synthesizing. If we had two separate functions, such code would likely appear twice. But with a single function, we can perform those operations before taking specific actions depending on whether we are checking or synthesizing a type.

9 Basic Examples

Now let us consider some examples demonstrating the features of CDLE, as implemented in Cedille.

9.1 Inductive reasoning about Church-encoded numbers

First, let us show that we can indeed perform dependent eliminations with Church-encoded numbers, by proving a basic inductive fact about addition. We can define Leibniz equality in the usual way, as shown in Figure 15. The statements shown in the figure are of the form \(x \leftarrow e = e'\), for checking expression \(e'\) against classifier (type or kind) \(e\), and then adding a definition of \(x\) to equal \(e'\) to the global context. So here we define the type Eq for Leibniz equality in a standard way, and then give an inhabitant refl for reflexive equalities. As noted above, more complex forms of equality can also be defined using constructor-constrained recursive types, but this is sufficient here.

We can write the following very basic inductive proof about addition, showing that \(x + 0 = x\) for all \(x\):

\[
\text{plusZ} ≜ Π x : \text{Nat} . \text{Eq} · \text{Nat} \ (\text{plus x Z}) x =
    λ x . x · (λ n : \text{Nat} . \text{Eq} · \text{Nat} \ (\text{plus n Z}) n) \ (λ n . λ u . λ P . λ v . u · (λ x : \text{Nat} . P (S x)) v) \ (\text{refl} · \text{Nat}-Z) .
\]

As is well known, this theorem does not hold simply by reducing add \(x\ Z\), because plus iterates on \(x\). The term we have given as the definition for add-zero has an induction on \(x\) matching this iteration. The induction is carried out by a dependent elimination, where \(x\) is applied, in the second line, to the predicate to be proved. The third line of the code gives the step case, where \(u\) is the proof of \(P\ (\text{add n Z})\), for an arbitrary predicate \(P\) postulated by Leibniz equality, and the return value is then the proof of \(P \ n\). The fourth line gives the base case, which follows trivially using conversion.
Figure 16 defines the Bool datatype using a constructor-constrained recursive type to support dependent eliminations on booleans. The figure also gives standard impredicative definitions for the types True and False.

Using these definitions, we may then write the following proof deriving a contradiction from an assumption that tt (boolean true) equals ff, where the notion of equality is again Leibniz equality:

\[
\text{tt-not-equal-ff} \iff \text{Eq} \cdot \text{Bool} tt ff \rightarrow \text{False} = \\
\lambda u . u \cdot (\lambda b . \lambda v . \\
\uparrow X . b \cdot (\lambda b . X) : (* \rightarrow * \rightarrow *) \cdot \text{True} \cdot \text{False}) \\
\text{triv}.
\]

Note that this fact is not provable for Church-encoded booleans in Coq, for instance (Werner, 1992). Here, we instantiate the variable \( P \) from the Leibniz equality with a predicate which uses lifting (the \( \uparrow \) expression) to compute the type False from boolean ff and True from tt. This allows us to cast \( \text{triv} \) from type True to type False. Using large eliminations is the standard way to prove this fact with primitive datatypes, but large eliminations are not available for lambda encodings in other theories. Lifting in CDLE makes this possible. In the Cedille implementation, we must explicitly introduce the type variable \( X \) (immediately following the \( \uparrow \) sign), which the term being lifted will use to indicate the positions in the type which are to be lifted to the kind \( * \).

### 9.3 Higher-order encoding of System F types

Let us see how CDLE allows large eliminations with higher-order encodings of datatypes. We would like to represent the types of System F (constructed by universal quantification and function-space formation from type variables), using a higher-order encoding. So we do not want to encode the universally bound variables as de Bruijn indices, for example. Rather, we will use CDLE’s variables to represent these System F type variables. We can declare this type \( \text{tp} \), of kind \( * \) to represent System F types:

\[
\text{tp} \iff * = \forall X : * . (X \rightarrow X \rightarrow X) \rightarrow ((X \rightarrow X) \rightarrow X) \rightarrow X.
\]

The type says that for all types \( X \), a \( \text{tp} \) can take in a function of type \( X \rightarrow X \rightarrow X \) and also one of type \( (X \rightarrow X) \rightarrow X \) and return a value of type \( X \). The first function is the one
to use if the \( tp \) is representing an arrow type (and then the values computed for the domain and range types will be supplied as the two arguments of type \( X \)). The second function takes in a \( X \to X \) function and returns a value of type \( X \). Here we see the higher-order aspect of the encoding. Due to the negative occurrence of \( X \) in the domain type \( X \to X \) of this type, this would not be allowed as part of an inductive datatype definition in Coq or Agda, though it could be defined in the pure \( \lambda \)-calculus fragment of Coq. But Coq does not have anything like the lifting operation of CDLE, and so one could not write the following type-level function, which interprets a \( tp \) as the corresponding actual type of CDLE.

\[
\text{interp} \leftarrow \text{tp} \to \star = \\
\lambda\ T\ .\ \uparrow\ Y\ .\ \left( (\star \to \star \to \star) \to ((\star \to \star) \to \star) \to \star \right) \cdot \\
\left( \lambda\ A\ .\ \lambda\ B\ .\ A \to B \right) \cdot \\
\left( \lambda\ F\ .\ \forall\ C:\ \star\ .\ F \cdot C \right)
\]

This definition lifts the \( tp \) to the type level, and then applies it to functions which compute either the arrow type or the universally quantified type. In the latter case, the higher-order encoding presents us with \( F \) of type \( \star \to \star \), which maps any input type to the interpretation of the encoded body of the universal type. So we just introduce a universally quantified \( C \) and apply \( F \) to that, to compute the interpretation.

For example, we may define the type of polymorphic identify functions as an inhabitant of \( tp \):

\[
\text{polyid-t} \leftarrow \text{tp} = \Lambda\ X\ .\ \lambda\ \text{arrow}\ .\ \lambda\ \text{forall}\ .\ \forall\ \left( \lambda\ x\ .\ \text{arrow}\ x\ x \right)\ .
\]

If we interpret this value using our \text{interp} function, Cedille tells us we get

\[
\forall\ C : \star\ .\ (C \to C)
\]

To demonstrate the point that we can eliminate data at multiple levels of the type theory, let us also define a function to compute the size (as a natural number) of a \( tp \):

\[
\text{size} \leftarrow \text{tp} \to \text{Nat} = \\
\lambda\ T\ :\ \text{tp}\ .
\]

\[
T \cdot \text{Nat} \\
\left( \lambda\ m\ .\ \lambda\ n\ .\ S\ (\text{plus}\ n\ m) \right) \\
\left( \lambda\ s\ .\ S\ (s\ \text{one}) \right)
\]

Cedille reports that normalizing \text{size} \text{polyid-t} results in Church-encoded four:

\[
\lambda\ s\ .\ \lambda\ z\ .\ (s\ (s\ (s\ (s\ z))))
\]

It is important to note that in this example, we are using lifting only. Constructor-constrained recursive types require positivity, which would not hold here. Even though we do not get a dependent elimination principle for a datatype like \( tp \), we still gain extra expressive power in CDLE over other impredicative type theories like that of Coq, due to CDLE’s lifting operation.
10 Formatted printing with local definitions

Let us now consider a more complex case of large eliminations with higher-order encodings: adding local definitions to format specifiers for formatted printing as with printf. Typing printf is now a standard and quite appealing example of dependently typed programming, introduced by Augustsson (Augustsson, 1998). Here, we will allow format specifiers – for which we will use a dedicated datatype, not a format string – to contain two types of let-declarations. \texttt{flet} \(x\ \ y\) will specify that the arguments required by \(x\) should be input to the call to \texttt{format}, and then the resulting string which is computed for \(x\) will be substituted into \(y\). More dynamic is \texttt{fletd} \(x\ \ y\), which just substitutes \(x\) into \(y\), and thus could duplicate requirements for arguments (leading to additional arguments to the call to \texttt{format}). We will print lists of booleans rather than lists of characters, to avoid dependence on a primitive type of characters.

10.1 Agda implementation

Figures 17 and 18 give Agda code for this example (based on the Iowa Agda Library), which we will walk through briefly, in the hopes that it will orient readers familiar with Agda or Haskell, for the subsequent Cedille implementation (Section 10.2). The first thing to note is that we must disable Agda’s positivity checker to use a higher-order encoding, thus sacrificing the termination property which Agda seeks to guarantee. No such sacrifice will be needed with the Cedille version. Next, we have a datatype \texttt{formatti} which will describe the argument requirements of format specifiers. A format specifier can require an argument (\texttt{iarg}), no argument (\texttt{inone}), or appended requirements (\texttt{iapp}), or requirements governed by a dynamic \texttt{let} (\texttt{ilet}). The type \texttt{formati} is the type for the actual format specifiers. The interesting cases are for \texttt{flet} and \texttt{fletd}, where we use higher-order encoding. In the static case (\texttt{flet}), we have a function from inputs with argument requirement \texttt{inone} to outputs with requirement \texttt{b}, and in the dynamic case, the requirement goes from \(a\) to \(b\ \ a\). The types of the inputs to these constructors use the \texttt{formati} in negative positions, and hence would be disallowed by Agda without the initial pragma disabling the positivity check.

The function \texttt{format-t} (Figure 18) computes the type for \texttt{format} from an argument requirement (of type \texttt{formati}), while \texttt{format} itself (or rather, the helper function \texttt{formath}) is defined by recursion on the format specifier (of type \texttt{formati}). The \texttt{formath} function uses a continuation so that interpretation of the format directive can take place before any input arguments are required (by an \texttt{farg} format specifier).

For a test case, we can define

\begin{verbatim}
  testi : formatti
  testi = ilet (iapp iarg (iapp inone inone))
         (λ x → iapp x (iapp inone x))

test : formati testi
test = fletd (flet farg (λ j → fapp j j))
       (λ i → fapp i (fapp (flit tt) i))
\end{verbatim}
module format-ilet where

open import lib

data formatti : Set where
  iarg : formatti
  inone : formatti
  iapp : formatti → formatti → formatti
  ilet : formatti →
    (formatti → formatti) → formatti

bitstr : Set
bitstr = L B

data formati : formatti → Set where
  farg : formati iarg
  fapp : {a b : formati} →
    formati a → formati b →
    formati (iapp a b)
  flet : {a b : formati} → formati a →
    (formati inone → formati b) →
    formati (iapp a b)
  fletd : {a : formati}
    {b : formati → formati} →
    formati a →
    (formati a → formati (b a)) →
    formati (ilet a b)
  fbitstr : bitstr → formati inone

flit : B → formati inone
flit b = fbitstr [ b ]

Fig. 17. Datatype definitions for format with local definitions, in Agda

The format specifier test says that we want to print a string consisting of i followed
by a boolean literal tt (flit tt), and then i again, where i is dynamically defined to be
the static definition flet farg (λ j → fapp j j). This requests one argument to be
named j, and then produces j appended to j. Agda’s normalizer reports that as expected,
format test normalizes to
λ x x₁ → x :: x :: tt :: x₁ :: x₁ :: []

10.2 Cedille implementation

Let us now implement this example in Cedille. It is worth emphasizing that no modification
to CDLE is required (whereas we had to disable positivity checking for the example to type
check in Agda). We should also note that similarly to the example of representing the types
of System F (Section 9.3), we will use higher-order encodings that prevent us from using
constructor-constrained recursive types. Lifting, however, is still available, and is sufficient
Stump

\[ \text{format-th}: \text{formatti} \to \text{Set} \to \text{Set} \]
\[ \text{format-th iarg } r = \mathbb{B} \to r \]
\[ \text{format-th inone } r = r \]
\[ \text{format-th (iapp } i i') r = \text{format-th } i \text{ (format-th } i' \text{ r)} \]
\[ \text{format-th (ilet } i i') r = \text{format-th } (i' \text{ i)} r \]

\[ \text{format-t}: \text{formatti} \to \text{Set} \]
\[ \text{format-t } i = \text{format-th } i \text{ bitstr} \]

\[ \text{formath}: \{i: \text{formatti}\} \to \text{formati } i \to \]
\[ \{A: \text{Set}\} \to (\text{bitstr} \to A) \to \text{format-th } i A \]
\[ \text{formath farg } f \ x = f \ [\ x \] \]
\[ \text{formath (fapp } i i') f = \]
\[ \text{formath } i \left(\lambda s \to \text{formath } i' \left(\lambda s' \to f \ (s \ ++ \ s')\right)\right) \]
\[ \text{formath (flet } i i') f = \]
\[ \text{formath } i \left(\lambda s \to \text{formath } (i' \ (f\text{bitstr } s)) f\right) \]
\[ \text{formath (fletd } i i') f = \text{formath } (i' \ i) f \]
\[ \text{formath (fbitstr } s) f = f \ s \]

\[ \text{format}: \{i: \text{formatti}\} \to \text{formati } i \to \text{format-t } i \]
\[ \text{format } i = \text{formath } i \left(\lambda x \to x\right) \]

Fig. 18. Formatted printing with local definitions, in Agda

for this example. First, we must declare the type \text{formatti} for argument requirements. We break this into two parts: a type-level function \text{formatto}, and then the universal type \text{formatti}:

\[ \text{formatto} \leftarrow \ast \to \ast = \]
\[ \lambda X : \ast . \ X \to X \to (X \to X \to X) \to (X \to (X \to X) \to X) \to X . \]
\[ \text{formatti} \leftarrow \ast = \forall X : \ast . \ \text{formatto} \cdot X . \]

We can define abbreviations for the constructors of this type, the last of which is the most interesting, since it is there that higher-order encoding shows up:

\[ \text{iarg} \leftarrow \text{formatti} = \Lambda X . \ \lambda a . \ \lambda n . \ \lambda p . \ \lambda l . \ a . \]
\[ \text{inone} \leftarrow \text{formatti} = \Lambda X . \ \lambda a . \ \lambda n . \ \lambda p . \ \lambda l . \ n . \]
\[ \text{iapp} \leftarrow \text{formatti} \to \text{formatti} \to \text{formatti} = \]
\[ \lambda x . \ \lambda y . \]
\[ \Lambda X . \ \lambda a . \ \lambda n . \ \lambda p . \ \lambda l . \]
\[ p \ (x \cdot X a n p l) \ (y \cdot X a n p l) . \]
\[ \text{ilet} \leftarrow \text{formatti} \to (\forall X : \ast . \ X \to \text{formatto} \cdot X) \to \text{formatti} = \]
\[ \lambda u . \ \lambda f . \]
\[ \Lambda X . \ \lambda a . \ \lambda n . \ \lambda p . \ \lambda l . \]
\[ l \ (u \cdot X a n p l) \ (\lambda x . f \cdot X a n p l) . \]

The argument \( f \) to \text{ilet} takes in an \( X \) and returns a \( \text{formatto} \cdot X \), for any type \( X \). This can be viewed as saying that \( f \) is a member of an extension of the datatype \( \text{formatti} \) with a new constructor (since \( f \) requires a value of type \( X \) for this constructor).

We elide a few easy definitions (Church-encoded booleans, an append operation on lists, and the \text{bsingleton} function for creating a singleton list from boolean input). Next
formato \leftarrow (\text{formatti} \to \ast) \to \text{formatti} \to \ast = \\
\lambda X : \text{formatti} \to \ast . \lambda i : \text{formatti} . \\
X \text{iarg} \to \\
(\forall a : \text{formatti} . \forall b : \text{formatti} . \\
X a \to X b \to X (\text{iapp} a b)) \to \\
(\forall a : \text{formatti} . \forall b : \text{formatti} . \\
X a \to (X \text{inone} \to X b) \to X (\text{iapp} a b)) \to \\
(\forall x : \text{formatti} . \forall F : \forall X : \ast . X \to \text{formato} \cdot X . \\
X x \to \\
(X x \to \\
X (X \cdot A X . \lambda a . \lambda n . \lambda p . \lambda l . F \cdot X (x \cdot X a n p l) a n p l)) \to \\
X (\text{ilet} x F)) \to \\
(\text{bitstr} \to X \text{inone}) \to \\
X i . \\
\text{formati} \leftarrow \text{formatti} \to \ast = \\
\lambda i : \text{formatti} . \forall X : \text{formatti} \to \ast . \text{formato} \cdot X i . \\

Fig. 19. The type \text{formati} for format strings

k \leftarrow \square = \ast \to \ast . \\
F_a \leftarrow k = \lambda r : \ast . (\text{Bool} \to r) . \\
F_n \leftarrow k = \lambda r : \ast . r . \\
F_p \leftarrow k \to k \to k = \lambda f : k . \lambda g : k . \lambda r : \ast . f \cdot (g \cdot r) . \\
F_l \leftarrow k \to (k \to k) \to k = \lambda f : k . \lambda g : k \to k . \lambda r : \ast . (g \cdot f \cdot r) . \\

format-th \leftarrow \text{formatti} \to \ast \to \ast = \\
\lambda i : \text{formatti} . \\
\uparrow X . i \cdot (X \to X) : ((\ast \to \ast) \to ((\ast \to \ast) \to (\ast \to \ast))) \to \\
((\ast \to \ast) \to ((\ast \to \ast) \to (\ast \to \ast)) \to (\ast \to \ast)) \to \\
((\ast \to \ast)) \cdot \text{Fa} \cdot \text{Fn} \cdot \text{Fp} \cdot \text{Fl} . \\

Fig. 20. Definition of the helper function computing the type for a call to \text{format} from a format string

comes the type \text{formati} for format specifiers. Again, we break it into two parts, shown in Figure 19.

The type for the dynamic let (beginning on the eighth line in the figure) is the trickiest, since the argument requirement for the body of the let depends on the argument requirement \text{x} for the let’s definiens. But our definition of ilet requires a \text{F} that can be extended with the value for a variable, which enables expression of this dependence. For space reasons, we must omit the definitions of constructors for this type, and turn to the definition of format-th. To make reasoning about this definition more manageable, we pre-define the type-level functions that will be used for the different cases of a formati term. The code is shown in Figure 20. The crucial point, of course, is to use lifting to define the type by higher-order iteration on the input of type formati.

It is convenient to break out the return type for format-th as a separate definition (formatthr), and then we have the code for format-th itself, shown in Figure 21. Instead of recursive calls, the higher-order iteration on \text{a} of type formati presents us with results \text{r} of recursive
Stump

\begin{verbatim}
formatthr \equiv \forall i : \text{formatti} . \forall A : * . (\text{bitstr} \rightarrow A) \rightarrow \text{format-th} i \cdot A .

format \equiv \forall i : \text{formatti} . \forall A : * . (\text{bitstr} \rightarrow A) \rightarrow \text{format-th} i \cdot A .
\end{verbatim}

Fig. 21. Definition of the helper function for |format— |

\begin{verbatim}
format-t \equiv \forall i : \text{formatti} . \forall A : * . (\text{bitstr} \rightarrow A) \rightarrow \text{format-th} i \cdot A .

format \equiv \forall i : \text{formatti} . \forall A : * . (\text{bitstr} \rightarrow A) \rightarrow \text{format-th} i \cdot A .
\end{verbatim}

Fig. 22. The definition of the \text{format} function and its return type calls, in each case. As we are computing a higher-order function (of type \text{formatthr}), these results are themselves functions, which we call with a continuation to obtain the printing function for the part of the format string from which the result was iteratively computed.

The final definition of the \text{format} function and its return type is then the following, where for the outermost continuation we use a function \text{CList} which converts Parigot-encoded to Church-encoded lists. This just makes the output produced by Cedille’s interpreter more readable in this case. The code is in Figure 22. We can use Cedille’s normalizer with the same test as we used for the Agda version, to obtain

\begin{verbatim}
(\lambda b' . \lambda b'' .
  \lambda c . \lambda e .
  (c b' (c b' (c (\lambda a' . \lambda b''' . a') (c b''' (c b''' e))))))
\end{verbatim}

This is indeed a Church-encoded version of the answer we computed with the Agda implementation (at the end of Section 10.1).

In typing the \text{formatth} term of Figure 21, several conversions dealing with lifting are required. These are the last two conversions shown in Figure 5 above. Let us see briefly how these arise. In typing the cases for \text{fapp} and \text{flet}, Cedille must check that the type \text{format-th} (\text{fapp} a b) \cdot A is convertible with

\begin{verbatim}
\text{format-th} a \cdot (\text{format-th} b \cdot A)
\end{verbatim}

The latter type arises from the terms \text{r} \cdot (\text{format-th} b \cdot A) in both cases, while the former type is the one required by the elimination of the format specifier \text{x}. Since lifting introduces new lifting redexes for arguments to a head variable, normalizing the first type
would, without the \( \eta \)-contraction lifting conversion of Figure 5 (the first conversion in the last row of the figure), produce what is essentially an \( \eta \)-expanded version of \( a \) to be lifted.

The last conversion of Figure 5 is needed for the \( \text{fletd} \) case, where Cedille must check that \( \text{format-th} \ (\text{ilet} \ x \ F) \cdot A \) is convertible with the type shown in Figure 23. Again, due to the way lifting produces new lifting redexes, normalization of the first type would result in a lifting of \( F \) being applied to a lifting of \( x \). Those two uses of the lifting operation need to be consolidated at the top level of the term, in order to match the type of Figure 23. This is what the final conversion of Figure 5 does.

### 11 Conclusion and Future Work

This paper has demonstrated that lambda encodings can be the basis for a dependent type theory supporting both induction and large eliminations, via the system CDLE and its implementation Cedille. Induction is enabled by the novel constructor-constrained recursive types \( \nu X : \kappa \mid \Theta . T \), where \( \Theta \) is a set of typing constraints on pure lambda terms which must be shown to hold for a top type \( \forall \kappa \) and then be preserved by the body \( T \). Under some light restrictions on the use of \( \kappa \) in the types in \( \Theta \), these typings hold not just for the elements of the infinite sequence of increasing dependent types one can associate with the \( \nu \)-type, but also for the limit of that sequence, which our semantics defines the meaning of the type to be. Large eliminations are enabled by a lifting construct \( \uparrow Lt \), which lifts simply typed lambda terms to the type level. We gave a rather simple semantics for types in terms of complete lattices, and proved the typing rules of CDLE sound with respect to this semantics. Logical consistency of the system is then a corollary. CDLE does not use a datatype system, and hence one could hope would be less cumbersome for formal meta-theoretic analysis. The most exciting application of CDLE is for dependently typed programming with higher-order encodings. We gave several examples, including the nontrivial one of formatted printing with local definitions.

Programming with higher-order lambda-encodings is a delicate matter (cf. (Washburn & Weirich, 2003) for one illuminating example). Much more exploration of this area is required. It would be interesting, for example, to see how much formalized metatheory one could do using higher-order encodings in Cedille. CDLE has shown that one can have dependent typing for higher-order encodings, via lifting. Induction for such encodings, however, is prevented currently by the positivity requirement for constructor-constrained recursive types. There is a way, however, to drop this requirement. We could modify the type system to express the idea that a type can only depend on a term nominally: the type could never possibly use the term in a lift type. In that case, an erasure semantics can be applied (as done by Fu and Stump (Fu & Stump, 2014)) to prove soundness of the type theory. This would enable us to use recursive types with no positivity requirement, as long as
there was no “true” dependency on data in those types. This might make it possible to make even greater use of higher-order encodings for applications like formalized metatheory.

Acknowledgments. I gratefully acknowledge NSF support of this project under award 1524519. I also deeply thank Madeliene Stump for her support while I was writing this paper. AMDG.

References

Abel, Andreas, & Matthes, Ralph. (2004). Fixed Points of Type Constructors and Primitive Recursion. Pages 190–204 of: Marcinkowski, Jerzy, & Tarlecki, Andrzej (eds), Computer Science Logic (CSL), 18th International Workshop.


Journal of Functional Programming


A Omitted Rules

In this section are listed some straightforward rules omitted from the definition of CDLE in the main text.

A.1 Rules defining judgement $X \in^p T$

Rules defining judgement $X \in^p T$ are in Figure A 1. We write $\check{p}$ for the other polarity besides $p$.

A.2 Additional rules for directed conversion

Figure A 2 gives additional rules for directed conversion. Computation rules were given in Figure 5. The additional rules include reflexivity, transitivity, and then congruence rules equating expressions where the corresponding subexpressions are equal. Passing under a binder extends the context, and passing into the body of a $\nu$-type adds the constructor set to the context (just like the kinding rule for $\nu$-types).

B Some Basic Lemmas

We collect here some basic lemmas, whose proofs are below C.

Lemma 15 (Term substitution and interpretation)

If $t' \equiv_{c,\rho} t\sigma$, then:

- $\llbracket T \rrbracket_{\sigma[x\mapsto t']}_{p,\rho} = \llbracket t / x \rrbracket_{\rho} T_{\sigma,\rho}$
- $\llbracket \kappa \rrbracket_{\sigma[x\mapsto t']}_{p,\rho} = \llbracket t / x \rrbracket_{\rho} \kappa_{\sigma,\rho}$
- $\llbracket \Theta \rrbracket_{\sigma[x\mapsto t']}_{p,\rho} \equiv \llbracket t / x \rrbracket_{\rho} \Theta_{\sigma,\rho}$
- $\subseteq_{\kappa,\sigma[x\mapsto t']}_{p,\rho} \equiv \subseteq_{\sigma[x\mapsto t]}_{\rho} \Theta_{\kappa,\sigma,\rho}$
Corollary 16 (Term equality and interpretation)
If $t =_\beta t'$, then:

- $[[T]]_{\sigma[v\mapsto\tau],\rho} = [[T]]_{\sigma[v\mapsto\tau'],\rho}$
- $[[K]]_{\sigma[v\mapsto\tau],\rho} = [[K]]_{\sigma[v\mapsto\tau'],\rho}$
- $[[\Theta]]_{\sigma[v\mapsto\tau],\rho} = [[\Theta]]_{\sigma[v\mapsto\tau'],\rho}$

Lemma 17 (Strengthening substitutions)
If $x \not\in \text{FV}(T)$, then $[[T]]_{\sigma[v\mapsto\tau],\rho} = [[T]]_{\sigma,\rho}$, $[[K]]_{\sigma[v\mapsto\tau],\rho} = [[K]]_{\sigma,\rho}$, and $[[\Theta]]_{\sigma[v\mapsto\tau],\rho} = [[\Theta]]_{\sigma,\rho}$.

If $X \not\in \text{FV}(T)$, then $[[T]]_{\sigma,\rho[X\mapsto S]} = [[T]]_{\sigma,\rho}$, $[[K]]_{\sigma,\rho[X\mapsto S]} = [[K]]_{\sigma,\rho}$, and $[[\Theta]]_{\sigma,\rho[X\mapsto S]} = [[\Theta]]_{\sigma,\rho}$.

Lemma 18 (Type substitution and interpretation)

- $[[T]]_{\sigma,\rho[X\mapsto T'],\rho} = [[T'/X]]_{\sigma,\rho}$
- $[[K]]_{\sigma,\rho[X\mapsto T'],\rho} = [[T'/X]]_{\sigma,\rho}$
- $[[\Theta]]_{\sigma,\rho[X\mapsto T'],\rho} = [[T'/X]]_{\sigma,\rho}$

Lemma 19 (Equivalence and inclusion)
If $t =_\beta t'$, then if defined, $\sqsubseteq_{\kappa,\sigma[v\mapsto\tau],\rho}$ equals $\sqsubseteq_{\kappa,\sigma[v\mapsto\tau'],\rho}$.

Lemma 20 (Complete lattices at higher kinds)
Suppose that $[[K]]_{\sigma,\rho}$ is defined. Then $([[K]]_{\sigma,\rho}, \sqsubseteq_{\kappa,\sigma,\rho}, \cap)$ forms a complete lattice, with greatest element $\top_{\kappa,\sigma,\rho}$ (see Figure 10).

Lemma 21 (Substitution and greatest lower bounds)
If defined, $\cap_{\kappa,\sigma[v\mapsto\tau'],\rho} = \cap_{(t/\tau)x_{\kappa,\sigma,\rho}}$, where $t' =_\beta \sigma t$. Similarly, if defined, $\cap_{\kappa,\sigma,\rho[X\mapsto T],\rho} = \cap_{(T/X)x_{\kappa,\sigma,\rho}}$.

Lemma 22 (Definedness and strengthening inclusion)
If $[[K]]_{\sigma,\rho}$ is defined, then $\sqsubseteq_{\kappa,\sigma,\rho}$ is equal to $\sqsubseteq_{\kappa,\sigma,\rho'\cap\sigma,\rho'}$, and similarly for $\cap$ and $\top$.

Lemma 23 (Top types)
$[[\top_{\kappa}]]_{\sigma,\rho}$ is defined and equals $\top_{\kappa,\sigma,\rho}$.
C Proofs for Basic Lemmas from the Start of Section 5 and Section B

Proof of Lemma 3
Since every set in $\mathcal{R}$ is the $\beta$-equivalence closure of a subset of $\mathcal{L}$, $[\mathcal{L}]_{\beta}$ is the greatest such set. Now let us prove that the intersection $\cap U$ of a nonempty set $U$ of reducibility candidates is still a reducibility candidate. We must show that there is some $V \subseteq \mathcal{L}$ such that $\cap U = [V]_{\beta}$. Since $U$ is nonempty, let $E \in U$. Define $V$ as $\{\xi(e) \mid e \in \cap U\}$. By definition, $V \subseteq \mathcal{L}$. Note that every $e \in \cap U$ is of the form $[\lambda x.t]_{\beta}$, since every element in $U$ is a reducibility candidate. So we can apply Lemma 2 as follows:
\[
[V]_{\beta} = \{\xi(e) \mid e \in \cap U\}_{\beta} = \{\xi(e) \mid e \in \cap U\} = \{e \mid e \in \cap U\} = \cap U
\]

Proof of Lemma 15
The nontrivial case of the proof by mutual structural induction on $T$ and $\kappa$ is when $T$ is of the form $T_1 t_1$. We have
\[
\begin{align*}
[t/x](T_1 t_1)_{\sigma, \rho} & = \text{ by semantics} \\
[t/x]T_1 t_1 & = \text{ IH} \\
T_1 t_1 & = \text{ assumption} \\
T_1 & = \text{ composition of substitutions} \\
\end{align*}
\]

Proof of Lemma 20
The proof is by induction on the structure of $\kappa$. If $\kappa$ is $\ast$, then the result follows immediately from Lemma 3. We will just consider the case where $\kappa = \Pi x : T. \kappa_1$, since the other case (where $\kappa$ is $\Pi X : \kappa_1, \kappa_2$) is similar. First let us show that $([\kappa])_{\sigma, \rho}$ is a poset. We use the IH to deduce that for all $E \in [T]_{\sigma, \rho}$, $([\kappa])_{\sigma, \rho} \subseteq [\kappa]_{\sigma, \rho}$ is a poset. The IH applies, since $[\kappa_1]_{\sigma, \rho}$ is defined for all $E \in [T]_{\sigma, \rho}$. If it were not, then that would contradict definedness of $([\kappa])_{\sigma, \rho}$, which depends on definedness of $[\kappa_1]_{\sigma, \rho}$, for all $E \in [T]_{\sigma, \rho}$.

The fact that $([\kappa])_{\sigma, \rho} \subseteq [\kappa_1]_{\sigma, \rho}$ is a poset for all $E \in [T]_{\sigma, \rho}$ is sufficient, except for the case when $[T]_{\sigma, \rho}$ is empty. In this case, every $S, S' \in ([\kappa])_{\sigma, \rho}$ satisfy $S \subseteq [\kappa_{\sigma, \rho} S'$. But in that situation, the semantics makes $[\Pi x : T. \kappa_1]_{\sigma, \rho}$ the set containing only the empty set (as the only function with domain $0$). So then $S = S'$, and antisymmetry is satisfied.

Now let $X \subseteq [\kappa]_{\sigma, \rho}$. We must show that $\cap_{\kappa, \sigma, \rho} X$ (call this $A$) is the greatest lower bound of $X$, with respect to the ordering $\subseteq_{\kappa, \sigma, \rho}$. If $X$ is empty, then since $\subseteq_{\kappa, \sigma, \rho}$ compares
Journal of Functional Programming  35

elements of \([\kappa]_{\sigma,\rho}\) pointwise in this case (where \(\kappa = \Pi x : T. \kappa_1\)), we can easily observe that \(x \subseteq_{\kappa,\sigma,\rho} T \cdot \kappa,\sigma,\rho\). for all \(x \in [\kappa]_{\sigma,\rho}\). (Again, if \([T]_{\sigma,\rho}\) is empty, then \(T \cdot \kappa,\sigma,\rho\) is the empty set, which is the sole element of \([\kappa]_{\sigma,\rho}\), as argued just above.) So \(T \cdot \kappa,\sigma,\rho\) is the greatest element of \([\kappa]_{\sigma,\rho}\), as required if \(X\) is empty.

Now consider the case where \(X\) is nonempty. First, let us show \(A \subseteq_{\kappa,\sigma,\rho} B\), for all \(B \in X\). For this, it suffices to consider arbitrary \(E \in [T]_{\sigma,\rho}\), and show \(A(E) \subseteq_{\kappa_1,\sigma[v \mapsto \zeta(E)],\rho} B(E)\). \(A(E)\) is defined to be \(\cap_{\kappa_1,\sigma[v \mapsto \zeta(E)],\rho}\{F(E) \mid F \in X\}\). The IH applies to show that

\[
([\kappa_1]_{\sigma[v \mapsto \zeta(E)],\rho} \cap_{\sigma[v \mapsto \zeta(E)],\rho} \cdot \kappa_1,\sigma[v \mapsto \zeta(E)],\rho) \subseteq_{\kappa_1,\sigma[v \mapsto \zeta(E)],\rho} B(E)
\]

is a complete lattice, and hence \(A(E) \subseteq_{\kappa_1,\sigma[v \mapsto \zeta(E)],\rho} B(E)\), for all \(B \in X\), as required. Similar reasoning shows \(A\) is the greatest such element.

\(\blacksquare\)

**Proof of Lemma 23**

*Proof* 

The proof is by induction on \(\kappa\). If \(\kappa = \star\), then \([\nu\kappa]_{\sigma,\rho} = [\nu]_{\sigma,\rho} = T_{\star,\sigma,\rho}\), as required. Suppose \(\kappa\) is \(\Pi x : T. \kappa'\). Then we have

\[
\begin{align*}
[\nu_{\Pi x : T. \kappa'}]_{\sigma,\rho} & = \text{by definition of } [\nu_{\Pi x : T. \kappa'}] \\
[\lambda x : T. \nu_{\kappa'}]_{\sigma,\rho} & = \text{by the semantics} \\
(E \in [T]_{\sigma,\rho} \mapsto [\nu_{\kappa'}]_{\sigma[v \mapsto \zeta(E)],\rho}) & = \text{IH} \\
(E \in [T]_{\sigma,\rho} \mapsto \kappa',\sigma[v \mapsto \zeta(E)],\rho) & = \text{definition from Figure 10}
\end{align*}
\]

\(\blacksquare\)

**Proof of Lemma 22**

Since \([\kappa]_{\sigma,\rho}\) is defined, we can prove by an easy induction that \(\kappa\) cannot contain any variable in the domain of \(\sigma'\) or \(\rho'\). Then by induction on \(\kappa\), and using Lemma 17, we easily show that \(\subseteq_{\kappa,\sigma,\rho}\) (which is defined by Lemma 20) equals \(\subseteq_{\kappa,\sigma \cup \sigma', \rho \cup \rho'}\). \(\blacksquare\)

**D Proofs for Lifting Lemmas from Section 5.2**

**Proof of Lemma 6**

*Proof* 

The proof is by induction on the structure of \(t\), with case analysis following the definition of \(\text{lift}_{\ldots, \ldots}(-)\).

**Case:** Suppose we have

\[
\text{lift}_{L_1 \rightarrow L_2,v}(\lambda x.t) = \lambda x : \text{lift}(L_1) \cdot t \cdot \text{lift}_{L_2,v,x \mapsto L_1}(t)
\]

The interpretation of this with respect to \(\theta\) and \(\rho'\) is

\[
S \in [\text{lift}(L_1)]_{\theta,\rho'} \mapsto [\text{lift}_{L_2,v,x \mapsto L_1}(t)]_{\theta,\rho'[x \mapsto S]}
\]
36

Stump

By the IH,
\[
\llbracket \text{lift}_{L_2,v}(x \mapsto L_1)(t) \rrbracket_{\theta, \rho'}[x \mapsto \bar{s}] = \llbracket t \rrbracket_{\theta, \rho'[x \mapsto (S, L_1)]}
\]
so the above displayed interpretation is equal to exactly what we get for the value of \(\llbracket \lambda x.t \rrbracket_{\theta, \rho'[x \mapsto (S, L_1)]} \) using the semantics for the latter expression.

**Case:** Suppose we have the following, with \((x \mapsto L') \in v\):

\[
\text{lift}_{L,v}(x \bar{t}) = x \text{ liftargs}_{L,v}(\bar{t})
\]

Let \(\bar{t} = n\). Since we are assuming \(\text{liftargs}_{L,v}(\bar{t})\) is defined, we must have \(L' = L \rightarrow L''\) for some \(L\) and \(L''\), and then

\[
x \text{ liftargs}_{L',v}(\bar{t}) = x ((\mapsto_{L_1} \lambda v.t_1) v) \cdots ((\mapsto_{L_n} \lambda v.t_n) v)
\]

Since we are assuming \(v(x) = L'\), our assumption relating \(v\) and \(\theta\) implies that \(\theta(x) = (S, L')\), for some \(S \in [\text{lift}(L')]_{\emptyset, \emptyset}\). So the interpretation of this term with respect to \(\emptyset\) and \(\rho'\) (where \(\rho'(x) = S\) by assumption) is

\[
S \llbracket ((\mapsto_{L_1} \lambda v.t_1) v) \rrbracket_{\emptyset, \rho'} \cdots \llbracket ((\mapsto_{L_n} \lambda v.t_n) v) \rrbracket_{\emptyset, \rho'}
\]

Suppose \(v = (x_1 \mapsto L'_1), \ldots, (x_k \mapsto L'_k)\), and let \(S_1, \ldots, S_k\) be such that \(\theta(x_1) = (S_1, L'_1), \ldots, \theta(x_k) = (S_k, L'_k)\). Then for all \(i \in [1..n]\), we have

\[
\llbracket ((\mapsto_{L_1} \lambda v.t_1) v) \rrbracket_{\emptyset, \rho'} = \llbracket ((\mapsto_{L_2} \lambda x.t) \bar{x}) \rrbracket_{\emptyset, \rho'} = (S_1 \in [\text{lift}(L'_1)]_{\emptyset, \rho'} \mapsto S_k \in [\text{lift}(L'_k)]_{\emptyset, \rho'} \mapsto \llbracket t_i \rrbracket_{\emptyset, \rho'}^{L_i} = \llbracket (t_i) \rrbracket_{\emptyset, \rho'}^{L_i}) = \llbracket (t_i) \rrbracket_{\emptyset, \rho'}^{L_i}
\]

So the meaning of the original term has been shown to be equal to

\[
S \llbracket t_1 \rrbracket_{\emptyset}^{L_1} \cdots \llbracket t_n \rrbracket_{\emptyset}^{L_n}
\]

This is exactly the value the semantics gives for \(\llbracket (x \bar{t}) \rrbracket_{\emptyset}^{L'}\) in this case. \(\square\)

**Proof of Lemma 10**

**Proof**

The proof is by induction on \(L\). If \(L = *\) then \(\llbracket L \rrbracket_{\emptyset, \rho} = R\), which is nonempty by assumption. If \(L = L_1 \rightarrow L_2\), then choose an element \(E\) from \(\llbracket L_2 \rrbracket_{\emptyset, \rho}\), which is nonempty by the induction hypothesis. Then \(\llbracket \lambda y.\xi(E) \rrbracket_{\emptyset, \rho}\) is easily seen to be in \(\llbracket L_1 \rightarrow L_2 \rrbracket_{\emptyset, \rho}\), by the semantics of function types. \(\square\)

**Proof of Lemma 11**

**Lemma 11**

The proof proceeds by induction on \(L_1\).

**Case:** \(L_1 = *\). Then our assumption that \([t_1]_{\emptyset} \notin \llbracket L_1 \rrbracket_{\emptyset, \rho}\) becomes just \([t_1]_{\emptyset} \notin R\). We can write \(L_2\) as \(L \rightarrow *\) for some \(L\). Let \(n = |L|\). Then we can return the term (to witness the existential in the statement of the lemma) \(\lambda y.\lambda \bar{x}.y\), where \(|\bar{x}| = |L|\), because this term
may easily be seen to be in [[L₁ → L → *]^1], but [[t/y]\hat\lambda\bar{x}.y]_{\beta} is not in [[L → *]^1], because by repeated application of Lemma 10, there are sets E_i ∈ [[L_i/x]_{\alpha,\rho} for all i ∈ [1,n], but then [[\xi(E_i)/x_1,\ldots,\xi(E_n)/x_n]t]_{\beta} = [t]_{\beta}, which is not in R by assumption.

Case: L₁ = L₀ → L₂. If t \neq \hat\lambda x.t' for any x and t', then this implies t \notin R, and so as in the previous case we can return \hat\lambda y.\hat\lambda\bar{x}.y, where |x| = |L|. Suppose t = \hat\lambda x.t' for some x and t'. From our assumption about t and using the semantics of function types, we deduce that there exists some E ∈ [[L_0/x]_{\alpha,\rho} such that [[\xi(E)/x]t']_{\beta} \notin [[L_0]_{\alpha,\rho}. By our IH we know there exists a term \hat\lambda y.\hat\lambda\bar{x}.y ∈ [[L_0 → L_1]_{\alpha,\rho} such that [[((\xi(E)/x)t')/\bar{y}]_{\beta} \notin [[L_1]_{\alpha,\rho}. We can then return the term \hat\lambda y.\hat\lambda\bar{x}.y and apply (Lemma 4). Let \hat\lambda y.\hat\lambda\bar{x}.y \notin [[L_0]_{\alpha,\rho}, because for any [\hat\lambda x.t']_{\beta} ∈ [[L_0]_{\alpha,\rho}, we have

(\hat\lambda y.\hat\lambda\bar{x}.y \xi(E)) =_{\beta} (\hat\lambda y.\hat\lambda\bar{x}.y (\xi(E)/z)^{\bar{y}})

where [[\xi(E)/z]t']_{\beta} ∈ [[L_0/x]_{\alpha,\rho} by our assumptions about E and [\hat\lambda x.t']_{\beta}, and hence, abbreviating [[\xi(E)/z]t']_{\beta} by E', we have [[\xi(E)/y]t']_{\beta} ∈ [[L_1/x]_{\alpha,\rho} by the fact we deduced by the IH about \hat\lambda y.\hat\lambda\bar{x}.y. But [[\xi((\hat\lambda x.t')/y')((\hat\lambda y.\hat\lambda\bar{x}.y (\xi(E)))c_{\beta} is not in [[L_2]_{\alpha,\rho}, because we have

\xi((\hat\lambda x.t')/y')((\hat\lambda y.\hat\lambda\bar{x}.y (\xi(E))) =_{\beta}
(\hat\lambda y.\hat\lambda\bar{x}.y (\xi(E)/x) =_{\beta}
(\hat\lambda x.t') (\xi(E)/x) t' =_{\beta}

And our IH already gave us that [[((\xi(E)/x)t')/y]t']_{\beta} \notin [[L_2]_{\alpha,\rho}, as required.

Proof of Lemma 9

Proof

Instantiate R with [\bar{x}]_{\beta} (the set of closed normalizing terms), which we already observed is in \mathcal{R} (Lemma 4). Let \rho = [x \mapsto [\bar{x}]]_{\beta}. The proof is now by induction on L. If L = *, then [t]_{\beta} ∈ [[L]_{\alpha,\rho} is just [t]_{\beta} ∈ [\bar{x}]_{\beta}. By confluence of reduction, this implies t is normalizing (since it is \beta-equivalent to a normal form). Now suppose L = L₁ → L₂. Then by the semantics, t =_{\beta} \hat\lambda y.\hat\lambda\bar{x}.y t' for some y and t', and we have (applying also Lemma 17)

\forall E \in [[L_1/x]_{\alpha,\rho}. [[\xi(E)/y]t']_{\beta} ∈ [[L_2]_{\alpha,\rho}

Writing L as L → *, we can easily see that [[L_1/x]_{\alpha,\rho} contains the set [\hat\lambda\bar{x}.\hat\lambda y.\bar{y}]_{\beta}, for this reduces just to checking that \hat\lambda y.\bar{y} t in [[*]_{\alpha,\rho}, for normalizing \bar{t}. Since [[*]_{\alpha,\rho} is just [\bar{x}]_{\beta}, this is obviously true. We are allowed to assume here that the values \bar{t} being substituted for the \bar{t} are normalizing, by the induction hypothesis. Now instantiate E in the displayed formula above with [\hat\lambda\bar{x}.\hat\lambda y.\bar{y}]_{\beta}. By the IH, we know that [(\xi(E)/y)t'] is normalizing, and hence so is [\hat\lambda\bar{x}.\hat\lambda y.\bar{y} /yt'], by confluence. This implies that \hat\lambda y.\bar{t} is normalizing, because if, say, normalizing reduction of that term ends up applying \hat\lambda\bar{x}.\hat\lambda y.\bar{y} \hat{\bar{t}} to more than |\bar{t}| arguments, we could just as well instantiate E with \hat\lambda\bar{x}.\hat\lambda y.\bar{y}, and then the result would fail to normalize. This shows that from a normalization perspective, \hat\lambda\bar{x}.\hat\lambda y.\bar{y} \hat{\bar{t}} behaves the same as just \bar{y}, and so t' is normalizing. 

\qed
Proof of Lemma 8

Proof

The proof is by induction on the structure of \( t \). Since \( t \) is normal, the only possibilities for the form of \( t \) are \( \lambda y.t' \) and \( y.t \).

**Case:** \( t = \lambda y.t' \) for some \( y \) and \( t' \). In this case, we cannot have \( L = * \), for then our main assumption would imply that for all \( R \in \mathcal{R} \), and for all \((\theta, R)\)-constrained \( \sigma \), we have \( \lambda y.\sigma t' \) should be in \( R \). No matter what the form of \( t' \) is, we can find either two members of \( \mathcal{R} \) or else two \((\theta, R)\)-constrained valuations that violate this. So \( L = L_1 \rightarrow L_2 \) for some \( L_1 \) and \( L_2 \). Then by the semantics, writing \( \theta' \) for \( \theta[y \mapsto (S, L_1)] \), we have

\[
\langle \langle t \rangle \rangle_{\theta}^{L} = S \in \llbracket \text{lift}(L_1) \rrbracket_{0,0} \rightarrow \langle \langle t' \rangle \rangle_{\theta'}^{L_2}.
\]

By the semantics of function kinds, it suffices to assume an arbitrary \( S \llbracket \text{lift}(L_1) \rrbracket_{0,0} \), and show \( \langle \langle t' \rangle \rangle_{\theta'}^{L_2} \in \llbracket \text{lift}(L_2) \rrbracket_{0,0} \). Now by the semantics for function types, our main assumption implies that for all \( R \in \mathcal{R} \) and for all \((\theta, R)\)-constrained \( \sigma \), we have

\[
\forall E \in \llbracket L_1 \rrbracket_{0,[X \rightarrow R]} \cdot \llbracket (\zeta(E)/y)[\sigma t'] \rrbracket_{c,\beta} \in \llbracket L_2 \rrbracket_{0,[X \rightarrow R]}
\]

This is equivalent to saying that for all \((\theta', R)\)-constrained \( \sigma' \), we have \( \langle \sigma' t' \rangle_{c,\beta} \in \llbracket L_2 \rrbracket_{0,[X \rightarrow R]} \).

That is all that is required to apply our induction hypothesis to conclude the desired \( \langle \langle t' \rangle \rangle_{\theta'}^{L_2} \in \llbracket \text{lift}(L_2) \rrbracket_{0,0} \).

**Case:** \( t = y.t \) for some \( y \) and \( t \). Since \( \text{dom}(\theta) \supseteq \text{FV}(t) \), we have \( \theta(y) = (S, L') \) with \( S \in \llbracket \text{lift}(L') \rrbracket_{0,0} \). Now \( L' \) must be of the form \( \bar{L} \rightarrow L'', \) with \( |L'| = |t| = n, \) for some \( n, \) since otherwise \( \bar{\lambda} \) \( L' = \bar{L} \rightarrow * \) with \( |L'| < |t| \) – we could instantiate our main assumption with \( \bar{\lambda} \cdot \cdot \cdot \cdot \text{for} \) \( R \) and a \((\theta, R)\)-constrained substitution \( \sigma \) where \( \sigma(y) = \bar{\lambda} \bar{x} \bar{\lambda} \bar{y} \bar{\omega} \), \( |\bar{x}| = |L| \), and other values of \( \sigma \) can be filled in arbitrarily to match \( \theta, \) by Lemma 10. This \( \sigma(y) \) is in \( \llbracket \bar{L} \rightarrow \bar{\lambda}x_0[x_0/x_1,t_1,t_1',|x_1|,|t_1|,\beta] \rrbracket, \) but \( \sigma(y) \bar{t} \) has no normal form and hence \( \sigma(y) \bar{\sigma} \bar{\sigma}_t \text{ is not in } \bar{\bar{\lambda}} \cdot \cdot \cdot \cdot \) \( \bar{\sigma} \bar{\sigma}_t \bar{\sigma} \bar{\sigma}_t \).

Similarly, for all \( i \in [1, n] \), our main assumption holds for \( t_i \) and \( L_i \), as we now argue. Suppose \( \bar{t} = \bar{t}', \bar{t}, \bar{t}' \) and \( \bar{L} = \bar{L}', \bar{L}, \bar{L}'' \). If there exists nonempty \( R \in \mathcal{R} \) and \((\theta, R)\)-constrained \( \sigma \) with \( \langle \sigma t_i \rangle_{c,\beta} \in \llbracket L_i \rrbracket_{0,[X \rightarrow R]} \), then by Lemma 11 there exists a \( \bar{\lambda} \bar{t} \) such that \( \bar{\lambda} \bar{t} \bar{t}' \) \( \in \llbracket L_i \rightarrow L'' \rrbracket_{0,[X \rightarrow R]} \) but \( \bar{t}_i \bar{t}_i' \in \llbracket L'' \rrbracket_{0,[X \rightarrow R]} \). We can instantiate our main assumption with \( R \) for \( R \) and a \((\theta, R)\)-constrained substitution \( \sigma \) where all values are arbitrarily filled in to match \( \theta \) using Lemma 10, except

\[
\sigma(y) = \bar{\lambda} \bar{x} \cdot \bar{\lambda} \bar{z} \cdot \bar{\lambda} \bar{x}' \cdot \bar{\lambda} \bar{y} \bar{t}'.
\]

Then \( \sigma(y) \bar{t} \text{ is not in } \bar{\bar{\lambda}} \cdot \cdot \cdot \cdot \) \( \sigma \sigma_\bar{t} \), and we have already deduced that \( \langle \sigma t_i \rangle_{c,\beta} \in \llbracket L'' \rrbracket_{0,[X \rightarrow R]} \). This contradicts our main assumption about \( t \) and \( L \), which implies we should have \( \langle \sigma(y) \bar{t} \rangle_{c,\beta} \in \llbracket L'' \rrbracket_{0,[X \rightarrow R]} \).

For all nonempty \( R \in \mathcal{R} \), we must have \( \llbracket L'' \rrbracket_{0,[X \rightarrow R]} \) a subset of \( \llbracket L \rrbracket_{0,[X \rightarrow R]} \). If not, then from a value in \( \llbracket L'' \rrbracket_{0,[X \rightarrow R]} \) but not in \( \llbracket L \rrbracket_{0,[X \rightarrow R]} \), we could construct a substitution \( \sigma \) again violating our main assumption, as above. So by Lemma 12, \( L = L'' \).

Having established that \( L' = \bar{L} \rightarrow L \) with \( |\bar{L}| = |\bar{t}| = n \), we apply the semantics to get

\[
\langle \langle t \rangle \rangle_{\theta}^{L} = S\langle \langle t_1 \rangle \rangle_{\theta}^{L_1} \cdots \langle \langle t_n \rangle \rangle_{\theta}^{L_n}
\]
Since we have shown that our main assumption holds for \( t_i \) and \( L_i \), for all \( i \in [1, n] \), we can apply the IH to each \( t_i \) to conclude \( \langle\langle t_i \rangle\rangle_\theta^L \in \llbracket \text{lift}(L_i) \rrbracket_{\emptyset, \emptyset} \), for all \( i \in [1, n] \). So by our assumption about \( S \), we have the following, which suffices:

\[
S \langle\langle t_1 \rangle\rangle_\theta^L \cdots \langle\langle t_n \rangle\rangle_\theta^L \in \llbracket \text{lift}(L) \rrbracket_{\emptyset, \emptyset}
\]

\( \square \)

### E Additional Lifting Lemmas

**Lemma 24**

Let \( \theta' = \theta[x_1 \mapsto (S_1, L_1), \cdots, x_n \mapsto (S_n, L_n)] \), and \( \bar{x} = x_1, \cdots, x_n \), with \( x_i \notin \text{FV}(t) \) and \( S_i \in \llbracket \text{lift}(L_i) \rrbracket_{\emptyset, \emptyset} \), for all \( i \in [1, n] \). Then if \( t \bar{x} \leadsto^i \bar{t} \), \( \langle\langle nt(t \bar{x}) \rangle\rangle_{\theta'}^L = \langle\langle nt(t) \rangle\rangle_{\theta'}^{L \rightarrow L}(\bar{S}) \)

**Proof**

The proof is by distinguishing cases related to \( nt(t \bar{x}) \).

**Case:** \( nt(t \bar{x}) = y \bar{t} \) with \( \theta(y) = (\bar{S}, L \rightarrow L) \), where \( |L| \geq |\bar{t}| \). In this case, we must be able to divide \( \bar{t} \) into \( \bar{t}, \bar{t}', \bar{s} \) into \( \bar{s}' \), \( \bar{t}', L \rightarrow L' \), and \( \bar{S} \) into \( \bar{S}', \bar{S}'' \) such that \( |\bar{s}'| = |\bar{t}'| = |L'| = |\bar{S}'| \), \( \bar{t}'' = \bar{s}' \), and \( nt(t) = \lambda x. y \bar{t}' \). In other words, \( t \) must accept some number of arguments and then produce a term headed by \( y \), with the arguments \( \bar{t}'' \) which are not consumed by \( t \) being equal to the suffix \( \bar{s}' \) of \( \bar{x} \). Let \( k = |\bar{t}'| \). Then by the semantics we have the following, where we write \( \langle\langle \bar{t}' \rangle\rangle_{\theta'}^L \) for \( \langle\langle t_1 \rangle\rangle_{\theta'}^L, \ldots, \langle\langle t_k \rangle\rangle_{\theta'}^L \):

\[
\langle\langle nt(t \bar{x}) \rangle\rangle_{\theta'}^L = \langle\langle y \bar{t}' \bar{s}' \rangle\rangle_{\theta'}^L
\]

\[\bar{S} \circ \chi_{\theta'}^{L \rightarrow L}(\bar{t}', \bar{s}') = \bar{S} \circ \chi_{\theta'}^{L \rightarrow L}(\langle\langle \bar{t}' \rangle\rangle_{\theta'}^L, \langle\langle \bar{s}' \rangle\rangle_{\theta'}^L) = \bar{S} \circ \chi_{\theta'}^{L \rightarrow L}(\langle\langle \bar{t}' \rangle\rangle_{\theta'}^L, \bar{S}'')\]

We must prove this equal to \( \langle\langle nt(t) \rangle\rangle_{\theta'}^{L \rightarrow L}(\bar{S}) \), which can be easily done using the semantics and the fact that \( nt(t) = \lambda x. y \bar{t}' \):

\[
\langle\langle nt(t) \rangle\rangle_{\theta'}^{L \rightarrow L}(\bar{S}) = \langle\langle \lambda x. y \bar{t}' \rangle\rangle_{\theta'}^{L \rightarrow L}(\bar{S}, \bar{S}'') = \langle\langle S_1 \in \llbracket \text{lift}(L_1) \rrbracket_{\emptyset, \emptyset} \cdots S_k \in \llbracket \text{lift}(L_k) \rrbracket_{\emptyset, \emptyset} \mapsto \langle\langle y \bar{t}' \rangle\rangle_{\theta'_{\varpi_{x_1 \mapsto S_1}, \cdots, x_k \mapsto S_k}}^{L \rightarrow L}(\bar{S}', \bar{S}'') \rangle_{\theta'_{\varpi_{x_1 \mapsto S_1}, \cdots, x_k \mapsto S_k}}(\bar{S}', \bar{S}'') = \bar{S} \circ \chi_{\theta'}^{L \rightarrow L}(\langle\langle \bar{t}' \rangle\rangle_{\theta'_{\varpi_{x_1 \mapsto S_1}, \cdots, x_k \mapsto S_k}}^{L \rightarrow L}(\bar{S}')) = \bar{S} \circ \chi_{\theta'}^{L \rightarrow L}(\langle\langle \bar{t}' \rangle\rangle_{\theta'_{\varpi_{x_1 \mapsto S_1}, \cdots, x_k \mapsto S_k}, \bar{S}''})
\]

The only difference in the final expressions of the above two equational-reasoning chains is that \( \bar{t}' \) is interpreted with \( \theta' \) in the first and \( \theta[x_1 \mapsto S_1, \cdots, x_k \mapsto S_k] \) in the second. But \( \bar{x} \) (and hence \( \bar{s}' \)) are not free in \( t \) by assumption, so the interpretations are the same by similar reasoning as for Lemma 17 (which we do not further explicate).

**Case:** \( nt(t \bar{x}) = \lambda y. t' \). Then we must have \( nt(t) = \lambda x. \lambda y. t' \), or else we could not eliminate the nested application to \( t \bar{x} \) to get \( \lambda y. t' \). Then we have

\[
\langle\langle nt(t \bar{x}) \rangle\rangle_{\theta'}^L = \langle\langle \lambda y. t' \rangle\rangle_{\theta'}^L
\]
But we also have

$$\langle\langle nf(t) \rangle\rangle^L_{\theta} =$$

$$\langle\langle \lambda x.\lambda y.t' \rangle\rangle^L_{\theta} =$$

$$\langle\langle \lambda y.t' \rangle\rangle^L_{\theta}$$

$$S_1 \in \text{lift}(L_1) \mapsto S_n \in \text{lift}(L_n) \mapsto \langle\langle \lambda y.t' \rangle\rangle^L_{\theta}$$

$\square$

F Proof of Theorem 13

Proof

The proof is by mutual induction on the assumed typing, kinding, or superkinding derivation, for each part of the lemma. We prove the parts successively. In proving a particular part, if we use the induction hypothesis for that part of the lemma, we will just refer to it as the IH. If we use the induction hypothesis for a different part of the lemma, we will indicate which part explicitly.

Proof of part (1)

Case:

$$\Gamma \vdash \star : \square$$

$\llbracket \star \rrbracket_{\sigma,\rho}$ is just $\mathcal{R}$, which is defined.

Case:

$$\Gamma \vdash T : \star \quad \Gamma, x : T \vdash \kappa : \square$$

$$\Gamma \vdash \Pi x : T, \kappa : \square$$

By the IH, $\llbracket T \rrbracket_{\sigma,\rho} \in \mathcal{R}$, and so $\llbracket \Pi x : T, \kappa \rrbracket_{\sigma,\rho}$ is $(E \in \llbracket T \rrbracket_{\sigma,\rho} \rightarrow \llbracket \kappa \rrbracket_{\sigma\rightarrow\zeta(E)}{\rho})$. The latter quantity is defined if for all $E \in \llbracket T \rrbracket_{\sigma,\rho}$, $\llbracket \kappa \rrbracket_{\sigma\rightarrow\zeta(E)}{\rho}$ is, too. Since $\llbracket T \rrbracket_{\sigma,\rho} \in \mathcal{R}$, every element $E$ of $\llbracket T \rrbracket_{\sigma,\rho}$ is of the form $\lambda y.t_{\beta}$, and so $\zeta(E)$ is defined. We may apply the IH to the second premise, since $(\sigma[x \rightarrow \zeta(E)],\rho) \in \llbracket \Gamma, x : T \rrbracket$, because $E \in \llbracket T \rrbracket_{\sigma,\rho}$ (by assumption) and $\zeta(E)_{\beta} = E$ by Lemma 2. This gives definedness of the semantics of the $\Pi$-kind.

Case:

$$\Gamma \vdash \kappa : \square \quad \Gamma, x : \kappa \vdash \kappa' : \square$$

$$\Gamma \vdash \Pi x : \kappa, \kappa' : \square$$

We must show $(S \in \llbracket \kappa \rrbracket_{\sigma,\rho} \rightarrow \llbracket \kappa \rrbracket_{\sigma,\rho}[x \rightarrow S])$ is defined. This is true if $\llbracket \kappa \rrbracket_{\sigma,\rho}$ is defined, which is the case by the IH applied to the first premise; and if for all $S \in \llbracket \kappa \rrbracket_{\sigma,\rho}$, $\llbracket \kappa \rrbracket_{\sigma,\rho}[x \rightarrow S]$ is defined. The latter is true by the IH applied to the second premise.

Proof of part (2)

Case:

$$\llbracket X : \kappa \rrbracket \in \Gamma$$

$$\Gamma \vdash X : \kappa$$
From the definition of $\llbracket \Gamma \rrbracket$, we obtain $\rho(x) \in \llbracket \kappa \rrbracket_{\sigma,\rho}$.

**Case:**

$$\Gamma \vdash \forall : *$$

$$\llbracket \forall \rrbracket_{\sigma,\rho} = \top_{*,\sigma,\rho} = [\mathcal{L}]_{\kappa} \in \mathcal{R} = \llbracket * \rrbracket_{\sigma,\rho}.$$  

**Case:**

$$\Gamma \vdash T : * \quad \Gamma, x : T \vdash T' : *$$

$$\Gamma \vdash \Pi x : T. T' : *$$

We must show $\llbracket \Pi x : T. T' \rrbracket_{\sigma,\rho} \in \mathcal{R}$. The semantics defines $\llbracket \Pi x : T. T' \rrbracket_{\sigma,\rho}$ to be $[A]_{\kappa}^\beta$ for a certain $A$, where if $A$ is defined, then $A \subseteq \mathcal{L}$. So it suffices to shown definedness. By the IH for the first premise, $\llbracket T \rrbracket_{\sigma,\rho} \in \mathcal{R}$. This means that if $E \in \llbracket T \rrbracket_{\sigma,\rho}$, $\zeta(E)$ is defined.

We can then apply the IH to the second premise, since $\sigma[x \mapsto \zeta(E)] \in \llbracket \Gamma, x : T \rrbracket$, to obtain definedness of $\llbracket T' \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho}$.

**Case:**

$$\Gamma \vdash T_1 : * \quad \Gamma, x : T_1 \vdash T_2 : *$$

$$\Gamma \vdash \forall x : T_1. T_2 : *$$

By the IH for the second premise, $\llbracket T_2 \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho} \in \mathcal{R}$, for every $E \in \llbracket T_1 \rrbracket_{\sigma,\rho}$ where $\llbracket T_1 \rrbracket_{\sigma,\rho} \in \mathcal{R}$. So the intersection of all the sets $\llbracket T_2 \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho}$ where $E \in \llbracket T_1 \rrbracket_{\sigma,\rho}$ is a reducibility candidate, since each of those sets is. By the semantics of $\forall$-types quantifying over terms, this is sufficient.

**Case:**

$$\Gamma \vdash \kappa : \Box \quad \Gamma, x : \kappa \vdash T : *$$

$$\Gamma \vdash \forall X : \kappa. T : *$$

Similarly to the previous case: by the IH for the second premise, $\llbracket T_2 \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho} \in \mathcal{R}$, for every $S \in \llbracket \kappa \rrbracket_{\sigma,\rho}$. By the IH part (1) for the first premise, $\llbracket \kappa \rrbracket_{\sigma,\rho}$ is defined. So the intersection of all the sets $\llbracket T_2 \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho}$ where $S \in \llbracket \kappa \rrbracket_{\sigma,\rho}$ is a reducibility candidate, since each of those sets is. By the semantics of $\forall$-types quantifying over types, this is sufficient.

**Case:**

$$\Gamma \vdash T : * \quad \Gamma, x : T \vdash T' : *$$

$$\Gamma \vdash \Pi x : T. T' : *$$

Since $\llbracket \tau x : T. T' \rrbracket_{\sigma,\rho}$ is explicitly defined to be a subset of $\llbracket T \rrbracket_{\sigma,\rho}$, which is in $\mathcal{R}$, by the IH applied to the first premise. Since for any $A \subseteq \mathcal{L}$, $[A]_{\kappa}^\beta$ is in $\mathcal{R}$, to show that $\llbracket \tau x : T. T' \rrbracket_{\sigma,\rho}$ is also in $\mathcal{R}$ it suffices to show definedness of $\llbracket T' \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho}$ (which is used in the predicate picking out the particular subset of $\llbracket T \rrbracket_{\sigma,\rho}$), for $E \in \llbracket T \rrbracket_{\sigma,\rho}$. For such $E$, $\zeta(E)$ is defined and in $E$, so $\sigma[x \mapsto \zeta(E)] \in \llbracket \Gamma, x : T \rrbracket$. So by the IH for the second premise, $\llbracket T' \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho}$ is defined.

**Case:**

$$\Gamma \vdash T : * \quad \Gamma, x : T \vdash T' : \kappa$$

$$\Gamma \vdash \lambda x : T. T' : \Pi x : T. \kappa$$

By the semantics, $\llbracket \lambda x : T. T' \rrbracket_{\sigma,\rho}$ is $(E \in \llbracket T \rrbracket_{\sigma,\rho} \mapsto \llbracket T' \rrbracket_{\sigma[x \mapsto \zeta(E)],\rho})$. We must show that this (meta-level) function is in $\llbracket \Pi x : T. \kappa \rrbracket_{\sigma,\rho}$. By the semantics of kinds, the latter quantity,
if defined, is \( E \in [T]_{\sigma, \rho} \rightarrow_{\iota, \beta} [K]_{\sigma \cdot v \rightarrow \zeta(E), \rho} \). \([T]_{\sigma, \rho}\) is defined by the IH for the first premise. So we must just show that for any \( E \in [T]_{\sigma, \rho} \), \([T]_{\sigma \cdot v \rightarrow \zeta(E), \rho}\) is defined, since \([T]_{\sigma, \rho} \cdot v \rightarrow \zeta(E), \rho\). But this follows by the IH for the second premise.

**Case:**

\[
\frac{\Gamma \vdash \kappa: \Box \quad \Gamma, X : \kappa \vdash T' : \kappa'}{\Gamma \vdash \lambda X : \kappa, T' : \Pi X : \kappa, \kappa'}
\]

This case is an easier version of the previous one. It suffices to assume an arbitrary \( S \in [K]_{\sigma, \rho} \) and show \([T']_{\sigma, \rho}[X \rightarrow S] \in [K']_{\sigma, \rho}[X \rightarrow S]\). But this follows by the IH applied to the second premise. And we have definedness of \([K]_{\sigma, \rho}\) by the IH part (1) for the first premise.

**Case:**

\[
\frac{\Gamma \vdash T : \Pi \times : \kappa, \kappa' \quad \Gamma \vdash t : \kappa'}{\Gamma \vdash T t : \kappa' / \times}[x / \kappa']
\]

By the IH for the first premise, \([T]_{\sigma, \rho} \in [\Pi \times : \kappa, \kappa']_{\sigma, \rho}\). By the semantics of \(\Pi\)-kinds, this means that \([T]_{\sigma, \rho}\) is a function which given any \( E \in [T']_{\sigma, \rho} \), will produce a result in \([K]_{\sigma \cdot v \rightarrow \zeta(E), \rho}\). By the semantics of type applications, \([T]_{\iota, \beta} \sigma \cdot \rho\) is equal to \([T']_{\sigma, \rho}((\sigma \iota)_{\beta})\). This is defined, since \((\sigma \iota)_{\beta} \in [T']_{\sigma, \rho}\), by the IH part (3) for the second premise. So the result of applying the function is indeed in \([t / x]_{\kappa, \sigma, \rho}\), since by Lemma 15, this equals \([K]_{\sigma \cdot v \rightarrow \zeta((\sigma \iota)_{\beta})}, \rho\) (the codomain of the function being applied).

**Case:**

\[
\frac{\Gamma \vdash T : \Pi \times : \kappa, \kappa' \quad \Gamma \vdash T' : \kappa}{\Gamma \vdash T T' : \Pi \times / \times}[x / \kappa']
\]

This case is similar to the previous one, so we skip the details.

**Case:**

\[
\frac{\Gamma \vdash T : \forall x : T', \kappa \quad \Gamma \vdash t : T'}{\Gamma \vdash T t : \forall x / [x / \kappa']}
\]

By the IH applied to the first premise, \([T]_{\sigma, \rho} \in [\forall x : T', \kappa]_{\sigma, \rho}\). By the semantics of \(\forall\)-kinds, this means that for any \( E \in [T']_{\sigma, \rho} \), \([T]_{\sigma, \rho}\) is in \([K]_{\sigma \cdot v \rightarrow \zeta(E), \rho}\). By the IH part (3) applied to the second premise, we have \([\sigma \iota]_{\beta} \in [T']_{\sigma, \rho}\). So we obtain \([T]_{\sigma, \rho} \in [K]_{\sigma \cdot v \rightarrow \zeta((\sigma \iota)_{\beta})}, \rho\). By Lemma 15, \([t / x]_{\kappa, \sigma, \rho}\) equals \([K]_{\sigma \cdot v \rightarrow \zeta((\sigma \iota)_{\beta})}, \rho\), so the case is complete.

**Case:**

\[
\frac{\Gamma \vdash T : \forall x : \kappa, \kappa' \quad \Gamma \vdash T' : \kappa}{\Gamma \vdash T T' : \forall x / [x / \kappa']}
\]

This case is similar to the previous one, using instead the IH part (2) for the second premise, and Lemma 18.

**Case:**

\[
\frac{\Gamma, X : * \vdash t : [L]_{x}}{\Gamma \vdash \uparrow L t : lift(L)}
\]

By the IH part (3) we have \([\sigma \iota]_{\beta} \in [L]_{\sigma, \rho[x \mapsto R]}\) and \([L]_{\rho[x \mapsto R]}\), for all \( R \in \mathcal{P} \). By repeated application of Lemma 17, we can drop \( \sigma \) and \( \rho \) from this statement, leaving only \([X \mapsto R]\). By Lemma 9, we know that \( \sigma \iota \sim_{\iota} i \) for some \( i \). It now suffices to prove
that \(\langle i \rangle^L_0 \in \|\text{lift}(L)\|_{0,0}\) (applying then Lemma 17 to return \(\sigma\) and \(\rho\) to the statement). This follows from Lemma 8, where the required conditions are trivial to establish, because \(FV(i) = \emptyset\).

**Case:**

\[
\begin{array}{c}
X \in^+ T \\
\Gamma \vdash \kappa : \Box \\
\text{Ctors}_X \Theta \\
\Gamma \vdash [\%_X / X]\Theta \\
\Gamma, X : \kappa, \Theta \vdash [T / X]\Theta \\
\Gamma, X : \kappa, \Theta \vdash T : \kappa \\
\end{array}
\]

The semantics defines \(\|\forall X : \kappa | \Theta. T\|_{\sigma, \rho}\) to be \(q\) iff \(F(q) = q\), where

\[
q = \cap_{\kappa, \sigma, \rho} \{F^n(\top_{\kappa, \sigma, \rho}) \mid n \in \mathbb{N}\}
\]

\[
F = (S \in \|\kappa\|_{\sigma, \rho} \mapsto \|T\|_{\sigma, \rho}[\kappa \mapsto S])
\]

Let \(V\) be \(\|\kappa | \Theta\|_{\sigma, \rho}\) (see Figure 13). Applying the IH part (1) to the second premise and the IH part (5) to the fifth premise (along with Lemmas 18 and 23), we see that the conditions of the IH part (8) are satisfied, using also the third and fourth premises. So \((V, \subseteq_{\kappa, \sigma, \rho} \cap_{\kappa, \sigma, \rho})\) is a complete lattice.

We will first prove that \(q\) is defined and in \(V\). It suffices to prove that \(F^n(\top_{\kappa, \sigma, \rho})\) is defined and in \(V\), for each (meta-level) \(n \in \mathbb{N}\), since then we will be applying the lattice operation \(\cap_{\kappa, \sigma, \rho}\) to a subset of \(V\). We show that \(F^n(\|T\|_{\sigma, \rho})\) is (defined and) in \(V\), by an inner induction on (meta-level) \(n \in \mathbb{N}\). For the base case, since \((V, \subseteq_{\kappa, \sigma, \rho} \cap_{\kappa, \sigma, \rho}, \top_{\kappa, \sigma, \rho})\) is a complete lattice, we have \(\cap_{\kappa, \sigma, \rho} \subseteq V\). For the step case, assume that \(F^n(\|T\|_{\sigma, \rho})\) is in \(V\), and show \(F^{n+1}(\|T\|_{\sigma, \rho})\) is also. We can apply the outer IH to the seventh premise to conclude \(F(\|T\|_{\sigma, \rho}^{n+1}([\kappa | \Theta])_{\sigma, \rho})\). The outer IH applies because by the inner IH, \(F^n(\|T\|_{\sigma, \rho})\) is also. So we can satisfy the conditions imposed by the typing context \(\Gamma, X : \kappa, \Theta\) of the seventh premise, as well as the sixth. Then by the IH part (4) applied to the sixth premise, we also have \(\|\Theta\|_{\sigma, \rho}[\kappa \mapsto F^n(\|T\|_{\sigma, \rho})]\). So \(F^{n+1}(\|T\|_{\sigma, \rho})\) is in \(V\), as required.

Next, let \(q' = \cap_{\kappa, \sigma, \rho} \{F^{n+1}(\top_{\kappa, \sigma, \rho}) \mid n \in \mathbb{N}\}\). We can easily observe that \(q' = q\), since \(q\) is nothing other than \(q' \cap_{\kappa, \sigma, \rho} \top_{\kappa, \sigma, \rho}\) and \(\cap_{\kappa, \sigma, \rho} \top_{\kappa, \sigma, \rho}\) is always equal to \(X\) (in any complete lattice). Now by definition of \(F\), \(q'\) is equal to

\[
\cap_{\kappa, \sigma, \rho} \{\|T\|_{\sigma, \rho}[X \mapsto F^n(\top_{\kappa, \sigma, \rho})] \mid n \in \mathbb{N}\}
\]

Since \(X \in^+ T\), by the IH part (7aii) applied to the sixth premise (note that the set in question is not empty), we obtain

\[
\cap_{\kappa, \sigma, \rho} \{\|T\|_{\sigma, \rho}[X \mapsto F^n(\top_{\kappa, \sigma, \rho})] \mid n \in \mathbb{N}\} \subseteq_{\kappa, \sigma, \rho} \{\|T\|_{\sigma, \rho}[X \mapsto q']\}
\]

This reasoning gives us an inclusion in one direction: \(q' \subseteq_{\kappa, \sigma, \rho} F(q)\).

For the other, since \(q \in V\), we may apply the IH part (7ai), since \(X \in^+ T\), to the seventh premise, to show that for all \(n \in \mathbb{N}\):

\[
\|T\|_{\sigma, \rho}[X \mapsto q'] \subseteq_{\kappa, \sigma, \rho} \|T\|_{\sigma, \rho}[X \mapsto F^n(\top_{\kappa, \sigma, \rho})]\]

By the definition of \(\cap_{\kappa, \sigma, \rho}\), this implies

\[
\{\|T\|_{\sigma, \rho}[X \mapsto q'] \mid n \in \mathbb{N}\} \subseteq_{\kappa, \sigma, \rho} \{\|T\|_{\sigma, \rho}[X \mapsto F^n(\top_{\kappa, \sigma, \rho})] \mid n \in \mathbb{N}\}
\]

This concludes the proof of the theorem.
This is equivalent to \( F(q) \subseteq_{\kappa, \sigma, \rho} q' \). Since we now have \( F(q) \subseteq_{\kappa, \sigma, \rho} q' \subseteq_{\kappa, \sigma, \rho} F(q) \), and \( q = q' \), we can conclude \( F(q) = q \) by antisymmetry of \( \subseteq_{\kappa, \sigma, \rho} \).

**Proof of part (3)**

For this part of the proof, we will often chain set-membership statements. So \( x \in y \in z \) means \( x \in y \land y \in z \).

**Case:**

\[
\frac{\Gamma \vdash t : T' \quad \Gamma \vdash T \triangleright T' \quad \Gamma \vdash T : \star}{\Gamma \vdash t : T}
\]

We have \( \sigma t,_{\beta} \in \llbracket T \rrbracket_{\sigma, \rho} \in \mathcal{R} \) by the IH applied to the first premise. By the IH part (2) applied to the third premise, we have \( \llbracket T \rrbracket_{\sigma, \rho} \) defined. We can then apply the IH part (6) to conclude that \( \llbracket T \rrbracket_{\sigma, \rho} = \llbracket T' \rrbracket_{\sigma, \rho} \), which suffices.

**Case:**

\[
\frac{(t \in T) \in \Gamma}{\Gamma \vdash t : T}
\]

The required facts follow from the definition of \( \llbracket \Gamma \rrbracket \), which requires \( \sigma t,_{\beta} \in \llbracket T \rrbracket_{\sigma', \rho'} \in \mathcal{R} \) for some \( \sigma' \) and \( \rho' \) where \( \sigma = \sigma' \cup \sigma'' \) and \( \rho = \rho' \cup \rho'' \) for some \( \sigma'' \) and \( \rho'' \). We may easily show that \( \llbracket T \rrbracket_{\sigma', \rho'} = \llbracket T \rrbracket_{\sigma, \rho} \) in this case.

**Case:**

\[
\frac{\Gamma \vdash t' : T \quad t \triangleright \ast t'}{\Gamma \vdash t : T}
\]

By the IH applied to the first premise, we have \( \sigma t,_{\beta} \in \llbracket T \rrbracket_{\sigma, \rho} \in \mathcal{R} \). Since \( t \triangleright \ast t' \), we know \( \sigma t,_{\beta} = \sigma t',_{\beta} \), which suffices.

**Case:**

\[
\frac{\Gamma \vdash t : T \quad \Gamma \vdash T \subseteq T' : \ast}{\Gamma \vdash t : T'}
\]

By the IH applied to the first premise, we have \( \sigma t,_{\beta} \in \llbracket T' \rrbracket_{\sigma, \rho} \). By the IH part (5) applied to the second premise, we have \( \llbracket T \rrbracket_{\sigma, \rho} \subseteq \llbracket T' \rrbracket_{\sigma, \rho} \) and \( \llbracket T' \rrbracket_{\sigma, \rho} \in \mathcal{R} \). This is sufficient for this case.

**Case:**

\[
\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}
\]

This follows from the definition of \( \llbracket \Gamma \rrbracket \).

**Case:**

\[
\frac{\text{FV}(t) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \lambda x . t : \mathcal{U}}
\]

We must show that \( \sigma \lambda x . t,_{\beta} \in \llbracket \mathcal{U} \rrbracket_{\beta} \). For this, it suffices to observe that \( \sigma \lambda x . t \) is closed, since \( \text{FV} \subseteq \text{dom}(\Gamma) \) (and we can easily prove that \( \text{dom}(\sigma) = \text{dom}(\Gamma) \)).
Since Case:
By Lemma 15, this is equivalent to Case:
the semantics of instantiate the quantifier in the previous formula to obtain
But Case:

\[ \exists \sigma \| x \mapsto [\xi(E)]_{\sigma} \| \beta \]

By the IH applied to the first premise, \([\sigma[t]]_{\beta} \in [\Pi x : T . T]_{\sigma, \rho} \). This means that there exists a \( \lambda \)-abstraction \( \lambda x . \bar{t} \) such that \( \lambda x . \bar{t} = _{c_{\beta}} \sigma \bar{t} \), by the definition of reducibility candidate. Furthermore, the semantics of \( \Pi \)-types tells us that for any \( E \in [T_{1}]_{\sigma, \rho} \), \( [\xi(E)/\bar{x}]_{\sigma} \| \beta \) is \( [\gamma(T)_{\sigma}]_{\sigma, \rho} \). By the IH applied to the second premise, \([\sigma[t']]_{\beta} \in [T_{1}]_{\sigma, \rho} \), so we can instantiate the quantifier in the previous formula to obtain
\[ [\exists \sigma[t']_{\beta} / \bar{x}]_{\beta} \in [T_{2}]_{\sigma, \rho_{[\xi]_{\sigma}}} \]

By Lemma 15, this is equivalent to
\[ [\exists \sigma[t']_{\beta} / \bar{x}]_{\beta} \in [T_{2}]_{\sigma, \rho} \]

Since \( \sigma(t') = _{c_{\beta}} (\lambda x . \bar{t}) \sigma[t'] = _{c_{\beta}} [\exists \sigma[t']_{\beta} / \bar{x}]_{\beta} \), this is sufficient.

Case:

\[ \Gamma \vdash t : \Pi x : T_{1} . T_{2} \quad \Gamma \vdash t' : T_{1} \]

By the IH, \([\sigma[t]]_{\beta} \in [\Pi x : T_{1} \Rightarrow T_{2}]_{\sigma, \rho} \), for all \( S \in [\Pi x \Rightarrow \sigma]_{\sigma, \rho} \). This is sufficient to prove \([\sigma[t]]_{\beta} \in [\forall x : \sigma . T]_{\sigma, \rho} \), by the semantics of \( \forall \)-types.

Case:

\[ \Gamma \vdash t : \forall X : \kappa . T \quad \Gamma \vdash T' : \kappa \]

By the semantics of \( \forall \)-types and the IH applied to the first premise, we have \([\sigma[t]]_{\beta} \in [T]_{\sigma, \rho[X \mapsto S]} \), for all \( S \in [\pi]_{\sigma, \rho} \). Since \([T']_{\sigma, \rho} \in [\forall x : \sigma . \pi]_{\sigma, \rho} \) by the IH part (2) applied to the second premise, we can derive \([\sigma[t]]_{\beta} \in [T']_{\sigma, \rho[X \mapsto T']} \). By Lemma 18, this is equivalent to the required \([\sigma[t]]_{\beta} \in [\forall X : \sigma . T']_{\sigma, \rho} \).

Case:

\[ \Gamma \vdash t : T \quad \Gamma \vdash t : [t/x]T' \]

By the IH, we have \([\sigma[t]]_{\beta} \in [T]_{\sigma, \rho} \) and \([\sigma[t]]_{\beta} \in [\exists x : T']_{\sigma, \rho} \). By Lemma 15, the latter is equivalent to \([\sigma[t]]_{\beta} \in [T']_{\sigma, \rho[X \mapsto \xi]_{\sigma, \rho}} \). These two facts about \([\sigma[t]]_{\beta} \) are sufficient, by the semantics of \( t \)-types, for the desired conclusion.

Case:

\[ \Gamma \vdash t : [t/x]T' \quad \Gamma \vdash t : T \]
The desired conclusion follows easily from the IH and the semantics of $\iota$-types.

**Case:**

\[
\Gamma \vdash t : T, T' \quad \Gamma \vdash t : [t/x]T'
\]

The desired conclusion follows easily from the IH, the semantics of $\iota$-types, and Lemma 15.

**Case:**

\[
\Gamma \vdash T : \star \\ \Gamma, x : T \vdash t : T' \quad x \notin \text{FV}(t)
\]

By the IH applied to the second premise, we have $[\sigma[x \mapsto \zeta(E)]t]_{\iota,\beta} \in [T']_{\sigma[x \mapsto \zeta(E)],\rho}$, for any $E \in [T]_{\sigma,\rho}$. This is because $[T]_{\sigma,\rho} \in \mathcal{R}$, by the IH part (2) applied to the first premise. Since $x \notin \text{FV}(t)$, we know $[[\sigma[x \mapsto \zeta(E)]t]_{\iota,\beta} = [\sigma t]_{\iota,\beta}$. By the semantics of $\forall$-types, this suffices to show the desired conclusion.

**Case:**

\[
\Gamma \vdash t : \forall x : T_1, T_2 \\ \Gamma \vdash t' : T_1 \\
\Gamma \vdash t : [t'/x]T_2
\]

The result follows easily by the IH applied to the premises, the semantics of $\forall$-types, and Lemma 15.

**Proof of parts (4) and (5)**

These both follow directly from the semantics, using the IH part (2), for proving (4), and the IH part (3), for proving (5).

**Proof of part (6)**

**Case:**

\[
N = \nu X : \kappa | \Theta, T \\
\Gamma \vdash N \triangleright [N/X]T
\]

By the definition of $[[N]_{\sigma,\rho}$, since this quantity is defined, we have

$[[N]_{\sigma,\rho} = [T]_{\sigma,\rho}[X \mapsto [N]_{\sigma,\rho}]$

By Lemma 18, this is sufficient.

**Case:**

\[
\Gamma \vdash (\lambda x : T, T') t \triangleright [t/x]T'
\]

By the semantics, we have

$[\lambda x : T, T'] t]_{\sigma,\rho} = (E \in [T]_{\sigma,\rho} \mapsto [T']_{\sigma[x \mapsto \zeta(E)],\rho}([\sigma t]_{\iota,\beta})$

The right-hand side is equal to $[[T']_{\sigma[x \mapsto \zeta(E)],\rho}]_{\sigma[x \mapsto \zeta(E)],\rho}$, which by Lemma 15 is equal to $[[t/x]T']_{\sigma,\rho}$, as required.
Case:

\[ \Gamma \vdash (\lambda X : \kappa. T) \Updownarrow [T'/X]T \]

By the semantics, we have

\[ \llbracket (\lambda X : \kappa. T) \rrbracket_{\sigma, \rho} = (S \in \llbracket \kappa \rrbracket_{\sigma, \rho} \mapsto \llbracket T \rrbracket_{\sigma, \rho}[X \mapsto S])(\llbracket T' \rrbracket_{\sigma, \rho}) \]

This is equal to \[ \llbracket T \rrbracket_{\sigma, \rho}[X \mapsto [T']_{\sigma, \rho}] \], which by Lemma 18 is equal to the desired \( \llbracket [T'/X]T \rrbracket_{\sigma, \rho} \).

Case:

\[ T \rightsquigarrow T' \]

By the semantics, we have

\[ \llbracket T \rrbracket_{\sigma, \rho} = \llbracket T \rrbracket_{\sigma, \rho}(\llbracket \sigma \rrbracket_{\beta}) \]

Since reduction of pure lambda-calculus terms cannot increase the set of free variables, we know that \( FV(t') \subseteq FV(t) \), and hence \( \sigma t \rightsquigarrow \sigma t' \). So \( \llbracket \sigma t \rrbracket_{\beta} = \llbracket \sigma t' \rrbracket_{\beta} \), and so we have

\[ \llbracket T \rrbracket_{\sigma, \rho}(\llbracket \sigma t \rrbracket_{\beta}) = \llbracket T \rrbracket_{\sigma, \rho}(\llbracket \sigma t' \rrbracket_{\beta}) \]

which again by the semantics equals the desired \( \llbracket T t' \rrbracket_{\sigma, \rho} \).

Case:

\[ \Gamma \vdash t : X \notin FV(T') \]

By the semantics,

\[ \llbracket \forall X : \kappa. T' \rrbracket_{\sigma, \rho} = \cap_{E} \{ \llbracket T' \rrbracket_{\sigma, \rho}[E \mapsto t] | E \in \llbracket T \rrbracket_{\sigma, \rho} \} \]

By the IH part (3) applied to the first premise, we know \( \llbracket \sigma t \rrbracket_{\beta} \in \llbracket T \rrbracket_{\sigma, \rho} \), and hence we can deduce that \( \llbracket T' \rrbracket_{\sigma, \rho} \) is non-empty. So by the definition of \( \cap_{E} \), we have

\[ \cap_{E} \{ \llbracket T' \rrbracket_{\sigma, \rho}[E \mapsto t] | E \in \llbracket T \rrbracket_{\sigma, \rho} \} = \cap_{E} \{ \llbracket T' \rrbracket_{\sigma, \rho} | E \in \llbracket T \rrbracket_{\sigma, \rho} \} \]

Since \( X \notin FV(T') \), we then have

\[ \cap_{E} \{ \llbracket T' \rrbracket_{\sigma, \rho} | E \in \llbracket T \rrbracket_{\sigma, \rho} \} = \cap_{E} \{ \llbracket T' \rrbracket_{\sigma, \rho} | E \in \llbracket T \rrbracket_{\sigma, \rho} \} \]

And the latter quantity is just equal to \( \llbracket T' \rrbracket_{\sigma, \rho} \), as required.

Case:

\[ X \notin FV(T) \]

By the semantics, we have

\[ \llbracket \forall X : \kappa. T \rrbracket_{\sigma, \rho} = \cap_{S} \{ \llbracket T \rrbracket_{\sigma, \rho}[X \mapsto S] | S \in \llbracket \kappa \rrbracket_{\sigma, \rho} \} \]

Since \( X \notin FV(T) \), we have

\[ \cap_{S} \{ \llbracket T \rrbracket_{\sigma, \rho}[X \mapsto S] | S \in \llbracket \kappa \rrbracket_{\sigma, \rho} \} = \cap_{S} \{ \llbracket T \rrbracket_{\sigma, \rho} | S \in \llbracket \kappa \rrbracket_{\sigma, \rho} \} \]

By Lemma 20, the set whose intersection we are trying to take is nonempty (as \( \llbracket \kappa \rrbracket_{\sigma, \rho} \) always contains \( T_{\kappa, \sigma, \rho} \)). So the latter quantity is equal to the desired \( \llbracket T \rrbracket_{\sigma, \rho} \).
48

Stump

Case: \[ \bar{x} \notin FV(t) \quad \bar{L} = |\bar{x}| \]
\[
\frac{}{\Gamma \vdash \bar{L} \downarrow \lambda \bar{x}.(t \bar{x}) \triangleright \bar{L} \downarrow t}
\]

Let \( n = |\bar{L}| \). An easy inner induction shows that the value of \( \| \bar{L} \downarrow \lambda \bar{x}.(t \bar{x}) \|_{\bar{L},\rho} \) equals
\[
S_1 \in \| \text{lift}(L_1) \|_{\bar{L},\rho} \rightarrow \cdots \rightarrow S_n \in \| \text{lift}(L_n) \|_{\bar{L},\rho} \rightarrow \langle \langle nf(t \bar{x}) \rangle \rangle_{\bar{L},\rho} \uparrow_{\bar{L}} (S_1, \ldots, S_n, L_n)
\]
The desired result then follows from Lemma 24.

Case: \[
\text{lift}_{L,\rho}(t) = T
\]
\[
\frac{}{\Gamma \vdash \text{lift}_{L,\rho}(t) \triangleright T}
\]

Note that this conversion is only allowed when the call in the premise to \( \text{lift}_{\bar{L},-}(-) \) is defined. The desired result follows from Lemma 6.

Case: \[
\bar{L} = |\bar{x}| = |\bar{T}|
\]
\[
\frac{}{\Gamma \vdash (\bar{L} \downarrow \lambda \bar{x}.t) \triangleright (\bar{L} \downarrow \lambda \bar{x}.t) \triangleright \bar{L} \downarrow (\lambda \bar{x}.(t \bar{x})) \triangleright \bar{T}}
\]

This follows by a straightforward application of the semantics.

Proof of parts (7ai), (7b), (7ci), and (7d)

Mostly we will just consider the cases where \( X \in^+ T \), since the others are dual. Note that under the conditions of the lemma, Lemma 20 applies to show that \( (\| \bar{L} \|_{\bar{L},\rho}) \subseteq \bar{L} \subseteq \bar{L} \subseteq \bar{L} \) and \( \cap \bar{L} \subseteq \bar{L} \) is a complete lattice. We will use this fact in the cases below.

Case: \[
(Y : \kappa) \in \Gamma
\]
\[
\frac{}{\Gamma \vdash Y : \kappa}
\]

If \( X = Y \), then for (7ai) we obtain the required \( S \subseteq \bar{L} \subseteq \bar{L} \subseteq \bar{L} \subseteq \bar{L} \subseteq \bar{L} \) by applying Lemma 17 to our assumption of \( S \subseteq \bar{L} \subseteq \bar{L} \subseteq \bar{L} \subseteq \bar{L} \subseteq \bar{L} \), to add in the irrelevant \( \sigma_2 \) and \( \rho_2 \). This case cannot arise for (7b), since it is not the case that \( X \in^+ X \). If \( X \neq Y \), then \( \| Y \|_{\bar{L},\rho} = \| Y \|_{\bar{L},\rho} \equiv \| Y \|_{\bar{L},\rho} \), and hence the subset relation holds by reflexivity.

Case: \[
\| \overline{N} \|_{\bar{L},\rho} = \| \overline{N} \|_{\bar{L},\rho}
\]
by the semantics of \( \overline{N} \), and the result follows by reflexivity of \( \subseteq^+\bar{L},\rho_{\bar{L}} \).

Case: \[
\frac{}{\Gamma \vdash T_1 : * \quad \Gamma, \bar{x} : T_1 \vdash T_2 : *}
\]
\[
\frac{}{\Gamma \vdash \Pi \bar{x} : T_1, T_2 : *}
\]

We must show \( \| \Pi \bar{x} : T_1, T_2 \|_{\bar{L},\rho} = \| \Pi \bar{x} : T_1, T_2 \|_{\bar{L},\rho} \).

So assume an arbitrary \( \| \lambda \bar{x}.t \|_{\bar{L},\rho} \subseteq \| \Pi \bar{x} : T_1, T_2 \|_{\bar{L},\rho} \), and show \( \| \lambda \bar{x}.t \|_{\bar{L},\rho} \subseteq \| \Pi \bar{x} : T_1, T_2 \|_{\bar{L},\rho} \).

For the latter, by the semantics of \( \Pi \)-types, it suffices to assume arbitrary \( E \in \| T_1 \|_{\bar{L},\rho} \).
and show \([\zeta(E)/x]t)c_\beta \in \llbracket T_2 \rrbracket_{\sigma(x \to \zeta(E)), \rho[X \to S']}\]. Now by the IH part (7b), \(E \in \llbracket T_1 \rrbracket_{\sigma, \rho[X \to S']}\), because \(X \in^\epsilon T_1\) by the definition of polarity for \(\Pi\)-types. So by the semantics of \(\Pi\)-types again, and the assumption we made that \([\lambda x : T_1 \cdot \lambda x : T_2]_{\sigma, \rho}\), we have \([\zeta(E)/x]t)c_\beta \in \llbracket T_2 \rrbracket_{\sigma(x \to \zeta(E)), \rho[X \to S']}\]. Now we may apply the IH to conclude \([\zeta(E)/x]t)c_\beta \in \llbracket T_2 \rrbracket_{\sigma(x \to \zeta(E)), \rho[X \to S']}\), as required.

**Case:**

\[
\Gamma \vdash T_1 \vdash \ast \quad \Gamma, x : T_1 \vdash T_2 \vdash \ast
\]

This case is very similar to the previous one, so we omit it.

**Case:**

\[
\Gamma \vdash \kappa : \Box \quad \Gamma, Y : \kappa \vdash T : \ast
\]

It suffices to assume arbitrary \(t \in \llbracket \forall Y : \kappa . T \rrbracket_{\sigma, \rho}\), and an arbitrary \(B \in \llbracket \kappa \rrbracket_{\sigma, \rho[X \to S']}\), and show \(t \in \llbracket T \rrbracket_{\sigma, \rho[X \to B, X \to S']}\). By the IH part (7d), which applies because \([\kappa]_{\sigma, \rho}\) is defined by the IH part (1) applied to the first premise, we have \(B \in \llbracket \kappa \rrbracket_{\sigma, \rho}\). This gives us \(t \in \llbracket \forall \beta \cdot Y \to \kappa \rrbracket_{\sigma, \rho}\) by the semantics of \(\forall\)-types and our assumption about \(t\). Then by the IH, we have \(t \in \llbracket \forall \beta \cdot Y \to \kappa \rrbracket_{\sigma, \rho[X \to B, X \to S']}\), as required.

**Case:**

\[
\Gamma \vdash T : \ast \quad \Gamma, x : T \vdash T' : \ast
\]

It suffices to assume an arbitrary \(E \in \llbracket tx : T . T' \rrbracket_{\sigma, \rho}\), and show \(E \in \llbracket tx : T . T' \rrbracket_{\sigma, \rho[X \to S']}\). By the semantics of \(t\)-types, we have \(E \in \llbracket T \rrbracket_{\sigma, \rho}\) and \(E \in \llbracket T \rrbracket_{\sigma(x \to \zeta(E)), \rho}\). By the IH applied to the first premise, we have \(E \in \llbracket T \rrbracket_{\sigma, \rho[X \to S']}\). Since \(\llbracket T \rrbracket_{\sigma, \rho} \in \mathcal{R}\) by the IH part (2) applied to the first premise, we may use the IH for the second premise to obtain \(E \in \llbracket T \rrbracket_{\sigma(x \to \zeta(E)), \rho[X \to S']}\), as well. These facts suffice to show \(E \in \llbracket tx : T . T' \rrbracket_{\sigma, \rho[X \to S']}\).

**Case:**

\[
\Gamma \vdash T : \ast \quad \Gamma, x : T \vdash T' : \kappa
\]

By the definition of \(\subseteq_{\Pi \to T, \kappa, \sigma_1, \rho_1}\) and the semantics of type-level \(\lambda\)-abstractions, it suffices to assume an arbitrary \(E \in \llbracket T \rrbracket_{\sigma_1, \rho_1}\), and show \(\llbracket T' \rrbracket_{\sigma(x \to \zeta(E)), \rho} \subseteq_{\kappa, \sigma_1(x \to \zeta(E)), \rho_1} \llbracket T \rrbracket_{\sigma(x \to \zeta(E)), \rho[X \to S']}\). But this follows by the IH, instantiated with \(\sigma_1[x \mapsto \zeta(E)\}\) for \(\sigma_1\) (note that we can apply Lemma 22 to get \(S \subseteq_{\kappa, \sigma_1[x \mapsto \zeta(E), \rho_1]} S\)).

**Case:**

\[
\Gamma \vdash \kappa : \Box \quad \Gamma, X : \kappa \vdash T' : \kappa'
\]

This case is very similar to the previous one, so we omit it.

**Case:**

\[
\Gamma \vdash T_1 : \Pi x : T' . \kappa \quad \Gamma \vdash t : T'
\]

\[
\Gamma \vdash \Pi t_1 : [t/x] \kappa
\]
By Lemma 15, and applying also the semantics for type-level applications, this is equivalent (since \(g\) gives us \(F\)), we are using \(\text{equivalent to}\) equivalent to, and the definition of \(\subseteq\) for \(\Pi\)-kinds, we have

\[
\forall E \in [T']_{\sigma_1, \rho_1} \cdot \|T\|_{\sigma, \rho}(E) \subseteq \kappa, [\tau_{\zeta(E)}]_{\rho_1} \cdot \|T\|_{\sigma, \rho}[x \rightarrow S'](E)
\]

By the IH applied to the second premise, we have \(\lbrack \sigma_i \rbrack_{\rho_1} \in [T']_{\sigma_1, \rho_1}\), since \(\lbrack \sigma_i \rbrack_{\rho_1}\) is defined (since \(\Pi x : T', \kappa\) is), we can prove by a straightforward induction that \(\lbrack T' \rbrack_{\sigma_1, \rho_1} = [T']_{\sigma_1, \rho_1}\). We can then instantiate the displayed formula above, with \(\lbrack \sigma_i \rbrack_{\rho_1}\) for \(E\). This gives us

\[
\|T\|_{\sigma, \rho}(\lbrack \sigma_i \rbrack_{\rho_1}) \subseteq \kappa, \lbrack x_{\tau_{\zeta}} \lbrack \sigma_i \rbrack_{\rho_1} \rbrack_{\rho_1} \cdot \|T\|_{\sigma, \rho}[x \rightarrow S'](\lbrack \sigma_i \rbrack_{\rho_1})
\]

By Lemma 15, and applying also the semantics for type-level applications, this is equivalent to what we had to prove:

\[
\|T\|_{\sigma, \rho} \subseteq \lbrack x / i \rbrack_{\kappa, \sigma_1, \rho_1} \|T\|_{\sigma, \rho}[x \rightarrow S']
\]

**Case:**

\[
\Gamma \vdash T : \Pi X : \kappa : \kappa' \quad \Gamma \vdash T' : \kappa
\]

\[
\Gamma \vdash T \uparrow T' : [T'/X]_{\kappa'}
\]

This case is mostly similar to the previous one, so we omit the details. The one main difference is that we must use the fact that, by the definition of \(X \in \uparrow T\), we cannot have \(X\) free in \(T'\). So \(\lbrack T' \rbrack_{\sigma, \rho} = [T']_{\sigma, \rho}[x \rightarrow S']\).

**Case:**

\[
\Gamma, X : * \vdash t : \lbrack L \rbrack_X
\]

\[
\Gamma \vdash \uparrow t : \text{type}(\lbrack L \rbrack)
\]

The type in question has no free type variables, so this case is trivial.

**Case:**

\[
\Gamma \vdash \kappa : \square \quad \text{Ctors}_{\kappa} \Theta \quad \Gamma, Y : \kappa \vdash \Theta : *
\]

\[
\Gamma \vdash [L_{\kappa}/Y]_{\Theta} \quad \Gamma, Y : \kappa, \Theta \vdash T : \kappa \quad \Gamma, Y : \kappa, \Theta \vdash [T/Y]_{\Theta} \quad Y \in \uparrow T
\]

\[
\Gamma \vdash \forall Y : \kappa \cdot [\Theta, T : \kappa]
\]

Let \(F = (S_1 \in \lbrack \kappa \rbrack_{\sigma, \rho} \mapsto \lbrack T \rbrack_{\sigma, \rho}[y \rightarrow S_1])\) and \(G = (S_1 \in \lbrack \kappa \rbrack_{\sigma, \rho} \mapsto \lbrack T \rbrack_{\sigma, \rho}[y \rightarrow S_1, y \rightarrow S_1])\). It suffices to show \(F^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1}) \subseteq \kappa, [\sigma_1]_{\rho_1} \cdot G^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1})\), for all (meta-level) \(n \in \mathbb{N}\); if this holds, it will imply that \(\lbrack \forall Y : \kappa \cdot [\Theta, T]_{\sigma, \rho}\) is a lower bound of \(\{G^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1}) \mid n \in \mathbb{N}\}\), and hence necessarily less than or equal to the great lower bound of that set, which is, by definition, \(\lbrack \forall Y : \kappa \cdot [\Theta, T]_{\sigma, \rho}[y \rightarrow S']\). We proceed by inner induction on \(n \in \mathbb{N}\). For the base case, we have \(\lbrack \kappa, \sigma_1, \rho_1 \subseteq \kappa, \sigma_1, \rho_1 \cdot \kappa, \sigma_1, \rho_1\) by reflexivity. For the step case, assume \(F^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1}) \subseteq \kappa, [\sigma_1]_{\rho_1} \cdot G^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1})\), and show \(F(F^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1})) \subseteq \kappa, [\sigma_1]_{\rho_1} \cdot G(G^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1}))\). The latter formula is equivalent to

\[
\|T\|_{\sigma, \rho}[y \rightarrow F^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1})] \subseteq \kappa, [\sigma_1]_{\rho_1} \cdot \|T\|_{\sigma, \rho}[y \rightarrow G^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1})]
\]

This now follows from the outer \(\text{IH}\) applied to the fifth premise, where for \(S \subseteq \kappa, [\sigma_1]_{\rho_1} S'\), we are using \(F^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1}) \subseteq \kappa, [\sigma_1]_{\rho_1} G^n(\lbrack T \rbrack_{\kappa, \sigma_1, \rho_1})\) (which holds by our inner IH).

**Case:**

\[
\Gamma \vdash * : \square
\]
It suffices to note $\llbracket \ast \rrbracket_{\sigma, \rho} = \llbracket \ast \rrbracket_{\sigma, \rho[X \leftarrow S']}$.

**Case:**

$$\frac{\Gamma \vdash T : \ast \quad \Gamma, x : T \vdash \kappa : \Box}{\Gamma \vdash \Pi x : T : \kappa : \Box}$$

By the IH part (1) applied to the first premise, we can deduce that $\llbracket \Pi x : T, \kappa \rrbracket_{\sigma, \rho}$ equals $(E \in \llbracket T \rrbracket_{\sigma, \rho} \rightarrow \llbracket \kappa \rrbracket_{\sigma[X \leftarrow \zeta(E)], \rho})$. So given a function $f$ from this set, it suffices to show that $f$ is also in $(E \in \llbracket T \rrbracket_{\sigma, \rho[X \leftarrow S']} \rightarrow \llbracket \kappa \rrbracket_{\sigma[X \leftarrow \zeta(E)], \rho[X \leftarrow S']})$. So assume an arbitrary $E \in \llbracket T \rrbracket_{\sigma, \rho[X \leftarrow S']}$. Since $X \in \neg T$ by the definition of polarity for a $\Pi$-kind, by the IH applied to the first premise, we have $E \in \llbracket T \rrbracket_{\sigma, \rho}$. So $f(E) \in \llbracket \kappa \rrbracket_{\sigma[X \leftarrow \zeta(E)], \rho[X \leftarrow S']}$, by our assumption about $f$. Then by the IH applied to the second premise, we obtain $f(E) \in \llbracket \kappa \rrbracket_{\sigma[X \leftarrow \zeta(E)], \rho[X \leftarrow S']}$, which is all we needed to show.

**Case:**

$$\frac{\Gamma \vdash \kappa : \Box \quad \Gamma, x : \kappa \vdash \kappa' : \Box}{\Gamma \vdash \Pi x : \kappa, \kappa' : \Box}$$

This case is similar to the previous one, so we omit it.

**Proof of parts (7aii) and (7ci)**

Let $q$ be $\cap_{\kappa, \sigma_1, \rho_1} A$ (and recall that $A$ is not empty, by assumption). As in the proof of part (7aii) above, we will use the fact that $(\llbracket \kappa \rrbracket_{\sigma_1, \rho_1}, \subseteq_{\kappa, \sigma_1, \rho_1}, \cap_{\kappa, \sigma_1, \rho_1})$ is a complete lattice, by Lemma 20 (we are implicitly assuming $\llbracket \kappa \rrbracket_{\sigma_1, \rho_1}$ is defined, by assuming $A \subseteq \llbracket \kappa \rrbracket_{\sigma_1, \rho_1}$. In particular, we will use the fact that $q \subseteq_{\kappa, \sigma_1, \rho_1} S$ for all $S \in A$, which holds by basic properties of complete lattices.

**Case:**

$$\frac{(Y : \kappa) \in \Gamma}{\Gamma \vdash Y : \kappa}$$

If $X = Y$ then $\kappa = \kappa'$, and $\cap_{\kappa, \sigma_1, \rho_1} \{ \llbracket X \rrbracket_{\sigma, \rho[X \leftarrow S']} \mid S \in A \} = \cap_{\kappa, \sigma_1, \rho_1} A = \llbracket X \rrbracket_{\sigma, \rho[X \leftarrow q]}$.

**Case:**

$$\frac{\Gamma \vdash \forall : \ast}{\cap \{ \llbracket \forall \rrbracket_{\sigma, \rho} \} = [\mathcal{L}]_{\varepsilon \beta} = \llbracket \forall \rrbracket_{\sigma, \rho[X \leftarrow q]}, \text{ where the last step uses Lemma 17.}$$

**Case:**

$$\frac{\Gamma \vdash T : \kappa' \kappa' \simeq \kappa \quad \Gamma \vdash \kappa : \Box}{\Gamma \vdash T : \kappa}$$

The proof for this case is very similar to the case for the proof of part (7ai) above, so we omit it.

**Case:**

$$\frac{\Gamma \vdash T_1 : \ast \quad \Gamma, x : T_1 \vdash T_2 : \ast}{\Gamma \vdash \Pi x : T_1, T_2 : \ast}$$

We must show $\cap \{ \llbracket \Pi x : T_1, T_2 \rrbracket_{\sigma, \rho[X \leftarrow S]} \mid S \in A \} \subseteq \llbracket \Pi x : T_1, T_2 \rrbracket_{\sigma, \rho[X \leftarrow q]}$. 

\[ \cap \{ \llbracket \Pi x : T_1, T_2 \rrbracket_{\sigma, \rho[X \leftarrow S]} \mid S \in A \} \subseteq \llbracket \Pi x : T_1, T_2 \rrbracket_{\sigma, \rho[X \leftarrow q]} \]
52

**Stump**

So assume an arbitrary $[\lambda x.t]_{\beta} \in \cap \{ [[\Pi x : T_1. T_2]_{\sigma \rho \kappa} x \to S] \mid S \in A \}$, and show $[\lambda x.t]_{\beta} \in [[\Pi x : T_1. T_2]_{\sigma \rho \kappa} x \to q]$. For the latter, assume an arbitrary $E \in [[T_1]_{\sigma \rho \kappa} x \to q]$, and show $[[\xi E/x]_{\beta}]_{\sigma \rho \kappa} x \to q$. Since $X \in T_1$ by the definition of polarity for $t$-types, and since $q \subseteq S \subseteq q$, we may apply the IH part (7b) to deduce $E \in [[T_1]_{\sigma \rho \kappa} x \to q]$, for all $S \in A$. Combining this with our assumption that $[\lambda x.t]_{\beta} \in \cap \{ [[\Pi x : T_1. T_2]_{\sigma \rho \kappa} x \to S] \mid S \in A \}$, we deduce, by the semantics of $t$-types, that $[[\xi E/x]_{\beta}]_{\sigma \rho \kappa} x \to q \subseteq \cap \{ [[T_2]_{\sigma \rho \kappa} x \to S] \mid S \in A \}$. Now we using the IH for the second premise, we obtain the required $[[\xi E/x]_{\beta}]_{\sigma \rho \kappa} x \to q$.

**Case:**

\[
\Gamma \vdash T_1 : \ast \quad \Gamma, x : T_1 \vdash T_2 : \ast \\
\hline
\Gamma \vdash \forall x : T_1, T_2 : \ast
\]

This case is similar to the previous one, so we omit it.

**Case:**

\[
\Gamma \vdash \kappa : \Box \quad \Gamma, y : \kappa \vdash T : \ast \\
\hline
\Gamma \vdash \forall y : \kappa. T : \ast
\]

This case is also similar to the previous two, so we omit it.

**Case:**

\[
\Gamma \vdash T : \ast \quad \Gamma, x : T \vdash T' : \ast \\
\hline
\Gamma \vdash \lambda x : T. T' : \ast
\]

This case follows directly by the IH and the semantics of $t$-types.

**Case:**

\[
\Gamma \vdash T : \ast \quad \Gamma, x : T \vdash T' : \kappa \\
\hline
\Gamma \vdash \lambda x : T. T' : \Pi x : T. \kappa
\]

We must show

\[
\cap_{\forall x : T. \kappa, \sigma_1, \rho_1} \{ [[\lambda x : T. T']_{\sigma \rho \kappa} x \to S] \mid S \in A \} \subseteq \cap_{\forall x : T. \kappa, \sigma_1, \rho_1} \{ [[\lambda x : T. T']_{\sigma \rho \kappa} x \to S] \mid S \in A \}
\]

By the definition of $\subseteq$, it suffices to show the following, for an arbitrary $E \in [[T]_{\sigma_1 \rho_1}$:

\[
\cap_{\forall x : T. \kappa, \sigma_1, \rho_1} \{ [[\lambda x : T. T']_{\sigma \rho \kappa} x \to S] \mid S \in A \} = \cap_{\forall x : T. \kappa, \sigma_1, \rho_1} \{ [[\lambda x : T. T']_{\sigma \rho \kappa} x \to S] \mid S \in A \}
\]

By the definition of $\cap$ for $t$-kinds and the semantics of type-level $\lambda$-abstractions, this is equivalent to

\[
\cap_{\forall x : T. \kappa, \sigma_1, \rho_1} \{ [[T']_{\sigma \rho \kappa} x \to S] \mid S \in A \} \subseteq \cap_{\forall x : T. \kappa, \sigma_1, \rho_1} \{ [[T']_{\sigma \rho \kappa} x \to S] \mid S \in A \}
\]

This now follows by the IH applied to the second premise.

**Case:**

\[
\Gamma \vdash \kappa : \Box \quad \Gamma, x : \kappa \vdash T' : \kappa' \\
\hline
\Gamma \vdash \lambda x : \kappa. T' : \Pi x : \kappa. \kappa'
\]

This case is similar to the previous one, so we omit it.
Case:

\[
\Gamma \vdash T : \Pi X : \kappa \kappa' \quad \Gamma \vdash T' : \kappa
\]

\[
\Gamma \vdash T : T' [X/X'] \kappa
\]

We must show

\[
\cap_{\gamma / X} \Pi_{\kappa, \sigma, \rho_1} \{ \llbracket T' \rrbracket_{\sigma, \rho[X = S]} \mid S \in A \} \subseteq \llbracket T \rrbracket_{\sigma, \rho[X = \gamma]}
\]

By the IH applied to the first premise, we have

\[
\cap_{\gamma / X} \Pi_{\kappa, \sigma, \rho_1} \{ \llbracket T' \rrbracket_{\sigma, \rho[X = S]} \mid S \in A \} \subseteq \Pi_{\kappa, \sigma, \rho_1} \llbracket T' \rrbracket_{\sigma, \rho[X = \gamma]}
\]

We have \[\sigma_{\gamma, \beta} \in \llbracket T' \rrbracket_{\sigma, \rho}\] by the IH part (3) applied to the second premise. Since \[\llbracket T' \rrbracket_{\sigma, \rho_1}\] is defined, we can argue by an easy separate induction that \[\llbracket T' \rrbracket_{\sigma, \rho_1} = \llbracket T' \rrbracket_{\sigma, \rho}\] (similar to Lemma 17). So \[\sigma_{\gamma, \beta} \in \llbracket T' \rrbracket_{\sigma, \rho_1}\]. Then by the semantics of \(\subseteq\) and \(\cap\) for \(\Pi\)-kinds, we obtain the following from the above displayed formula:

\[
\cap_{\gamma, \sigma_{\gamma \rightarrow \eta}, \rho_1} \{ \llbracket T' \rrbracket_{\sigma, \rho[X = S]} \mid (\sigma_{\gamma, \beta}) \mid S \in A \} \subseteq \cap_{\gamma, \sigma_{\gamma \rightarrow \eta}, \rho_1} \llbracket T' \rrbracket_{\sigma, \rho[X = \gamma]}
\]

Now by the semantics of type-level application, this is equivalent to

\[
\cap_{\gamma, \sigma_{\gamma \rightarrow \eta}, \rho_1} \{ \llbracket T' \rrbracket_{\sigma, \rho[X = S]} \mid (\sigma_{\gamma, \beta}) \mid S \in A \} \subseteq \cap_{\gamma, \sigma_{\gamma \rightarrow \eta}, \rho_1} \llbracket T \rrbracket_{\sigma, \rho[X = \gamma]}
\]

By Lemma 19, this is equivalent to what we had to show.

Case:

\[
\Gamma \vdash T : \Pi X : \kappa, \kappa' \quad \Gamma \vdash T' : \kappa
\]

\[
\Gamma \vdash T : T' [X/X'] \kappa
\]

This case is similar to the previous one, so we omit it.

Case:

\[
\Gamma, X : * \vdash t : L_X\kappa
\]

\[
\Gamma \vdash \uparrow_L L : \text{lift}(L)
\]

The type in question has no free type variables, so this case is trivial.

Case:

\[
\Gamma \vdash \kappa' : \square \quad \text{Ctor} \quad \Theta \quad \Gamma, Y : \kappa' \vdash \Theta : *
\]

\[
\Gamma \vdash \uparrow Y : \kappa' \mid \Theta, T : \kappa
\]

Let \(F_S = (S' \in \llbracket \kappa' \rrbracket \rightarrow \llbracket T \rrbracket_{\sigma, \rho[X = S], Y = S'})\). Similarly, let \(G = (S' \in \llbracket \kappa' \rrbracket \rightarrow \llbracket T \rrbracket_{\sigma, \rho[X = S], Y = S'})\).

We must show

\[
\cap_{\kappa', \sigma, \rho_1} \{ \cap_{\kappa', \sigma, \rho} \{ F_S^n(\Pi_{\kappa', \sigma, \rho}[X = S]) \mid n \in \mathbb{N} \} \mid S \in A \} \subseteq \cap_{\kappa', \sigma, \rho_1} \{ G^n(\Pi_{\kappa', \sigma, \rho}[X = S]) \mid n \in \mathbb{N} \}
\]

Since \[\llbracket \kappa' \rrbracket_{\sigma, \rho_1}\] is defined, we may use Lemma 22 to help reorganize the intersections. This takes us to the following equivalent goal:

\[
\cap_{\kappa', \sigma, \rho_1} \{ \cap_{\kappa', \sigma, \rho} \{ F_S^n(\Pi_{\kappa', \sigma, \rho}[X = S]) \mid S \in A \} \mid n \in \mathbb{N} \} \subseteq \cap_{\kappa', \sigma, \rho_1} \{ G^n(\Pi_{\kappa', \sigma, \rho}[X = S]) \mid n \in \mathbb{N} \}
\]

This may be proved by showing

\[
\cap_{\kappa', \sigma, \rho} \{ F_S^n(\Pi_{\kappa', \sigma, \rho}[X = S]) \mid S \in A \} \subseteq \cap_{\kappa', \sigma, \rho_1} G^n(\Pi_{\kappa', \sigma, \rho}[X = S])
\]
by inner induction on \( n \in \mathbb{N} \). For the base case, \( T_{\kappa',\sigma,\rho[x \rightarrow S]} \) equals \( T_{\kappa',\sigma,\rho} \), by Lemma 22, and in turn equals \( T_{\kappa',\sigma,\rho[x \rightarrow \varnothing]} \). For the step case, we just apply the outer IH to the fifth premise of the kindling derivation, making use of the inner IH.

**Case:**

\[
\Gamma \vdash \star: \square
\]

We have \( \cap \{ \llbracket \star \rrbracket_{\sigma,\rho[x \rightarrow S]} \mid S \in A \} = \cap \{ \llbracket S' \rrbracket \} = \llbracket \star \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \).

**Case:**

\[
\Gamma \vdash T: \star \quad \Gamma, x: T \vdash \kappa: \square
\]

\[
\Gamma \vdash \Pi x : T. \kappa : \square
\]

We must show
\[
\cap \{ \llbracket \Pi x : T. \kappa \rrbracket_{\sigma,\rho[x \rightarrow S]} \mid S \in A \} \subseteq \llbracket \Pi x : T. \kappa \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]}
\]

So assume an arbitrary \( S' \in \cap \{ \llbracket \Pi x : T. \kappa \rrbracket_{\sigma,\rho[x \rightarrow S]} \mid S \in A \} \), and show \( S' \in \llbracket \Pi x : T. \kappa \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \). For the latter, it suffices to assume an arbitrary \( E \in \llbracket T \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \), and show \( S'(E) \in \llbracket \kappa \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \).

By the IH part (7ai), we also have \( E \in \llbracket T \rrbracket_{\sigma,\rho[x \rightarrow S]} \) for all \( S \in A \), because \( q \subseteq \kappa, \sigma, \rho, S \) for all \( S \in A \). So \( E \in \cap \{ \llbracket T \rrbracket_{\sigma,\rho[x \rightarrow S]} \mid S \in A \} \) (recall that \( A \neq \emptyset \)). This gives us \( S'(E) \in \cap \{ \llbracket \kappa \rrbracket_{\sigma[x \rightarrow \xi(X)]}, \rho[x \rightarrow \varnothing] \mid S \in A \} \). Now by the IH, \( S'(E) \in \llbracket \kappa \rrbracket_{\sigma[x \rightarrow \xi(X)]}, \rho[x \rightarrow \varnothing] \), as required.

**Case:**

\[
\Gamma \vdash \kappa : \square \quad \Gamma, X : \kappa \vdash \kappa' : \square
\]

\[
\Gamma \vdash \Pi x : \kappa. \kappa' : \square
\]

The proof in this case is similar, so we omit it.

**Proof of part (8)**

It suffices to prove that if \( A \subseteq \llbracket \kappa \mid \Theta \rrbracket \), then \( \llbracket \Theta \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \) holds, since by Lemma 20, \( \llbracket \kappa \rrbracket_{\sigma,\rho} \subseteq \kappa, \sigma, \rho \cap \llbracket \kappa, \sigma, \rho \rrbracket \) is a complete lattice (so we already have \( \cap \{ \kappa, \sigma, \rho A \} \in \llbracket \kappa \rrbracket_{\sigma,\rho} \)). Let \( q \) denote \( \cap \{ q \mid \kappa, \sigma, \rho A \} \). By definition of \( q \) and basic lattice properties, we have \( q \subseteq \kappa, \sigma, \rho, S \) for all \( S \in A \). If \( A \) is empty, then \( q = \top \kappa \) by definition of \( \cap \{ \kappa, \sigma, \rho \} \), and then we have \( \llbracket \Theta \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \) by one of the assumptions for this part of the theorem. So for the rest of this part of the proof, we assume \( A \) is not empty. To show \( \llbracket \Theta \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \), we proceed by inner induction on the derivation of \( \text{Ctors}_X \Theta \). If \( \Theta \) is empty, then \( \llbracket \Theta \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \) holds trivially.

**Case:**

\[
\text{Ctor}_T \quad \text{Ctors}_X \Theta'
\]

\[
\text{Ctors}_X \quad \text{Ctors}_X \quad \text{Ctors}_X (t \in T, \Theta')
\]

So \( \Theta = (t \in T, \Theta') \). The IH gives us \( \llbracket \Theta \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \), so we must just prove that the constraint \( t \in T \) is satisfied. From our assumption that \( A \subseteq \llbracket \kappa \mid \Theta \rrbracket_{\sigma,\rho} \), we have \( \{ \sigma \mid t \in A \} \subseteq \llbracket \kappa \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]} \) (since \( t \in T \) is one of the constraints in \( \Theta \)). So it suffices to prove
\[
\cap \{ \llbracket T \rrbracket_{\sigma,\rho[x \rightarrow S]} \mid S \in A \} \subseteq \llbracket \Theta \rrbracket_{\sigma,\rho[x \rightarrow \varnothing]}
\]

Note that we can invert our derivation of \( \Gamma \vdash \Theta : \star \) to obtain a smaller derivation of \( \Gamma \vdash T : \star \).

We proceed now by a further inner induction on the derivation of \( \text{Ctor}_T \).
Case: $X \in^+ T_1 \quad \text{CtorTp}_X \quad T_2$

\[
\frac{\text{CtorTp}_X \quad \Pi x : T_1, T_2}{\text{CtorTp}_X \quad \Pi x : T_1, T_2}
\]

So $T$ is $\Pi x : T_1, T_2$. By inversion on our derivation of $\Gamma \vdash T : \ast$, we see it must be by the corresponding $\Pi$-kinding rule. (This is because the kinding derivation is assumed normalized, and hence cannot end in a conversion by $\ast \simeq \ast$.) So we have smaller derivations of $\Gamma \vdash T_1 : \ast$ and $\Gamma, x : T_1 \vdash T_2 : \ast$. Assume now arbitrary $[\lambda x. t]_{c, \beta} \in \bigcap_{\ast, \sigma, \rho} \{[\Pi x : T_1, T_2]_{\sigma, \rho [x \leftarrow S]} \mid S \in A\}$, and show $[\lambda x. t]_{c, \beta} \in [\Pi x : T_1, T_2]_{\sigma, \rho [x \leftarrow q]}$. For the latter, it suffices to assume arbitrary $E \in [T_1]_{\sigma, \rho [x \leftarrow q]}$, and show $[[\zeta(E)/x]_{c, \beta} \in [T_2]_{\sigma, \rho [x \leftarrow \zeta(E)] [x \leftarrow q]}$. Since $q \subseteq_{\sigma, \rho} S$, for all $S \in A$ (as noted above), we may apply the IH part (7ai) to conclude that for all $S \in A$, $[T_1]_{\sigma, \rho [x \leftarrow q]} \subseteq [T_1]_{\sigma, \rho [x \leftarrow S]}$. Since $A$ is nonempty, this implies that $[T_1]_{\sigma, \rho [x \leftarrow q]} \subseteq \bigcap_{\ast, \sigma, \rho} \{[T_1]_{\sigma, \rho [x \leftarrow S]} \mid S \in A\}$. So from our assumption that $[\lambda x. t]_{c, \beta} \in \bigcap_{\ast, \sigma, \rho} \{[\Pi x : T_1, T_2]_{\sigma, \rho [x \leftarrow S]} \mid S \in A\}$ and the semantics of $\Pi$-types, we can now deduce

$[[\zeta(E)/x]_{c, \beta} \in [T_2]_{\sigma, \rho [x \leftarrow \zeta(E)] [x \leftarrow q]}$ as required.

Case: $\text{HeadOnly}_X \quad T$

We proceed by inner induction on the derivation of $\text{HeadOnly}_X \quad T$ to show that in this case, with a structurally smaller derivation of $\Gamma \vdash T : \kappa'$, we have $\bigcap_{\kappa', \sigma, \rho} \{[T]_{\sigma, \rho [x \leftarrow S]} \mid S \in A\} = [T]_{\sigma, \rho [x \leftarrow q]}$.

Case: $\text{HeadOnly}_X \quad X$

We must have $\kappa = \kappa'$ in this case, and we then deduce $\bigcap_{\kappa, \sigma, \rho} \{[X]_{\sigma, \rho [x \leftarrow S]} \mid S \in A\} = \bigcap_{\kappa, \sigma, \rho} \{S \mid S \in A\} = [X]_{\sigma, \rho [x \leftarrow q]}$.

Case: $X \notin \text{FV}(T)$

In this case, by Lemma 17, $\bigcap_{\kappa', \sigma, \rho} \{[T]_{\sigma, \rho [x \leftarrow S]} \mid S \in A\} = \bigcap_{\kappa', \sigma, \rho} \{[T]_{\sigma, \rho} \mid S \in A\} = [T]_{\sigma, \rho} = [T]_{\sigma, \rho [x \leftarrow q]}$.

Case: $\text{HeadOnly}_X \quad T \quad X \notin \text{FV}(t)$

Inversion on our derivation of $\Gamma \vdash T : \kappa'$ gives us:

Case: $\Gamma \vdash T : \Pi x : T', \kappa'' \quad \Gamma \vdash t : T'$

\[
\frac{\Gamma \vdash t : \Pi x : T', \kappa'' \quad \Gamma \vdash t : T'}{\Gamma \vdash t \vdash [t/x]_{\kappa''}}
\]
56

**Stump**

By our inner IH applied to the derivation of HeadOnly\(_X\) \(T\) and the first premise of the above kinding inference, we have \(\cap \Pi \in T : \kappa', \sigma, \rho \{ [T]_{\sigma, \rho[X \rightarrow S]} \mid S \in A \} = [T]_{\sigma, \rho[X \rightarrow q]} \). Since \(A\) is nonempty, the definition of \(\cap \Pi \in T : \kappa', \sigma, \rho \) gives us:

\[
(E \in [T]_{\sigma, \rho} \rightarrow \cap \kappa', \sigma, \chi(E), \rho ([T]_{\sigma, \rho[X \rightarrow S]}(E) \mid S \in A)) = [T]_{\sigma, \rho[X \rightarrow q]}
\]

By the semantics, \([T t]_{\sigma, \rho[X \rightarrow q]} = [T]_{\sigma, \rho[X \rightarrow q]}([\sigma t]_{\chi})\). Using the above equation, we then have

\[
\cap \kappa', \sigma, \chi([\sigma t]_{\chi}), \rho ([T]_{\sigma, \rho[X \rightarrow S]}([\sigma t]_{\chi}) \mid S \in A)) = [T]_{\sigma, \rho[X \rightarrow q]}([\sigma t]_{\chi})
\]

By the semantics, this is equivalent to

\[
\cap \kappa', \sigma, \chi([\sigma t]_{\chi}), \rho ([T t]_{\sigma, \rho[X \rightarrow S]} \mid S \in A)) = [T t]_{\sigma, \rho[X \rightarrow q]}
\]

Lemma 21 tells us this is equivalent to the following, which is what we had to prove:

\[
\cap \kappa', \sigma, \chi([T t]_{\sigma, \rho[X \rightarrow S]} \mid S \in A)) = [T t]_{\sigma, \rho[X \rightarrow q]}
\]

**Case:**

\[
\Gamma \vdash T : \Pi X : \kappa, \kappa' \quad \Gamma \vdash T' : \kappa
\]

\[
\Gamma \vdash T T' : [T'/X]_{\kappa'}
\]

This case is similar to the previous one, so we omit it.

\(\square\)