Using the Hereditary Substitution Function in Normalization Proofs

Harley Eades and Aaron Stump

Abstract

This paper summarizes the use of the hereditary substitution function due to K. Watkins et al., in proofs of normalization for typed λ-calculi. We prove normalization for three different type theories. The first system analyzed is the Simply Typed λ-Calculus (STLC). Then we extend STLC with predicative polymorphism called Stratified System F (SSF) which was studied by D. Leivant. Finally, we show how to prove normalization of Stratified System F$^{\omega}$ due to J. Girard, by using the normalization results for STLC and SSF. We also point out several new properties of the hereditary substitution function and a critical general property, semantic inversion, which is required for the normalization proof method using hereditary substitution.

1 Introduction

A functional programming language is one that is based on a mathematical foundation rather than being based on the structure of a machine. This allows one to reason formally about the language itself. The mathematical foundation we speak of is the λ-calculus, a formal language defined by A. Church [2]. Originally, Church defined the λ-calculus to be used in his research on the foundations of mathematics, but it was later discovered that it can be used as the basis of programming languages. In fact it so happens that the λ-calculus model's computation quite well. In fact it is Turing complete. The main idea is that all computation can be defined in terms of only function application. The syntax of the λ-calculus is defined in the following definition.

Definition 1. *The language of the λ-calculus consists of only variables, functions, and applications. The grammar is as follows:*

$$t \quad ::= \quad x \quad | \quad \lambda x.t \quad | \quad t\,t$$

We call expressions in the λ-calculus terms which can be considered as programs. We denote the set of all terms (programs) definable in the λ-calculus as $\mathscr{T}$. Functions are denoted by $\lambda x.t$ where x is the argument of the function and t is the body of the function. Keep in mind that arguments to functions do not have to be used by the body of the function. Function application is denoted by $t_1\ t_2$. For example, if we want to apply the function $\lambda x.x$ to an argument t we denote this as $(\lambda x.x)\ t$. We will see below that application will amount to replacing the argument to the function with the term the function is being applied to in the body of the function. Now not all applications consist of a function being applied to an argument. For example, $x\ t$ is a valid term in the λ-calculus.

We now define some useful functions and terminology. A variable x is defined to to be free in a term t if and only if $x \in FV(t)$ where the following definition defines the function FV.

Definition 2. *The set of free variables present in a term is constructed by the meta-level function $FV : \mathscr{T} \to \mathscr{P}(\mathscr{T})$, where $\mathscr{P}$ denotes the power set function. We define FV by induction on the form of its argument, where $-$ denotes set theoretic difference.*

$$\begin{array}{lcl} FV(x) & = & x \\ FV(\lambda x.t) & = & FV(t) - \{x\} \\ FV(t_1\ t_2) & = & FV(t_1) \cup FV(t_2). \end{array}$$

We do not redefine FV for each language we analyze below. The definition is a natural extension of the definition above. A variable x is consider bound in a term t if and only if $x \notin FV(t)$. With respect to the λ-calculus all bound variables are arguments to functions. Next we define when two terms are considered syntactically equivalent.

Definition 3. *The meta-level infix function* $\equiv: \mathscr{T} \times \mathscr{T} \to Bool$ *equates two terms that are syntactically equal. It is defined as follows:*

$\lambda x.t \equiv \lambda y.t'$
When $x \equiv y$ *and* $t \equiv t'$.

$t_1\ t_2 \equiv t'_1\ t'_2$
When $t_1 \equiv t'_1$ *and* $t_2 \equiv t'_2$.

$t \equiv t$

$t' \equiv t$
When $t \equiv t'$.

$t_1 \equiv t_3$
When $t_1 \equiv t_2$ *and* $t_2 = t_3$.

Note that the only thing a variable is syntactically equal to is itself. Next we define when a term is a subterm of another.

Definition 4. *We say a term* t *is a subterm of a term* t' *if and only if either*

i. $t \equiv t'$,

ii. if $t' \equiv \lambda x : \phi.t''$ *and* t *is a subterm of* t', *or*

iii. if $t' \equiv t'_1\ t'_2$ *and* t *is a subterm of* t'_1 *or* t'_2.

Similarly to the other definitions above we do not redefine this notion for the other languages considered in this paper, because it has a natural extension to those languages.

Lets consider some examples of interesting terms in this language.

Example 1. *Here are a few interesting terms one can define in the* λ*-calculus:*

Identity Function:	$\lambda x.x$
Squaring Function:	$\lambda x.x\ x$
True:	$\lambda x.\lambda y.x$
False:	$\lambda x.\lambda y.y$
Conjunction:	$\lambda x.\lambda y.x\ y\ x$
Disjunction:	$\lambda x.\lambda y.x\ x\ y$
Zero:	$\lambda s.\lambda z.z$
One:	$\lambda s.\lambda z.s\ z$
Plus:	$\lambda n_1.\lambda n_2.\lambda s.\lambda z.n_1\ s\ (n_2\ s\ z)$
Multiplication:	$\lambda n_1.\lambda n_2.\lambda s.\lambda z.n_2\ (plus\ n_1)\ z$

The examples above show how one can define all of the natural numbers and some logical operations as functions. In fact everything we define in the λ-calculus is a function, because that is all we have to work with. Hence, the name functional programming. The previous list is far from complete. The λ-calculus is surprisingly expressive.

We can define a lot of interesting terms in this language, but the language does not give us any way of actually carrying out computation. For this we must define what is called the operational semantics. The operational semantics defines the computational meaning for a programming language. That is it tells us exactly how to compute using the constructs of the language. The definition of the operational semantics depends on a meta-level function called capture-avoiding substitution which is defined next.

Definition 5. *Capture-avoiding substitution is a meta-level function defined to replace a variable in some term with another. We denote this function by* $[t/x]t'$ *which is read as substitute* t *for* x *in* t' *and it has the type* $\mathscr{T} \times \mathscr{T} \times \mathscr{T} \to \mathscr{T}$. *It is defined by induction on the form of* t' *the term we are substituting into.*

$[t/x]x = t$

$[t/x]y = y$

$[t/x](\lambda x.t') = \lambda x.t'$

$[t/x](\lambda y.t') = \lambda y.[t/x]t'$
Where $y \notin FV(t)$.

$[t/x](\lambda y.t') = \lambda y.[([z/y]t)/x]t'$
Where $y \in FV(t)$ and z is a variable distinct from all variables (free or bound) in t.

$[t/x](t_1\ t_2) = ([t/x]t_1)\ ([t/x]t_2)$

Note that if $x \notin FV(t')$ then $[t/x]t' = t'$. We call the substitution function capture avoiding, because if we substitute a term t for x in t' when a free variable in t has the same name as a bound variable in t' then the free variable remains free after the substitution. Lets make this concrete by an example.

Example 2. *Suppose $t \equiv y$ and $t' \equiv \lambda y.y\ x$. Then $[t/x]t' = \lambda y.y\ z$, because the substitution function by definition is forced to rename y to z for some variable z distinct from all variables in t. Now suppose the function did not do this renaming. Then $[t/x]t' = \lambda y.y\ y$ and the variable y is captured by the binder λy which is not desirable.*

We are now in a position to define the operational semantics of the λ-calculus.

Definition 6. *The operational semantics for the λ-calculus is the following:*

$$(\lambda x.t)\ t' \rightsquigarrow [t'/x]t$$

This is called full β-reduction and the rule above is called the β-rule. The left-hand side of the rule above is called a β-reducible expression or β-redex and the right-hand side is called the contractum. This rule tells us that anywhere one sees a subterm matching the pattern $(\lambda x.t)\ t'$ one may replace that subterm with the result of $[t'/x]t$. That is all redexes may be replaced with their corresponding contractions in any order. This implies that full β-reduction is a non-deterministic operational semantics. We call a term with no redexes as subterms a normal form unless otherwise stated. We denote the reflexive-transitive closure of $\rightsquigarrow$ as $\rightsquigarrow^*$ and say t reduces to t' in zero or more steps if and only if $t \rightsquigarrow^* t'$.

Full β-reduction gives us everything we need to carry out computation. Next we give a few example computations using the terms we defined in Example 1.

Example 3. *The first example shows how the identity function can be applied to an argument:*

$$(\lambda x.x)\ y \rightsquigarrow y$$

We can see that we do not write down $[y/x]x$ and say $[y/x]x \rightsquigarrow y$, because the substitution is a meta-level function. We simply carry out the substitution.

Now lets consider a more complex example, where we compute $3 + 2 = 5$ using the definition of 3, 2, and the plus function from above. First, let $3 = \lambda s.\lambda z.s\ (s\ (s\ z))$, $2 = \lambda s.\lambda z.s\ (s\ z)$, and $plus = \lambda n_1.\lambda n_2.\lambda s.\lambda z.n_1\ s\ (n_2\ s\ z)$. Then

$$\begin{array}{lll}
(plus\ 3)\ 2 & \equiv & ((\lambda n_1.\lambda n_2.\lambda s.\lambda z.n_1\ s\ (n_2\ s\ z))\ 3)\ 2\\
& \rightsquigarrow_\beta & (\lambda n_2.\lambda s.\lambda z.3\ s\ (n_2\ s\ z))\ 2\\
& \rightsquigarrow_\beta & \lambda s.\lambda z.3\ s\ (2\ s\ z)\\
& \equiv & \lambda s.\lambda z.(\lambda s.\lambda z.s\ (s\ (s\ z)))\ s\ (2\ s\ z)\\
& \rightsquigarrow_\beta & \lambda s.\lambda z.(\lambda z.s\ (s\ (s\ z)))\ (2\ s\ z)\\
& \equiv & \lambda s.\lambda z.(\lambda z.s\ (s\ (s\ z)))\ ((\lambda s.\lambda z.s\ (s\ z))\ s\ z)\\
& \rightsquigarrow_\beta & \lambda s.\lambda z.(\lambda z.s\ (s\ (s\ z)))\ ((\lambda z.s\ (s\ z))\ z)\\
& \rightsquigarrow_\beta & \lambda s.\lambda z.(\lambda z.s\ (s\ (s\ z)))\ (s\ (s\ z))\\
& \rightsquigarrow_\beta & \lambda s.\lambda z.s\ (s\ (s\ (s\ (s\ z))))\\
& \equiv & 5.
\end{array}$$

Each of the example computations above terminate. One might be wondering if one can write down an infinite loop in the λ-calculus. It turns out we can. Consider the following example.

Example 4. *The following computation does not terminate:*

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \rightsquigarrow_\beta (\lambda x.x\ x)\ (\lambda x.x\ x) \rightsquigarrow_\beta \cdots .$$

Terms like in Example 4 are called diverging terms. The presence of divergence in the λ-calculus is good and bad. It depends on what the language is being used for. If the language is being used for general purpose programming then the presence of infinite loops is expected, but if the language is being used to formalizing mathematics or being used for logical reasoning then divergence is very bad. Church initially thought he could use the λ-calculus for logical reasoning, but S. Kleene and J. Rosser were able to define Russell's paradox in the λ-calculus which shows that it is inconsistent as a logic. In response to this Church defined a typed version of the λ-calculus called The Simply Typed λ-calculus. Where types are added to the language of the λ-calculus to prevent paradoxes like Russell's. We will see in Section 3 how a type λ-calculi can be considered a logic.

2 The Simply Typed λ-Calculus

The Simply Typed λ-Calculus adds the notion of a type to the λ-calculus. Types play many roles. When the language was first defined a type was used to rule out paradoxes from the λ-calculus. With respect to programming language types play two important roles. The first role is to describe the type of data being operated on. As we will see it can also be used as a correspondence to formulas of a logic. The former role can be used to rule out by type-checking (a process we will describe next) the application of programs to incorrect data. For example, applying the logical negation operator to the natural number forty two. In this paper we will only concern ourselves with the later role. The syntax of the language is defined by the following definition.

Definition 7. *The language of STLC consists of the same term syntax as the λ-calculus and a new syntactic category for the syntax of types. The syntactic category Γ describes the syntax for contexts (also known as environments) whose purpose will be to keep track of the types of free variables. How Γ is used should become more clear after the reader sees the full definition of the language. To insure substitutions over contexts behave in an expected manner we rename variables as necessary to ensure contexts have at most one declaration per variable.*

$$\begin{array}{lclclcl} t & ::= & x & \mid & \lambda x:\phi.t & \mid & t\,t \\ \phi & ::= & X & \mid & \phi \to \phi & & \\ \Gamma & ::= & \cdot & \mid & \Gamma,\, x:\phi & & \end{array}$$

Definition 8. *The following set of inference style rules define a decidable algorithm for testing if a term t has a given type ϕ in context Γ and is denoted $\Gamma \vdash t : \phi$, where Γ, t, and ϕ are inputs and there is no output.*

$$\frac{\Gamma(x) = \phi}{\Gamma \vdash x : \phi}\ \textsc{Var} \qquad \frac{\Gamma, x:\phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x:\phi_1.t : \phi_1 \to \phi_2}\ \textsc{Lam} \qquad \frac{\Gamma \vdash t_1 : \phi_1 \to \phi_2 \qquad \Gamma \vdash t_2 : \phi_1}{\Gamma \vdash t_1\ t_2 : \phi_2}\ \textsc{App}$$

Type checking is the process of deciding whether or not a give term t can be assigned a given type T within some context Γ. This process begins with a desired conclusion of the form $\Gamma \vdash t : \phi$ and tries to use the previously defined type-checking rules to build a derivation in a bottom up fashion. The process terminates when no more rules can be applied. That is we always end with some axiom(s). If the process does not terminate at only axioms then the term in question cannot be assigned the given type. Lets consider a few examples.

Example 5. *First lets try and type-check $\cdot \vdash \lambda s : X \to X.\lambda z : X.s\ z : (X \to X) \to X \to X$.*

$$\cfrac{\cfrac{\cfrac{\cfrac{}{s : X \to X, z : X \vdash s : X \to X}\ \textsc{Var} \qquad \cfrac{}{s : X \to X, z : X \vdash z : X}\ \textsc{Var}}{s : X \to X, z : X \vdash s\ z : X}\ \textsc{App}}{s : X \to X \vdash \lambda z : X.s\ z : X \to X}\ \textsc{Lam}}{\cdot \vdash \lambda s : X \to X.\lambda z : X.s\ z : (X \to X) \to X \to X}\ \textsc{Lam}$$

We can see that the previous derivation terminates with all axioms the Var type-checking rule. So lets consider trying to type a term that is not typeable in STLC.

$$\dfrac{\dfrac{}{x : X \to X \vdash x : X \to X}\ \textsc{Var} \qquad \dfrac{}{x : X \to X \vdash x : X}\ ???}{\dfrac{x : X \to X \vdash x\ x : X \to X}{\cdot \vdash \lambda x : X \to X.x\ x : X \to X}\ \textsc{Lam}}\ \textsc{App}$$

We can see in the previous darivation that we get stuck. We are not able to give the variable x two different types. This latter example also shows that the diverging term $(\lambda x.x\ x)\ (\lambda x.x\ x)$ is not typeable in STLC. We will see that it is actually impossible to give any diverging term a type in STLC. That is all terms of STLC are terminating programs on all input. This example also shows that when we restrict $\mathscr{T}$ to only typeable terms in STLC it is a strict subset of the set of all terms definable in the λ-calculus.

At the beginning of this section we defined the roles of a type and one such role is that a type corresponds to a formula of logic. In the next section we make this role precise by showing how STLC can be considered as intuitionistic propositional logic where types are formulas of the logic and terms (programs) are proofs of formulas.

3 Typed λ-Calculi as Logics

In this section we describe how typed λ-calculi like STLC can be considered as intuitionistic logics. Now it is possible to define a typed λ-calculi for classical logic, but in this paper we only concern ourselves with intuitionistic logic. We show how STLC corresponds to propositional intuitionistic logic.

3.1 A Bit about Intuitionism

Before revealing the details behind the correspondence between intuitionistic logic and type theory we first take a brief look at propositional intuitionistic logic with only implication. By type theory we mean a typed λ-calculus like STLC defined above.

3.2 Natural Deduction for Intuitionistic Propositional Logic

We now present a slight variation of Gentzen's natural deduction for intuitionistic propositional logic defined in [13]. Our version has a less complicated syntax and only consists of implication. The following definition gives the syntax for formulas and defines the inference rules.

Definition 9. *We denote propositional variables by x, x_i, y, and so on. We assume an infinite number of them. All formulas will be denoted by ϕ_i. We denote sets of assumptions by Γ_i.*

$$\phi \quad ::= \quad x \quad | \quad \phi_1 \to \phi_2$$

$$\dfrac{}{\phi \vdash \phi}\ \mathrm{I} \qquad \dfrac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \to \phi_2}\ \to_i \qquad \dfrac{\Gamma_1 \vdash \phi_1 \to \phi_2 \qquad \Gamma_2 \vdash \phi_1}{\Gamma_1, \Gamma_2 \vdash \phi_2}\ \to_e$$

In intuitionistic logic also known as constructive logic the only provable formulas are ones whose proofs are computable. That is we can actually construct the proof. This is contrary to classical logic where we consider existence proofs to be valid means of establishing the validity of an existentially quantified formula. In such a proof we usually do not have to give an exact algorithm for constructing the "thing" we are showing to exist. In intuitionistic logic we do. Brouwer, Heyting, and Kolmogorov constructed an interpretation called the BHK-interpretation which describes an algorithm for actually constructing a witness (proof) of a formula. This interpretation can be used to interpret proofs of formulas in intuitionistic logic as programs in a programming language. We define a simplified version of the interpretation in the following definition. This definition is due to Mints [13]. It uses λ-notation for function definitions.

Definition 10. *The BHK-interpretation:*

$$c\,r\,(\phi_1 \to \phi_2) \quad \textit{iff} \quad \begin{array}{l} c \textit{ is a function, } \lambda x.t, \textit{ such that for any } d\,r\,\phi_1 \\ (\lambda x.t)\,d\,r\,\phi_2. \end{array}$$

We say a construction c realizes ϕ iff $c\,r\,\phi$.

An example use of the BHK-interpretation is $(\lambda x.x)\,r\,(\phi \to \phi)$. That is a proof of $\phi \to \phi$ is just the identity function. According to the BHK-interpretation all we need to construct proofs of formulas in intuitionistic propositional logic with only implication are functions and function applications. Well that is all STLC consists of. We will see that STLC is exactly intuitionistic propositional logic with only implication. In fact there is a one-to-one correspondence between them.

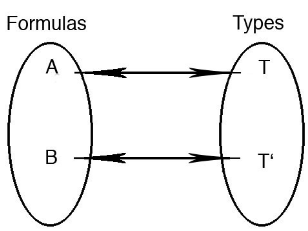
3.3 Propositions-as-Types

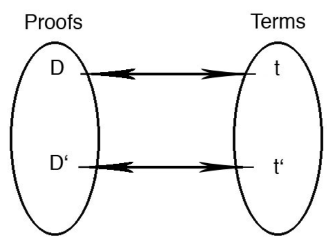
Through the work of Curry, Howard, Tait, Laüchli, De Bruijn and Prawitz there exists an important correspondence between type theory and intuitionistic logic [9]. It is usually stated as

Propositions = Types and Proofs = Programs.

I am going to briefly explain this correspondence by showing how STLC is isomorphic to a simplified version of Gentzen's natural deduction, that is, propositional intuitionistic logic with only implication [13, 25]. This presentation is based on chapters 4 and 5 of [13].

If one removes all of the terms from the type-checking rules defined in Definition 8 then one will end up with essentially the rules of natural deduction defined in Definition 9. This relationship was no accident, in fact this is part of what is know as the proposition-as-types and proofs-as-programs correspondence as mentioned in the introduction of this section. According to this correspondence types denote formulas and terms denote their proofs. This relationship is depicted by the following images.





Now this is not just a simple mapping it is in fact an isomorphism. Types and formulas can be consider equivalent up to isomorphism and the same goes for terms and natural deductions. It is somewhat simple to see that the types and formulas correspond, but it is not so clear how terms and proofs correspond. Lets take a closer look at this now.

The central idea comes from the BHK-interpretation which was defined in Definition 10. Recall this interpretation tells us exactly how to construct a proof of a formula using only functions and function application. We explain how this works through an example. It is out of the scope of this paper to prove this correspondence formally. Consider the following natural deduction proof of syllogism i.e. $(p \to q) \to (q \to u) \to (p \to u)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(p \to q), (q \to u), p \models p \\ (p \to q), (q \to u), p \models p \to q\end{array}}{(p \to q), (q \to u), p \models q} \to_e \qquad (p \to q), (q \to u), p \models q \to u}{(p \to q), (q \to u), p \models u} \to_e}{(p \to q), (q \to u) \models p \to u} \to_i}{(p \to q) \models (q \to u) \to (p \to u)} \to_i}{\cdot \models (p \to q) \to (q \to u) \to (p \to u)} \to_i$$

The next step is to use the BHK-interpretation to find a construction that realizes the formula in question, that is, syllogism. According to the interpretation the construction should be a function when given a construction of $p \to q$ and $q \to u$ returns a construction of $p \to u$. We then obtain $\lambda p : (P \to Q).\lambda q : (Q \to U).\lambda z : P.q\,(p\,z)$ for the construction which realizes syllogism. The natural deduction proof above then corresponds to the type-checking derivation for the following judgment $\cdot \vdash \lambda p : (P \to Q).\lambda q : (Q \to U).\lambda z : P.q\,(p\,z) : (P \to Q) \to (Q \to U) \to (P \to U)$. The derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} p : (P \to Q), q : (Q \to U), z; P \vdash z : P \\ p : (P \to Q), q : (Q \to U), z; P \vdash p : (P \to Q) \end{array}}{p : (P \to Q), q : (Q \to U), z; P \vdash p\,z : Q}\ \textsc{App} \quad p : (P \to Q), q : (Q \to U), z; P \vdash q : Q \to U}{p : (P \to Q), q : (Q \to U), z : P \vdash q\,(p\,z) : U}\ \textsc{App}}{p : (P \to Q), q : (Q \to U) \vdash \lambda z : P.q\,(p\,z) : (P \to U)}\ \textsc{Lam}}{p : (P \to Q) \vdash \lambda q : (Q \to U).\lambda z : P.q\,(p\,z) : (Q \to U) \to (P \to U)}\ \textsc{Lam}}{\cdot \vdash \lambda p : (P \to Q).\lambda q : (Q \to U).\lambda z : P.q\,(p\,z) : (P \to Q) \to (Q \to U) \to (P \to U)}\ \textsc{Lam}$$

We have seen that it is possible for types to correspond to formulas of a logic and programs (terms) to correspond to their proofs. Now the previous type-checking derivation of syllogism parallels that of the natural deduction derivation. This is not a coincidence and is true in general. In fact when type-checking succeeds we are assured that the proof of whatever formula one is trying to prove is a correct proof. The correspondence between natural deduction and type-checking is the driving force behind machine checked proofs and proof assistants like Coq and Agda. The correspondences discussed in this section hold for all the type λ-calculi discussed within this paper.

3.4 The Normalization Property

There is one important property that a type theory being used as a logic must have, and that is that all programs must terminate with some normal form. That is for any input there must exist some computation path (with respect to the languages operational semantics) that terminates with a normal form. We do not require every computational path to terminate. We just need to know there exists one. This property is known as weak normalization or just normalization for short. If there is a weak version of normalization what does it mean to have strong normalization. This just means that all computation paths terminate with a normal form. In fact it is possible to prove that STLC has the strong normalization property. Of course strong normalization implies weak normalization trivially. In this paper we only concern ourselves with weak normalization. Now logics really should have strong normalization as a property, but weak normalization seems to be enough to carry out proofs of logical formulas especially when we do not plan to actually run proofs at runtime, but this is out of the scope of this paper.

The normalization property is important, because mathematical proofs must be finite and total. What truth would a diverging computation establish? It could not establish any truth. The normalization property insures that all proofs terminate and thus all functions are total. Now this is not a trivial property it must be proved. The proof of this property is a meta-level proof, because this property is expressed about a theory and thus cannot be proved within the theory. The collection of these types of properties and their proofs are called the meta-theory of a theory. Here our theories are all functional programming languages which correspond to logics. Normalization proofs are often complicated and very detailed requiring many auxiliary results. We describe how to carry out such a proof throughout the reminder of this paper using a proof technique called proof by hereditary substitution (also known as the hereditary substitutions method).

4 The Hereditary Substitution Function

Proving normalization of a typed λ-calculi using hereditary substitution first requires us to understand a function first defined in [26] by K. Watkins et al., called the hereditary substitution function.

4.1 The Hereditary Substitution Function

In [26] K. Watkins et al. define a dependently typed λ-calculus called the canonical logical framework (CLF). It is called canonical because the entire language consists of only normal forms. In fact the syntactic category describing the syntax for terms only allows normal forms. That is you cannot even write down a non-normal form! Now restricting the language in this way requires a very substantial change to the type theory at the meta-level. Lets consider an example, $(\lambda x : X \to X.x\ y)\ (\lambda x : X.x) \rightsquigarrow [(\lambda x : X.x)/x](x\ y)$, where y is some free variable of type X. Notice that the two terms in the previous application are normal forms, but the capture avoiding substitution results in a non-normal form. In CLF this will not do, because the result of the substitution is not a valid term syntactically. To remedy this problem K. Watkins came up with a really good idea, which is to replace the capture avoiding substitution function with an equivalent function except that when a new redex is created as a result of a substitution that redex is automatically reduced. This new function is called the hereditary substitution function. Using our previous example, when using the hereditary substitution function instead of capture avoiding substitution $[(\lambda x : X.x)/x](x\ y) = y$ instead of $(\lambda x : X.x)\ y$. The hereditary substitution function has one important property that CLF requires, which is, if one applies the function to normal forms then the result of the function is a normal form. This will be made precise and proved for each system considered in this paper.

To better understand how the hereditary substitution works lets define the function for STLC. The definition of the hereditary substitution function depends on an auxillary partial function called $ctype_\phi$ which stands for "construct type". It is defined by the following definition.

Definition 11. *The $ctype_\phi$ function is defined with respect to a fixed type ϕ and has two arguments, a free variable x, and a term t where x may be free in t. We define $ctype_\phi$ by induction on the form of t.*

$$ctype_\phi(x, x) = \phi$$

$$ctype_\phi(x, t_1\ t_2) = \phi''$$

Where $ctype_\phi(x, t_1) = \phi' \to \phi''$.

The following lemma states two very important properties of $ctype_\phi$. We do not include any proofs in this paper, but all proofs of all lemmas have been done and can be found in the compaion report [5].

Lemma 1 (Properties of $ctype_\phi$)**.**

i. If $ctype_\phi(x, t) = \phi'$ then $head(t) = x$ and ϕ' is a subexpression of ϕ.

ii. If $\Gamma, x : \phi, \Gamma' \vdash t : \phi'$ and $ctype_\phi(x, t) = \phi''$ then $\phi' \equiv \phi''$.

iii. If $\Gamma, x : \phi, \Gamma' \vdash t_1\ t_2 : \phi'$, $\Gamma \vdash t : \phi$, $[t/x]^\phi t_1 = \lambda y : \phi_1.q$, and t_1 is not then there exists a type ψ such that $ctype_\phi(x, t_1) = \psi$.

The previous lemma used a function called $head$ whose type is $\mathscr{T} \to \mathscr{T}$. It is defined by the following definiton.

Definition 12. *The head function is defined as follows:*

$$head(x) = x$$

$$head(\lambda x : \phi.t) = \lambda x : \phi.t$$

$$head(t_1\ t_2) = head(t_1)$$

The purpose of $ctype_\phi$ will be easier to understand after we define the hereditary substitution function. So lets consider the following definition of the hereditary substitution function for STLC.

Definition 13. *The following defines the hereditary substitution function for STLC. It is defined by recursion on the form of the term being substituted into and the cut type ϕ.*

$$[t/x]^\phi x = t$$

$$[t/x]^\phi y = y$$

Where y is a variable distinct from x.

$[t/x]^\phi(\lambda y : \phi'.t') = \lambda y : \phi'.([t/x]^\phi t')$

$[t/x]^\phi(t_1\ t_2) = ([t/x]^\phi t_1)\ ([t/x]^\phi t_2)$

Where $([t/x]^\phi t_1)$ *is not a* λ*-abstraction, or both* $([t/x]^\phi t_1)$ *and* t_1 *are* λ*-abstractions,*

or $ctype_\phi(x, t_1)$ *is undefined.*

$[t/x]^\phi(t_1\ t_2) = [(([t/x]^\phi t_2)/y]^{\phi''} s'_1$

Where $([t/x]^\phi t_1) \equiv \lambda y : \phi''.s'_1$ *for some* y*,* s'_1*, and* ϕ'' *and* $ctype_\phi(x, t_1) = \phi'' \to \phi'$*.*

We can see that every case of the previous definition except the application cases are identical to the definition of capture-avoiding substitution. In fact one can think of the mechanics of the hereditary substitution function as first doing a capture-avoiding substitution and then deciding if any newly created redexes need reducing. Remember this is intentional, because the hereditary substitution function should only differ when a new redex is created as a result of a capture-avoiding substitution. The creation of a new redex as a result of a capture-avoiding substitution can only occur when substituting into an application with respect to STLC.

Before describing how we can decide if a new redex was introduced by the capture-avoiding substitution funcion we make some general comments about the hereditary substitution function above. The first major thing to note about our definition of the hereditary substitution function defined above is that we define it in terms of all terms not just normal forms. That is its type is $\mathscr{T} \times \mathscr{T} \times \mathscr{T} \to \mathscr{T}$. This was first done by H. Eades and A. Stump in [6] in their work on using the hereditary substitution function to show normalization of Stratified System F. Secondly, the definition of the hereditary substitution function is nearly total by definition. In fact it is only the second case for applications that prevents totality from being trivial. Now if this case was used we know that $ctype_\phi(x, t_1) = \phi'' \to \phi'$ and by Lemma 1 $\phi'' \to \phi'$ is a subexpression of ϕ. This implies that ϕ'' is a strict subexpression on ϕ. So in this case the type decreases by the strict subexpression ordering. In fact we prove totality of the hereditary substitution function for STLC using the lexicographic combination (ϕ, t) of the strict subexpression ordering. This reveals exactly the purpose of $ctype_\phi$. Its purpose is to reveal information about the types of the input terms to the hereditary substitution function that allows us to obtain a well-founded ordering which can be used to prove properties of the hereditary substitution function. Now we do not want to underplay the importance of the ordering on types. In order to be able to even define the hereditary substitution function and prove that it is indeed a total function one must have an ordering on types. This is very important. Now in the case of STLC the ordering is just the subexpression ordering, while for other systems the ordering can be much more complex. For some type theories no ordering exists on just the types. Whatever ordering we use for the types the construct type function brings this ordering into the definition of the hereditary substitution function.

So how do we know when a new redex was created as a result of a capture-avoiding substitution? A new redex was created when the hereditary substitution function is being applied to an application, and if the the hereditary substitution function is applied to the head of the application and the head was not a λ-abstraction to begin with, but the result of the hereditary substitution function was a λ-abstraction. If this is not the case then no redex was created. The first case for applications in the definition of the hereditary substitution function takes care of this situation. Now the final case for applications handles when a new redex was created. In this case we know applying the hereditary substitution function to the head of the application results in a λ-abstraction and we know $ctype_\phi$ is defined. So by Lemma 1 we know the head of t_1 is x so t_1 cannot be a λ-abstraction. Thus, we have created a new redex so we reduce this redex by hereditary substituting $[t/x]^\phi t'_2$ for for y of type ϕ'' into the body of the λ-abstraction t'_1. We use hereditary substitution here because we may create more redexes as a result of reducing the previously created redex.

Throughout this section STLC has been the running example where the only way to create redexes is through hereditarily substituting into the head of an application. This is because according to our operational semantics for STLC (full β-reduction) the only redex is the β-rule. If our operational semantics included more redexes we would have more ways to create redexes and the definition of the hereditary substitution function would need to account for this. Hence, the definition of the hereditary substitution function is guided by the chosen operational semantics. Lastly, the properties discussed so far in this section apply to all hereditary substitution functions in general not just for the one defined for STLC.

4.2 Properties of the Hereditary Substitution Function

In this section we formally state several properties every hereditary substitution function must satisfy if they are to be used to show normalization of a type theory. Below the hereditary substitution function denoted $[t/x]^{\phi}t'$ is an arbitray hereditary substitution function. While these properties must be satisfied by the hereditary substitution function when proving normalization most of them should be requirements in general, because they show the correctness of the definition of the hereditary substitution function. The properties are totality, normality and type preserving, redex preservation, and soundness with respect to reduction. Now the first three properties are known and can be found within the literature while the last two are novel properties.

We mentioned earlier that the hereditary substitution function is total. Not only is it total, but when given typeable input the output of the hereditary substitution function is also typeable with the same type as the term we are substituting into. We state the totality and type preserving property as follows:

Lemma 2 (Total and Type Preserving)**.** *Suppose* $\Gamma \vdash t : \phi$ *and* $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$. *Then there exists a term* t'' *such that* $[t/x]^{\phi}t' = t''$ *and* $\Gamma, \Gamma' \vdash t'' : \phi'$.

The next property is normality preserving which states that when the hereditary substitution function is applied to normal forms then the result of the hereditary substitution function is a normal form. We state this formally as follows:

Lemma 3 (Normality Preserving)**.** *If* $\Gamma \vdash n : \phi$ *and* $\Gamma, x : \phi \vdash n' : \phi'$ *then there exists a normal term* n'' *such that* $[n/x]^{\phi}n' = n''$.

The next property, soundness with respect to reduction, if one thinks about it seems obvious, but we have not been able to find any actual proofs of it in the literature. It shows that the hereditary substitution function does nothing more than full β-reduction.

Lemma 4 (Soundness with Respect to Reduction)**.** *If* $\Gamma \vdash t : \phi$ *and* $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$ *then* $[t/x]t' \rightsquigarrow^* [t/x]^{\phi}t'$.

All the properties we have considered so far must be satisfied by all definitions of the hereditary substitution function in order for the definition to be considered correct. These properties insure the function is doing what it is supposed to be doing and nothing more. The final property is redex preservation which states that the number of redexes in the input of the hereditary substitution function is larger than the number of redexes in the output. We state this property in the following lemma.

Lemma 5 (Redex Preserving)**.** *If* $\Gamma \vdash t : \phi$, $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$ *then* $|rset(t', t)| \geq |rset([t/x]^{\phi}t')|$.

The statement of the previous lemma depends on a function called $rset : \mathscr{T} \to \mathscr{P}(\mathscr{T})$. This function takes a term and returns the set of redexes within the input term. This function is extended to multiple inputs as $rset(t_1, \ldots, t_n) = rset(t_1) \cup \cdots \cup rset(t_2)$. We do not explicitly state this function here because it is language dependent; that is the definition depends on the definition of the type theory we are working with, especially the operational semantics. We will see several explicit definitions of this function in the next section. Now the reader might be wondering why it is we called the previouse property redex preservation, because the lemma shows a reduction in the number of redexes which might seem counter intuitive. This lemma is just a start at what we conjecture to really be going on. Yes, it is true that in some cases the number of redexes decrease, but this is not the case in general. We plan to show in the future exactly when this reduction occurs and when redexes are preserved. We conjecture that more redexes are preserved then removed by the hereditary substitution function. At this point we know everything we need to know in order to use hereditary substitution to prove normalization of different type theories.

5 Normalization by Hereditary Substitution

5.1 The Language

Before proving normalization of several different type theories we first give some intuition of how a proof of normalization goes. This overview can be thought of as a template to follow when understanding the following normalization proofs.

5.2 Semantics of Types and Concluding Normalization

Proving normalization of some type theory using hereditary substitution involves six main steps:

i. define a well-founeded ordering on types,

ii. define the hereditary substitution function,

iii. prove the properties of the hereditary substitution function,

iv. define a semantics for types called the interpretation of types,

v. prove the semantics is closed under hereditary substitutions (this implies that the semantics is closed under capture avoiding substitutions), and

vi. prove all typeable terms are members of the interpretation of their type. This is known as type soundness.

We have already discussed parts one through three. The next major part of proving normalization using hereditary substitution is defining a semantics for the types of the type theory under consideration. The interpretations of types are essentially sets of terms with a common type in a context. We define the interpretation of types on normal terms and then extend this definition to non-normal terms. A non-normal term is a member of the interpretation of a type if and only if it normalizes to a term in that interpretation. For normal terms, we use a modification of an interpretation of types proposed by Prawitz, who defines the meaning of open terms in terms of their ground instances [17, 18]. In contrast, our definition treats open terms directly. The definition for non-normal terms implies that if the typing rules are sound with respect to the interpretation of types, then all typable terms are normalizing.

Semantic inversion. If we are to prove normalization by hereditary substitution using an interpretation of types then the key property of *semantic inversion* must hold with respect to the interpretation of types. Semantic inversion is defined just like an inversion lemma for a typing relation:

$$\text{If } t_1\ t_2 \in [\![\phi']\!]_\Gamma, \textit{then } \exists \phi. t_1 \in [\![\phi \to \phi']\!]_\Gamma \textit{ and } t_2 \in [\![\phi]\!]_\Gamma.$$

Here, $[\![\phi]\!]_\Gamma$ is an interpretation of types defined with respect to a context Γ. We have found, in the developments below, that semantic inversion must be true for all interpretations $[\![\phi]\!]_\Gamma$, in order for a proof by hereditary substitution to go through. We will see this in more detail below.

The second piece of proving normalization using hereditary substitution is to prove that the semantics is closed under hereditary substitutions. The following lemma states exactly what we mean by the interpretation of types being closed under hereditary substitutions, where $[\![\phi]\!]_\Gamma$ denotes the interpretation of types (the semantics).

Lemma 6 (Substitution for the Interpretation of Types). *If* $n' \in [\![\phi']\!]_{\Gamma,x:\phi,\Gamma'}$, $n \in [\![\phi]\!]_\Gamma$, *then* $[n/x]^\phi n' \in [\![\phi']\!]_{\Gamma,\Gamma'}$.

The substitution lemma defined above is the entry point of where hereditary substitution comes into play in the proof of normalization. This lemma is then used in the proof of the type soundness theorem which makes this possibly the second most important result in the entire proof of normalization next to the type soundness theorem. The type soundness theorem is the final piece of the puzzle. It just states that if a term is typeable then that term is in the interpretation of its type. Using this it is a trivial result to conclude that for any typeable term that term normalizes. We now know all we need to know about hereditary substitution and proving normalization. In the following sections we prove normalization using hereditary substitution following the template defined in this section of a verity of different type theories.

5.3 Normalization of The Simply Typed λ-Calculus

Everything we have seen so far is enough to conclude normalization of STLC. Our well-founded ordering on types is just the subexpression ordering. The hereditary substitution function was defined in Definition 13 and its properties in Section 4.2. The definition of the interpretation of types is defined by the following definition.

Definition 14. *First we define when a normal form is a member of the interpretation of type ϕ in context Γ*

$$n \in [\![\phi]\!]_\Gamma \iff \Gamma \vdash n : \phi,$$

and this definition is extended to non-normal forms in the following way

$$t \in [\![\phi]\!]_\Gamma \iff t \rightsquigarrow^! n \in [\![\phi]\!]_\Gamma,$$

where $t \rightsquigarrow^! t'$ is syntactic sugar for $t \rightsquigarrow^* t' \not\rightsquigarrow$.

Next we show that the definition of the interpretation of types is closed under hereditary substitutions.

Lemma 7 (Substitution for the Interpretation of Types)**.** *If $n' \in [\![\phi']\!]_{\Gamma,x:\phi,\Gamma'}$, $n \in [\![\phi]\!]_\Gamma$, then $[n/x]^\phi n' \in [\![\phi']\!]_{\Gamma,\Gamma'}$.*

Proof. By Lemma 2 we know there exists a term $\hat{n}$ such that $[n/x]^\phi n' = \hat{n}$ and $\Gamma, \Gamma' \vdash \hat{n} : \phi'$ and by Lemma 3 $\hat{n}$ is normal. Therefore, $[n/x]^\phi n' = \hat{n} \in [\![\phi']\!]_{\Gamma,\Gamma'}$. □

Finally, by the definition of the interpretation of types the following result implies that STLC is normalizing.

Theorem 1 (Type Soundness)**.** *If $\Gamma \vdash t : \phi$ then $t \in [\![\phi]\!]_\Gamma$.*

Corollary 1 (Normalization)**.** *If $\Gamma \vdash t : \phi$ then $t \rightsquigarrow^! n$.*

5.4 Normalization of Stratified System F (SSF)

In [6] H. Eades and A. Stump show that SSF is normalizing using a proof method which uses the hereditary substitution function implicitly. We find it apparent that the hereditary substitution technique we have developed throughout this paper is easier to understand, use, and is more informative with respect to the hereditary substitution function and its interaction with the type theory than the implicit version of the proof method. So in this section we reprove normalization of SSF using the hereditary function explicitly as we developed throughout this paper.

We extend STLC with predicative polymorphism to obtain a language first analyzed by D. Leivant and N. Danner in [11] called Stratified System F. In terms of programming polymorphism allows one to define programs of generic type. For example, $\Lambda X.\lambda x : X.x$ would be the generic identity function. This increases the overall expressive power of the type theory. In STLC there is no way of defining the generic identity function. Now in terms of logic polymorphism amounts to universal quantification. So instead of propositional intuitionistic logic with only implication we obtain first-order intuitionistic logic with only implication and no predicates.

Stratified System F, consists of types which are stratified into levels (or ranks) based on type-quantification. The types that belong to level zero have no type-quantification, the types at level one only quantify over types of level zero, and the types at level n quantify over the types of level $n-1$. Stratifying System F into levels effectively prevents impredicativity, where a type $\phi = \forall X.\phi'$ quantifies over types including itself. In Stratified System F, we can only have $\phi = \forall X : *_n.\phi'$ where X ranges over the set of types of level $n-1$ which does not include ϕ.

We consider only the finite version of Stratified System F as proposed by Leivant in [11] with some slight modifications. One of the major differences of our version of Stratified System F compared to Leivant's is that we use a kinding relation which relates a level to a type with respect to some context, using algorithmic type/kind-checking rules. The syntax for Stratified System F can be found in the next definition followed by the definition of the reduction rules.

Definition 15. *The syntax for terms, types, and kinds:*

$$\begin{array}{lcl} K & := & *_0 \mid *_1 \mid \ldots \\ \phi & := & X \mid \phi \to \phi \mid \forall X : K.\phi \\ t & := & x \mid \lambda x : \phi.t \mid t\,t \mid \Lambda X : K.t \mid t[\phi] \end{array}$$

Definition 16. *Full β-reduction for SSF:*

$$\begin{array}{lcl}
(\Lambda X : *_p.t)[\phi] & \rightsquigarrow & [\phi/X]t \\
(\lambda x : \phi.t)t' & \rightsquigarrow & [t'/x]t
\end{array}$$

As we can see the syntax for SSF is essentionally identical to STLC except we have added two new terms $\Lambda X : K.t$ and $t[\phi]$. The former is called a type abstraction while the later is called type application. This is where polymorphism is coming in. A type abstraction allows one to write a generic program and then type application allows one to instantiate a generic program with a concrete type. In Definition 26 we extend full β-reduction to include a new redex for type application. Both the kinding and typing relations depend on well-formed contexts which is defined next.

Definition 17. *Context well-formedness rules:*

$$\frac{}{\cdot\; Ok} \qquad \frac{\Gamma\; Ok}{\Gamma, X : *_p\; Ok} \qquad \frac{\Gamma \vdash \phi : *_p \qquad \Gamma\; Ok}{\Gamma, x : \phi\; Ok}$$

As stated before we use kinds to denote the level of a type. We define algorithmic kind-checking rules in the following definition.

Definition 18. *The kind assignment rules for SSF are defined as follows:*

$$\frac{\Gamma \vdash \phi_1 : *_p \qquad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{max(p,q)}} \qquad \frac{\Gamma, X : *_q \vdash \phi : *_p}{\Gamma \vdash \forall X : *_q.\phi : *_{max(p,q)+1}} \qquad \frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\; Ok}{\Gamma \vdash X : *_q}$$

The context Γ, type ϕ, and kind $*_l$ are inputs and there are no outputs. In the previous definition we extended STLC with kinding rules. These rules serve the exact same purpose as typing rules, but instead of typing terms they type types. They insure that types are well-formed. Here types are well-formed when they have the proper leveling assigned to them. The following lemma shows that all kindable types are kindable with respect to a well-formed context.

Lemma 8. *If $\Gamma \vdash \phi : *_p$ then $\Gamma\; Ok$.*

The previous lemma insures that if a type is kindable then we could not have used a "bogus" context where we assume something we are not allowed to assume. We define algorithmic type-checking rules in the next definition.

Definition 19. *Type assignment rules for SSF:*

$$\frac{\Gamma(x) = \phi \qquad \Gamma\; Ok}{\Gamma \vdash x : \phi} \qquad \frac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x : \phi_1.t : \phi_1 \to \phi_2} \qquad \frac{\Gamma \vdash t_1 : \phi_1 \to \phi_2 \qquad \Gamma \vdash t_2 : \phi_1}{\Gamma \vdash t_1\; t_2 : \phi_2}$$

$$\frac{\Gamma, X : *_p \vdash t : \phi}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.\phi} \qquad \frac{\Gamma \vdash t : \forall X : *_l.\phi_1 \qquad \Gamma \vdash \phi_2 : *_l}{\Gamma \vdash t[\phi_2] : [\phi_2/X]\phi_1}$$

The type-checking rules depend on the kinding relation defined above. As with the earlier parts of the language we extend the type-checking rules with rules for type abstraction and type application. Note that the level of the type ϕ and the level of the type variable X in the type application rule must be the same.

5.4.1 Basic Syntactic Lemmas

We now state several results about the kinding relation. All of these are just basic results needed by the proofs of the key lemmas and theorems later. The reader may wish to just quickly read through them. In Section 5.4.2 we will define an ordering on types and prove that it is well founded. This proof depends on the following lemma. Lemma 9 shows that if a type is kindable at some level then it is also kindable at strictly larger levels.

Lemma 9 (Level Weakening for Kinding)**.** *If $\Gamma \vdash \phi : *_r$ and $r < s$ then $\Gamma \vdash \phi : *_s$.*

Lemma 10 is used in the proof of Substitution for Typing (Lemma 23), Lemma 11 is used in the main substitution lemma (Lemma 21), and Lemma 12 is used in the proof of Context Weakening for the Interpretation of Types (Lemma 22).

Lemma 10 (Substitution for Kinding, Context-Ok)**.** *Suppose* $\Gamma \vdash \phi' : *_p$*. If* $\Gamma, X : *_p, \Gamma' \vdash \phi : *_q$ *with a derivation of depth* d*, then* $\Gamma, [\phi'/X]\Gamma' \vdash [\phi'/X]\phi : *_q$ *with a derivation of depth* d*. Also, if* $\Gamma, X : *_p, \Gamma'$ *Ok with a derivation of depth* d*, then* $\Gamma, [\phi'/X]\Gamma'$ *Ok with a derivation of depth* d*.*

Lemma 11 (Context Strengthening for Kinding, Context-Ok)**.** *If* $\Gamma, x : \phi', \Gamma' \vdash \phi : *_p$ *with a derivation of depth* d*, then* $\Gamma, \Gamma' \vdash \phi : *_p$ *with a derivation of depth* d*. Also, if* $\Gamma, x : \phi, \Gamma'$ *Ok with a derivation of depth* d*, then* Γ, Γ' *Ok with a derivation of depth* d*.*

Lemma 12 (Context Weakening for Kinding)**.** *If* $\Gamma, \Gamma'', \Gamma'$ *Ok, and* $\Gamma, \Gamma' \vdash \phi : *_p$ *each with a derivation of depth* d *then* $\Gamma, \Gamma'', \Gamma' \vdash \phi : *_p$ *with a derivation of depth* d*.*

Lemma 13 (Regularity)**.** *If* $\Gamma \vdash t : \phi$ *then* $\Gamma \vdash \phi : *_p$ *for some* p*.*

We have stated all the basic lemmas we will need. We now step through our template outlined in Section 5 to conclude normalization for SSF.

5.4.2 Well-Founded Ordering on Types

The following definition defines a well-founded ordering on the types of SSF. It consists of essentially the strict-subexpression ordering with an additional case for universal types. For the case of universal types the ordering states that a universal type is always larger than the instantiation of the body of the universal type. Now this seems odd, because syntactically the instantiation could have increased the size of ϕ to be larger than the size of the universal type, but it turns out that the level of the type actually decreases. That is we know the level of the universal type is larger than the level of the instantiation.

Definition 20. *The ordering* $>_\Gamma$ *is defined as the least relation satisfying the universal closures of the following formulas:*

$$\begin{array}{lll}
\phi_1 \to \phi_2 & >_\Gamma & \phi_1 \\
\phi_1 \to \phi_2 & >_\Gamma & \phi_2 \\
\forall X : *_l.\phi & >_\Gamma & [\phi'/X]\phi \text{ where } \Gamma \vdash \phi' : *_l.
\end{array}$$

We need transitivity in a number of places so we state that next.

Lemma 14 (Transitivity of $>_\Gamma$)**.** *Let* ϕ*,* ϕ'*, and* ϕ'' *be kindable types. If* $\phi >_\Gamma \phi'$ *and* $\phi' >_\Gamma \phi''$ *then* $\phi >_\Gamma \phi''$*.*

To prove that the ordering on types ($>_\Gamma$) is well founded we need a function which computes the depth of a type. We will use this in a lexicographic ordering in the proof of Lemma 15 and is vital to showing that our ordering on types is well founded.

Definition 21. *The depth of a type* ϕ *is defined as follows:*

$$\begin{array}{lll}
depth(X) & = & 1 \\
depth(\phi \to \phi') & = & depth(\phi) + depth(\phi') \\
depth(\forall X : *_l.\phi) & = & depth(\phi) + 1
\end{array}$$

We define the following metric (l, d) in lexicographic combination, where l is the level of a type ϕ and d is the depth of ϕ. The following lemma shows that if $\phi >_\Gamma \phi'$ then $(l, d) > (l', d')$. We will use this lemma to show well-foundedness of the ordering on types $>_\Gamma$.

Lemma 15 (Well-Founded Measure)**.** *If* $\phi >_\Gamma \phi'$ *then* $(l, d) > (l', d')$*, where* $\Gamma \vdash \phi : *_l$*,* $depth(\phi) = d$*,* $\Gamma \vdash \phi : *_{l'}$*, and* $depth(\phi') = d'$*.*

We now have the desired results to prove that the ordering $>_\Gamma$ is well-founded.

Theorem 2 (Well-Founded Ordering)**.** *The ordering* $>_\Gamma$ *is well-founded on types* ϕ *such that* $\Gamma \vdash \phi : *_l$ *for some* l*.*

5.4.3 The Hereditary Substitution Function

Moving forward with our proof of normalization based on our template we now need to define the hereditary substitution function for SSF. The definition of the hereditary substitution function is a basic extension of hereditary substitution function for STLC. Before defining the hereditary substitution function we first define the construct type function for SSF. This function is now defined for three different types of input. It is defined for variables and applications just as before with the addition of type application.

Definition 22. *The construct type function for SSF is defined as follows:*

$ctype_\phi(x, x) = \phi$

$ctype_\phi(x, t_1\ t_2) = \phi''$
Where $ctype_\phi(x, t_1) = \phi' \to \phi''$.

$ctype_\phi(x, t[\phi']) = [\phi'/X]\phi''$
Where $ctype_\phi(x, t) = \forall X : *_l.\phi''$.

Finally, we can define the hereditary substitution function for SSF.

Definition 23. *We define the hereditary substitution function for SSF as follows:*

$[t/x]^\phi x = t$

$[t/x]^\phi y = y$
Where y *is a variable distinct from* x.

$[t/x]^\phi(\lambda y : \phi'.t') = \lambda y : \phi'.([t/x]^\phi t')$

$[t/x]^\phi(\Lambda X : *_l.t') = \Lambda X : *_l.([t/x]^\phi t')$

$[t/x]^\phi(t_1\ t_2) = ([t/x]^\phi t_1)\ ([t/x]^\phi t_2)$
Where $([t/x]^\phi t_1)$ *is not a* λ*-abstraction, or both* $([t/x]^\phi t_1)$ *and* t_1 *are* λ*-abstractions, or* $ctype_\phi(x, t_1)$ *is undefined.*

$[t/x]^\phi(t_1\ t_2) = [([t/x]^\phi t_2)/y]^{\phi''} s'_1$
Where $([t/x]^\phi t_1) \equiv \lambda y : \phi''.s'_1$ *for some* y, s'_1, *and* ϕ'' *and* $ctype_\phi(x, t_1) = \phi'' \to \phi'$.

$[t/x]^\phi(t'[\phi']) = ([t/x]^\phi t')[\phi']$
Where $[t/x]^\phi t'$ *is not a type abstraction or* t' *and* $[t/x]^\phi t'$ *are type abstractions.*

$[t/x]^\phi(t'[\phi']) = [\phi'/X]s'_1$
Where $[t/x]^\phi t' \equiv \Lambda X : *_l.s'_1$, *for some* X, s'_1 *and* $\Gamma \vdash \phi' : *_q$, *such that,* $q \leq l$ *and* $ctype_\phi(x, t') = \forall X : *_l.\phi''$.

The next lemma states the familiar properties of the construct type function. The first property is slightly different then the one defined for STLC. The difference arises from the fact that the ordering on types is not just the subexpression ordering, but relies on the level of the type in order to order universal types. So instead of ϕ being a subexpression of the output of $ctype_\phi$ it will be greater than or equal to the output of $ctype_\phi$. The remainder of the properties are as usual.

Lemma 16 (Properties of $ctype_\phi$)**.**

i. *If* $\Gamma, x : \phi, \Gamma' \vdash t : \phi'$ *and* $ctype_\phi(x,t) = \phi''$ *then* $head(t) = x$, $\phi' \equiv \phi''$, *and* $\phi' \leq_{\Gamma,\Gamma'} \phi$.

ii. *If* $\Gamma, x : \phi, \Gamma' \vdash t_1\ t_2 : \phi'$, $\Gamma \vdash t : \phi$, $[t/x]^\phi t_1 = \lambda y : \phi_1.q$, *and* t_1 *is not then there exists a type* ψ *such that* $ctype_\phi(x, t_1) = \psi$.

iii. *If* $\Gamma, x : \phi, \Gamma' \vdash t'[\phi''] : \phi'$, $\Gamma \vdash t : \phi$, $[t/x]^\phi t' = \Lambda X : *_l.t''$, *and* t' *is not then there exists a type* ψ *such that* $ctype_\phi(x, t') = \psi$.

5.4.4 Properties of the Hereditary Substitution Function

We now extend $rset$ to add type application redexes to the set of overall redexes of a term. It is defined in the following definition.

Definition 24. *The following function constructs the set of redexes within a term:*

$$rset(x) = \emptyset$$

$$rset(\lambda x : \phi.t) = rset(t)$$

$$rset(\Lambda X : *_l.t) = rset(t)$$

$$\begin{array}{lll} rset(t_1\ t_2) & & \\ \quad = & rset(t_1, t_2) & \textit{if } t_1 \textit{ is not a } \lambda\textit{-abstraction.} \\ \quad = & \{t_1\ t_2\} \cup rset(t'_1, t_2) & \textit{if } t_1 \equiv \lambda x : \phi.t'_1. \end{array}$$

$$\begin{array}{lll} rset(t''[\phi'']) & & \\ \quad = & rset(t'') & \textit{if } t'' \textit{ is not a type abstraction.} \\ \quad = & \{t''[\phi'']\} \cup rset(t''') & \textit{if } t'' \equiv \Lambda X : *_l.t'''. \end{array}$$

The extension of $rset$ *to multiple arguments is defined as follows:*

$$rset(t_1, \ldots, t_n) =^{def} rset(t_1) \cup \cdots \cup rset(t_n).$$

Next we state all the properties of the hereditary substitution function. They are equivalent to the properties stated in Section 4.2 the only difference are their proofs.

Lemma 17 (Total and Type Preserving)**.** *Suppose* $\Gamma \vdash t : \phi$ *and* $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$. *Then there exists a term* t'' *such that* $[t/x]^\phi t' = t''$ *and* $\Gamma, \Gamma' \vdash t'' : \phi'$.

Lemma 18 (Redex Preserving)**.** *If* $\Gamma \vdash t : \phi$, $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$ *then* $|rset(t', t)| \geq |rset([t/x]^\phi t')|$.

Lemma 19 (Normality Preserving)**.** *If* $\Gamma \vdash n : \phi$ *and* $\Gamma, x : \phi \vdash n' : \phi'$ *then there exists a normal term* n'' *such that* $[n/x]^\phi n' = n''$.

Lemma 20 (Soundness with Respect to Reduction)**.** *If* $\Gamma \vdash t : \phi$ *and* $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$ *then* $[t/x]t' \rightsquigarrow^* [t/x]^\phi t'$.

5.4.5 Substitution for the Interpretation of Types.

The definition of the interpretation of types is identical to the definition in Section 5.3 so we do not repeat it here. Before concluding normalization we state the main substitution lemma for the interpretation of types.

Lemma 21 (Substitution for the Interpretation of Types)**.** *If* $n' \in [\![\phi']\!]_{\Gamma,x:\phi,\Gamma'}$, $n \in [\![\phi]\!]_{\Gamma}$, *then* $[n/x]^{\phi} n' \in [\![\phi']\!]_{\Gamma,\Gamma'}$.

Proof. By Lemma 17 we know there exists a term $\hat{n}$ such that $[n/x]^{\phi}n' = \hat{n}$ and $\Gamma, \Gamma' \vdash \hat{n} : \phi'$ and by Lemma 19 $\hat{n}$ is normal. Therefore, $[n/x]^{\phi}n' = \hat{n} \in [\![\phi']\!]_{\Gamma,\Gamma'}$. □

Before moving on to proving soundness of typing and concluding normalization we need a couple of results about the interpretation of types: context weakening and type substitution. They both are used in the proof of the type soundness theorem (Theorem 3).

Lemma 22 (Context Weakening for Interpretations of Types)**.** *If* $\Gamma, \Gamma', \Gamma''$ *Ok and* $n \in [\![\phi]\!]_{\Gamma,\Gamma''}$ *then* $n \in [\![\phi]\!]_{\Gamma,\Gamma',\Gamma''}$.

Lemma 23 (Type Substitution for the Interpretation of Types)**.** *If* $n \in [\![\phi']\!]_{\Gamma,X:*_l,\Gamma'}$ *and* $\Gamma \vdash \phi : *_l$ *then* $[\phi/X]n \in [\![[\phi/X]\phi']\!]_{\Gamma,[\phi/X]\Gamma'}$.

5.4.6 Concluding Normalization.

We are now ready to present our main result. The next theorem shows that the type-assignment rules are sound with respect to the interpretation of types.

Theorem 3 (Type Soundness)**.** *If* $\Gamma \vdash t : \phi$ *then* $t \in [\![\phi]\!]_{\Gamma}$.

Therefore, we conclude normalization of SSF.

Corollary 2 (Normalization)**.** *If* $\Gamma \vdash t : \phi$ *then* $t \rightsquigarrow^! n$.

5.5 Normalization of Stratified System F$^{\omega}$

In this section we do something a bit different from the previous sections. Here we show that if we combine the type theories STLC and SSF that we can use the previous normalization results to conclude normalization of this new language. First we define the new language.

5.6 The Language

J. Girard defined a well known type-theory where the types not only consist of type variables, function types, and universal quantification, but also contain λ-abstractions and application. This type theory is called System F$^{\omega}$ [1]. Now these new λ-abstractions represent functions from types to types. Logically these correspond to predicates which means this language corresponds to a restricted form of second-order intuitionistic logic with only implication as a primitive and no quantification over predicates. System F$^{\omega}$ contains impredicativity, but this adds a lot of complexity to the language so we restrict the theory just as we did for SSF to obtain a more less expressive theory called Stratified System F$^{\omega}$ (SSF$^{\omega}$). Lets define the language. The following definition defines the syntax for SSF$^{\omega}$.

Definition 25. *The syntax for terms, types, and kinds:*

$$\begin{array}{lll} K & := & K \to K \mid *_0 \mid *_1 \mid \ldots \\ \phi & := & X \mid \phi \to \phi \mid \forall X : K.\phi \mid \lambda X : *_l.\phi \mid \phi\ \phi \\ t & := & x \mid \lambda x : \phi.t \mid t\ t \mid \Lambda X : K.t \mid t[\phi] \end{array}$$

Now we define our operational semantics for SSF$^{\omega}$ in the following definition.

Definition 26. *Full β-reduction for SSF$^{\omega}$:*

$$\begin{array}{lcl}(\Lambda X : K.t)[\phi] & \rightsquigarrow & [\phi/X]t \\ (\lambda x : \phi.t)\ t' & \rightsquigarrow & [t'/x]t \\ (\lambda X : *_l.\phi)\ \phi' & \rightsquigarrow & [\phi'/x]\phi\end{array}$$

We can see that this system is almost the same as SSF except we have added essentially STLC to the type level of the theory, and we now allow quantification over predicates. That is take STLC where the terms are types and the types are kinds and that is essentially what we have added to the type level of SSF to obtain SSF$^\omega$. This allows us to compute types, so in terms of programming we are able to write very generic programs where the type of the program may need to be computed. Next we define the kinding and typing rules for this system. Now the rules for well-formed contexts do not change from SSF so we do not define them here.

Definition 27. *The kind assignment rules for SSF$^\omega$ are defined as follows:*

$$\frac{\Gamma \vdash \phi_1 : *_p \qquad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{max(p,q)}} \qquad \frac{\Gamma, X : *_q \vdash \phi : *_p}{\Gamma \vdash \forall X : *_q.\phi : *_{max(p,q)+1}} \qquad \frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\ Ok}{\Gamma \vdash X : *_q}$$

$$\frac{\Gamma, X : K_1 \vdash \phi : K_2}{\Gamma \vdash \lambda X : K_1.\phi : K_1 \to K_2} \qquad \frac{\Gamma \vdash \phi_1 : K_1 \to K_2 \qquad \Gamma \vdash \phi_2 : K_1}{\Gamma \vdash \phi_1\ \phi_2 : K_2}$$

To obtain SSF$^\omega$ from SSF all we have to do is add the kinding rules for type-level λ-abstractions and type-level application. The typing rules are the exactly same as SSF. We state them next.

Definition 28. *Type assignment rules for SSF$^\omega$:*

$$\frac{\Gamma(x) = \phi \qquad \Gamma\ Ok}{\Gamma \vdash x : \phi} \qquad \frac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x : \phi_1.t : \phi_1 \to \phi_2} \qquad \frac{\Gamma \vdash t_1 : \phi_1 \to \phi_2 \qquad \Gamma \vdash t_2 : \phi_1}{\Gamma \vdash t_1\ t_2 : \phi_2}$$

$$\frac{\Gamma, X : *_p \vdash t : \phi}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.\phi} \qquad \frac{\Gamma \vdash t : \forall X : *_l.\phi_1 \qquad \Gamma \vdash \phi_2 : *_l}{\Gamma \vdash t[\phi_2] : [\phi_2/X]\phi_1}$$

Now all the basic syntactic lemmas proved for SSF in Section 4.2 hold for SSF$^\omega$ so we do not state them here.

5.7 Concluding Normalization

Concluding normalization of SSF$^\omega$ is a little bit tricky, because the type level no longer consists of just constant types. We conduct the proof similarly to how H. Barendregt proves normalization of System F$^\omega$ in [1] except we use hereditary substitution where he uses a technique called reducibility. Since types contain computation we first must prove normalization of the type level and then using this result we prove normalization for the term (program) level.

Now the type level is exactly STLC with constructors for function types and universal types. So all we have to do is define the hereditary substitution function and then use the results from Section 5.3 to obtain normalization. Similarly, the term level of SSF$^\omega$ is exactly SSF so we do not need to define the hereditary substitution for SSF$^\omega$. We can just use the results from Section 5.4 to conclude normalization. Lets move onto defining the hereditary substitution functions for the type level. First we have to define the construct kind function. Note that $ckind_K$ is the exact same as $ctype_\phi$ except at the type level. Now all the properties of $ctype_\phi$ are also properties of $ckind_K$ so we do not prove them again here. The construct type function is exactly the same as SSF so we do not redefine it here. Also the kind K is called the cut kind and is used in the exact same way as the cut type ϕ.

Definition 29. *The $ckind_K$ function is defined with respect to a fixed kind K and has two arguments, a free variable X, and a type ϕ where X may be free in ϕ. We define $ckind_K$ by induction on the form of ϕ.*

$$ckind_K(X, X) = K$$

$$ckind_K(X, \phi_1\ \phi_2) = K'$$

Where $ckind_K(X, \phi_1) = K'' \to K'$.

Now we defined the type-level hereditary substitution function.

Definition 30. *The following defines the hereditary substitution function for the type level of SSF$^\omega$. It is defined by recursion on the form of the term being substituted into and the cut kind K.*

$\{\phi/X\}^K X = \phi$

$\{\phi/X\}^K Y = Y$

Where Y is a variable distinct from X.

$\{\phi/X\}^K(\phi_1 \to \phi_2) = (\{\phi/X\}^K \phi_1) \to (\{\phi/X\}^K \phi_2)$

$\{\phi/X\}^K(\forall Y : *_l.\phi') = \forall Y : *_l.\{\phi/X\}^K \phi'$

$\{\phi/X\}^K(\lambda Y : K_1.\phi') = \lambda Y : K_1.(\{\phi/X\}^K \phi')$

$\{\phi/X\}^K(\phi_1\ \phi_2) = (\{\phi/X\}^K \phi_1)\ (\{\phi/X\}^K \phi_2)$

Where $(\{\phi/X\}^K \phi_1)$ is not a λ-abstraction, or both $(\{\phi/X\}^K \phi_1)$ and ϕ_1 are λ-abstractions,

or $ckind_K(X, \phi_1)$ is undefined.

$\{\phi/X\}^K(\phi_1\ \phi_2) = \{(\{\phi/X\}^K \phi_2)/y\}^{K''} \phi_1'$

Where $(\{\phi/X\}^K \phi_1) \equiv \lambda Y : K''.\phi_1'$ for some Y, ϕ_1', and K'' and $ckind_K(X, \phi_1) = K'' \to K'$.

Now all the properties of the hereditary substitution function from Section 4.2 are easily modified to obtain the exact same results for the function just defined. The statements of the properties do not change except where there are terms we use types and where there are types we use kinds so we do not repeat the properties here. We now prove normalization for the type-level of SSF$^\omega$. The following defines the interpretation of types and the main substitution lemma.

Definition 31. *First we define when a normal form is a member of the interpretation of type ϕ in context Γ*

$$\psi \in [\![K]\!]_\Gamma \iff \Gamma \vdash \phi : K,$$

and this definition is extended to non-normal forms in the following way

$$\phi \in [\![K]\!]_\Gamma \iff \phi \rightsquigarrow^! \psi \in [\![K]\!]_\Gamma,$$

where we use ψ to denote a normal form at the type level.

Next we show that the definition of the interpretation of types is closed under hereditary substitutions.

Lemma 24 (Substitution for the Interpretation of Types)**.** *If $\phi' \in [\![K']\!]_{\Gamma, X:*_l, \Gamma'}$, $\phi \in [\![K]\!]_\Gamma$, then $\{\phi/X\}^K \phi' \in [\![K']\!]_{\Gamma,\Gamma'}$.*

Finally, by the definition of the interpretation of types the following result implies that the type-level of SSF$^\omega$ is normalizing.

Theorem 4 (Type Soundness)**.** *If $\Gamma \vdash \phi : K$ then $\phi \in [\![K]\!]_\Gamma$.*

Corollary 3 (Normalization)**.** *If $\Gamma \vdash \phi : K$ then $\phi \rightsquigarrow^! \psi$.*

Now we have to use the fact that we know the type level of SSF$^\omega$ is normalizing to conclude normalization of the term level. First we define the interpretation of types for the term level.

Definition 32. *First we define when a normal form is a member of the interpretation of normal type ϕ in context Γ*

$$n \in [\![\phi]\!]_\Gamma \iff \Gamma \vdash n : \phi,$$

and this definition is extended to non-normal forms in the following way

$$t \in [\![\phi]\!]_\Gamma \iff t \rightsquigarrow^! n \in [\![\phi]\!]_\Gamma.$$

Next we show that the definition of the interpretation of types is closed under hereditary substitutions.

Lemma 25 (Substitution for the Interpretation of Types)**.** *If* $n' \in [\![\phi']\!]_{\Gamma,x:\phi,\Gamma'}$, $n \in [\![\phi]\!]_\Gamma$, *then* $[n/x]^\phi n' \in [\![\phi']\!]_{\Gamma,\Gamma'}$.

At this point we have to show that for any term t of some type ϕ no matter if ϕ is normal or not then t reduces to a normal form. We have to do this in two steps. We first show that for all typeable terms where their types are normal they reduces to a normal form. Then knowing this we show that for all typeable terms where their types are not necessarily normal they reduce to a normal form. We first show the former.

Theorem 5 (Type Soundness Normal Types)**.** *If* $\Gamma \vdash t : \phi$ *and* ϕ *is normal then* $t \in [\![\phi]\!]_\Gamma$.

The following lemmas are used in the proof of the type soundness lemma which shows that all typeable terms are in the interpretation of the normal form of their type.

Lemma 26 (Preservation of Types)**.**

i. If $(\Gamma, x : \phi, \Gamma')$ *Ok and* $\phi \rightsquigarrow \phi'$ *then* $(\Gamma, x : \phi', \Gamma')$ *Ok.*

ii. If $\Gamma \vdash \phi : K$ *and* $\phi \rightsquigarrow \phi'$ *then there exists a* Γ' *such that* $\Gamma' \vdash \phi' : K$.

Lemma 27. *If* $\Gamma \vdash t : \phi$ *and* $\phi \rightsquigarrow \phi'$ *then there exists a* Γ' *such that* $\Gamma' \vdash t : \phi'$.

Finally, we conclude normalization by showing type soundness.

Theorem 6 (Type Soundness)**.** *If* $\Gamma \vdash t : \phi$ *then* $\phi \rightsquigarrow^! \psi$, *and there exists a* Γ' *such that* $t \in [\![\psi]\!]_{\Gamma'}$.

Proof. By regularity we know $\Gamma \vdash \phi : K$ for some kind K and by Corollary 3 there exists a normal type ψ such that $\phi \rightsquigarrow^! \psi$. Finally, by Lemma 27 there exists a Γ' such that $\Gamma' \vdash t : \psi$. Thus, by Theorem 5 $t \in [\![\psi]\!]_{\Gamma'}$. ☐

Corollary 4 (Normalization)**.** *If* $\Gamma \vdash t : \phi$ *then* $t \rightsquigarrow^! n$.

6 Conclusion

We started with an introduction to functional programming, type theory, and intuitionistic logic. Then we moved onto how to prove normalization of typed λ-calculi using hereditary substitution. Finally, we proved normalization of three typed theories: the Simply Typed λ-Calculus, Stratified System F and Stratified System F^ω. Our long-term goal is to apply the hereditary-substitutions method to proof-theoretically more complex type theories, in particular Gödel's System T. For this, we conjecture that a proof-theoretically more complex ordering on types will be required, and hence are exploring extensions to SSF to higher ordinals.

References

[1] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[2] F. Cardone and R. Hindley. History of lambda-calculus and combinatory logic. 2006.

[3] C. Chen and H. Xi. Combining Programming with Theorem Proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Tallinn, Estonia, September 2005.

[4] N. Danner and D. Leivant. Stratified polymorphism and primitive recursion. *Mathematical. Structures in Comp. Sci.*, 9(4):507–522, 1999.

[5] H. Eades and A. Stump. Using the Hereditary Substitution Function in Normalization Proofs. Available from `http://www.cs.uiowa.edu/~heades/`.

[6] H. Eades and A. Stump. Hereditary substitution for stratified system f. *International Workshop on Proof-Search in Type Theories, PSTT'10*, July 2010.

[7] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, April 1989.

[8] M. Hofmann and T. Streicher. The Groupoid Model Refutes Uniqueness of Identity Proofs. In *Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 208–212. IEEE Computer Society, 1994.

[9] W. A. Howard. The formulae-as-types notion of construction. 1969-1980.

[10] C. Keller and T. Altenkirch. Normalization by hereditary substitutions. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010. ACM.

[11] D. Leivant. Finitely stratified polymorphism. *Inf. Comput.*, 93(1):93–113, 1991.

[12] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

[13] Grigori Mints. *A short introduction to intuitionistic logic*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[14] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[15] F. Pfenning. On the undecidability of partial polymorphic type reconstruction. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[16] F. Pfenning. Structural cut elimination. In *LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, page 156, Washington, DC, USA, 1995. IEEE Computer Society.

[17] D. Prawitz. *Logical Consequence from a Constructivist Point of View*, pages 671–695. Volume 1 of Shapiro [19], 2005.

[18] D. Prawitz. Meaning approached via proofs. *Synthese*, 148(3):507–524, 2006.

[19] S. Shapiro. *The Oxford Handbook of Philosophy of Mathematics and Logic*. Oxford University Press, 2005.

[20] T. Sheard. Type-Level Computation Using Narrowing in Ωmega. In *Programming Languages meets Program Verification*, 2006.

[21] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languages meets Program Verification*, 2008. to appear.

[22] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, 2009.

[23] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007.

[24] M. Tatsuta and G. Mints. A simple proof of second-order strong normalization with permutative conversions. *Annals of Pure and Applied Logic*, 136:134–155, 2005.

[25] Philip Wadler. Proofs are programs: 19th century logic and 21st century computing, 2011.

[26] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. *Types for Proofs and Programs*, 3085:355–377, 2004.

A Proofs of Results Pertaining to STLC

A.1 Proof of Properties of $ctype_\phi$

We prove part one first. This is a proof by induction on the structure of t.

Case. Suppose $t \equiv x$. Then $ctype_\phi(x, x) = \phi$. Clearly, $head(x) = x$ and ϕ is a subexpression of itself.

Case. Suppose $t \equiv t_1\ t_2$. Then $ctype_\phi(x, t_1\ t_2) = \phi''$ when $ctype_\phi(x, t_1) = \phi' \to \phi''$. Now $t > t_1$ so by the induciton hypothesis $head(t_1) = x$ and $\phi' \to \phi''$ is a subexpression of ϕ. Therefore, $head(t_1\ t_2) = x$ and certainly ϕ'' is a subexpression of ϕ.

We now prove part two. This is also a proof by induction on the structure of t.

Case. Suppose $t \equiv x$. Then $ctype_\phi(x, x) = \phi$. Clearly, $\phi \equiv \phi$.

Case. Suppose $t \equiv t_1\ t_2$. Then $ctype_\phi(x, t_1\ t_2) = \phi_2$ when $ctype_\phi(x, t_1) = \phi_1 \to \phi_2$. By inversion on the assumed typing derivation we know there exists type ϕ'' such that $\Gamma, x : \phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$. Now $t > t_1$ so by the induciton hypothesis $\phi_1 \to \phi_2 \equiv \phi'' \to \phi'$. Therefore, $\phi_1 \equiv \phi''$ and $\phi_2 \equiv \phi'$.

Next we prove part three. This is a proof by induction on the structure of $t_1\ t_2$.

The only possiblities for the form of t_1 is x, $\hat{t}_1\ \hat{t}_2$. All other forms would not result in $[t/x]^\phi t_1$ being a λ-abstraction and t_1 not. If $t_1 \equiv x$ then there exist a type ϕ'' such that $\phi \equiv \phi'' \to \phi'$ and $ctype_\phi(x, x\ t_2) = \phi'$ when $ctype_\phi(x, x) = \phi \equiv \phi'' \to \phi'$ in this case. We know ϕ'' to exist by inversion on $\Gamma, x : \phi, \Gamma' \vdash t_1\ t_2 : \phi'$.

Now suppose $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$. Now knowing t_1' to not be a λ-abstraction implies that $\hat{t}_1$ is also not a λ-abstraction or $[t/x]^\phi t_1$ would be an application instead of a λ-abstraction. So it must be the case that $[t/x]^\phi \hat{t}_1$ is a λ-abstraction and $\hat{t}_1$ is not. Since $t_1\ t_2 > t_1$ we can apply the induction hypothesis to obtain there exists a type ψ such that $ctype_\phi(x, \hat{t}_1) = \psi$. Now by inversion on $\Gamma, x : \phi, \Gamma' \vdash t_1\ t_2 : \phi'$ we know there exists a type ϕ'' such that $\Gamma, x : \phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$. We know $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$ so by inversion on $\Gamma, x : \phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$ we know there exists a type ψ'' such that $\Gamma, x : \phi, \Gamma' \vdash \hat{t}_1 : \psi'' \to (\phi'' \to \phi')$. By part two of Lemma 16 we know $\psi \equiv \psi'' \to (\phi'' \to \phi')$ and $ctype_\phi(x, t_1) = ctype_\phi(x, \hat{t}_1\ \hat{t}_2) = \phi'' \to \phi'$ when $ctype_\phi(x, \hat{t}_1) = \psi'' \to (\phi'' \to \phi')$, because we know $ctype_\phi(x, \hat{t}_1) = \psi$.

A.2 Proof of Total and Type Preserving

This is a proof by induction on the lexicorgraphic combination (ϕ, t') of the strict subexpression ordering. We case split on t'.

Case. Suppose t' is either x or a variable y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : \phi_1.t_1'$. By inversion on the typing judgement we know $\Gamma, x : \phi, \Gamma', y : \phi_1 \vdash t_1' : \phi_2$. We also know $t' > t_1'$, hence we can apply the induction hypothesis to obtain $[t/x]^\phi t_1' = \hat{t}_1'$ and $\Gamma, \Gamma', y : \phi_1 \vdash \hat{t} : \phi_2$ for some term $\hat{t}_1'$. By the definition of the hereditary substitution function $[t/x]^\phi t' = \lambda y : \phi_1.[t/x]^\phi t_1' = \lambda y : \phi_1.\hat{t}_1'$. It suffices to show that $\Gamma, \Gamma' \vdash \lambda y : \phi_1.\hat{t}_1' : \phi_1 \to \phi_2$. By simply applying the λ-abstraction typing rule using $\Gamma, \Gamma', y : \phi_1 \vdash \hat{t} : \phi_2$ we obtain $\Gamma, \Gamma' \vdash \lambda y : \phi_1.\hat{t}_1' : \phi_1 \to \phi_2$.

Case. Suppose $t' \equiv t'_1\ t'_2$. By inversion we know $\Gamma, x : \phi, \Gamma' \vdash t'_1 : \phi'' \to \phi'$ and $\Gamma, x : \phi, \Gamma' \vdash t'_2 : \phi''$ for some types ϕ' and ϕ''. Clearly, $t' > t'_i$ for $i \in \{1, 2\}$. Thus, by the induction hypothesis there exists terms m_1 and m_2 such that $[t/x]^\phi t'_i = m_i$, $\Gamma, \Gamma' \vdash m_1 : \phi'' \to \phi'$ and $\Gamma, \Gamma' \vdash m_2 : \phi''$ for $i \in \{1, 2\}$. We case split on whether or not m_1 is a λ-abstraction, and t'_1 is not, or $ctype_\phi(x, t'_1)$ is undefined. We only consider the non-trivial case when $m_1 \equiv \lambda y : \phi''.m'_1$ and t'_1 is not a λ-abstraction.

Now by Lemma 1 it is the case that there exists a ψ such that $ctype_\phi(x, t'_1) = \psi$, $\psi \equiv \phi'' \to \phi'$, and ψ is a subexpression of ϕ, hence $\phi >_{\Gamma,\Gamma'} \phi''$. Then $[t/x]^\phi(t'_1\ t'_2) = [m_2/y]^{\psi''} m'_1$. Therefore, by the induction hypothesis there exists a term m such that $[m_2/y]^{\phi''} m'_1 = m$ and $\Gamma, \Gamma' \vdash m : \phi''$.

A.3 Proof of Redex Preservation

This is a proof by induction on the lexicorgraphic combination (ϕ, t') of the strict subexpression ordering. We case split on the structure of t'.

Case. Let $t' \equiv x$ or $t' \equiv y$ where y is distinct from x. Trivial.

Case. Let $t' \equiv \lambda x : \phi_1.t''$. Then $[t/x]^\phi t' \equiv \lambda x : \phi_1.[t/x]^\phi t''$. Now

$$\begin{array}{lll} rset(\lambda x : \phi_1.t'', t) & = & rset(\lambda x : \phi_1.t'') \cup rset(t) \\ & = & rset(t'') \cup rset(t) \\ & = & rset(t'', t). \end{array}$$

We know that $t' > t''$ by the strict subexpression ordering, hence by the induction hypothesis $|rset(t'', t)| \geq |rset([t/x]^\phi t'')|$ which implies $|rset(t', t)| \geq |rset([t/x]^\phi t')|$.

Case. Let $t' \equiv t'_1\ t'_2$. First consider when t'_1 is not a λ-abstraction. Then

$$rset(t'_1\ t'_2, t) = rset(t'_1, t'_2, t)$$

Clearly, $t' > t'_i$ for $i \in \{1, 2\}$, hence, by the induction hypothesis $|rset(t'_i, t)| \geq |rset([t/x]^\phi t'_i)|$. We have two cases to consider. That is whether or not $[t/x]^\phi t'_1$ is a λ-abstraction or $ctype_\phi(x, t'_1)$ is undefined. Suppose the former. Then by Lemma 1 $ctype_\phi(x.t'_1) = \psi$ and by inversion on $\Gamma, x : \phi, \Gamma' \vdash t'_1\ t'_2 : \phi'$ there exists a type ϕ'' such that $\Gamma, x : \phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$. Again, by Lemma 1 $\psi \equiv \phi'' \to \phi'$. Thus, $ctype_\phi(x, t'_1) = \phi'' \to \phi'$ and $\phi'' \to \phi'$ is a subexpression of ϕ. So by the definition of the hereditary substitution function $[t/x]^\phi t'_1\ t'_2 = [([t/x]^\phi t'_2)/y]^{\phi''} t''_1$, where $[t/x]^\phi t'_1 = \lambda y : \phi''.t''_1$. Hence,

$$|rset([t/x]^\phi t'_1\ t'_2)| = |rset([([t/x]^\phi t'_2)/y]^{\phi''} t''_1)|.$$

Now $\phi > \phi''$ so by the induction hypothesis

$$\begin{array}{lll} |rset([([t/x]^\phi t'_2)/y]^{\phi''} t''_1)| & \leq & |rset([t/x]^\phi t'_2, t''_1)| \\ & \leq & |rset(t'_2, t''_1, t)| \\ & = & |rset(t'_2, [t/x]^\phi t'_1, t)| \\ & \leq & |rset(t'_2, t'_1, t)| \\ & = & |rset(t'_1, t'_2, t)|. \end{array}$$

Suppose $[t/x]^\phi t'_1$ is not a λ-abstractions or $ctype_\phi(x, t'_1)$ is undefined. Then

$$\begin{array}{lll} rset([t/x]^\phi(t'_1\ t'_2)) & = & rset([t/x]^\phi t'_1\ [t/x]^\phi t'_2) \\ & = & rset([t/x]^\phi t'_1, [t/x]^\phi t'_2). \\ & \leq & rset(t'_1, t'_2, t). \end{array}$$

Next suppose $t'_1 \equiv \lambda y : \phi_1.t''_1$. Then

$$rset((\lambda y : \phi_1.t''_1)\ t'_2, t) \quad = \quad \{(\lambda y : \phi_1.t''_1)\ t'_2\} \cup rset(t''_1, t'_2, t).$$

By the definition of the hereditary substitution function,

$$\begin{array}{rcl} rset([t/x]^{\phi}(\lambda y : \phi_1.t''_1)\ t'_2) & = & rset([t/x]^{\phi}(\lambda y : \phi_1.t''_1)\ [t/x]^{\phi}t'_2) \\ & = & rset((\lambda y : \phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2) \\ & = & \{(\lambda y : \phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2\} \cup rset([t/x]^{\phi}t''_1) \cup rset([t/x]^{\phi}t'_2). \end{array}$$

Since $t' > t''_1$ and $t' > t'_2$ we can apply the induction hypothesis to obtain, $|rset(t''_1, t)| \geq |rset([t/x]^{\phi}t''_1)|$ and $|rset(t'_2, t)| \geq |rset([t/x]^{\phi}t'_2)|$. Therefore,
$|\{(\lambda y : \phi_1.t''_1)\ t'_2\} \cup rset(t''_1, t) \cup rset(t'_2, t)| \geq |\{(\lambda y : \phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2\} \cup rset([t/x]^{\phi}t''_1) \cup rset([t/x]^{\phi}t'_2)|$.

A.4 Proof of Normality Preservation

By Lemma 2 we know there exists a term n'' such that $[n/x]^{\phi}n' = n''$ and by Lemma 5 $|rset(n', n)| \geq |rset([n/x]^{\phi}n')|$. Hence, $|rset(n', n)| \geq |rset(n'')|$, but $|rset(n', n)| = 0$. Therefore, $|rset(t)| = 0$ which implies n'' is normal.

A.5 Proof of Soundness with Respect to Reduction

This is a proof by induction on the lexicorgraphic combination (ϕ, t') of the strict subexpression ordering. We case split on the structure of t'. When applying the induction hypothesis we must show that the input terms to the substitution and the hereditary substitution functions are typeable. We do not explicitly state typing results that are simple conseqences of inversion.

Case. Suppose t' is a variable x or y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : \phi'.\hat{t}$. Then $[t/x]^{\phi}(\lambda y : \phi'.\hat{t}) = \lambda y : \phi'.([t/x]^{\phi}\hat{t})$. Now $t' > \hat{t}$ so we can apply the induction hypothesis to obtain $[t/x]\hat{t} \rightsquigarrow^* [t/x]^{\phi}\hat{t}$. At this point we can see that since $\lambda y : \phi'.[t/x]\hat{t} \equiv [t/x](\lambda y : \phi'.\hat{t})$ and we may conclude that $\lambda y : \phi'.[t/x]\hat{t} \rightsquigarrow^* \lambda y : \phi'.[t/x]^{\phi}\hat{t}$.

Case. Suppose $t' \equiv t'_1\ t'_2$. By Lemma 2 there exists terms $\hat{t}'_1$ and $\hat{t}'_2$ such that $[t/x]^{\phi}t'_1 = \hat{t}'_1$ and $[t/x]^{\phi}t'_2 = \hat{t}'_2$. Since $t' > t'_1$ and $t' > t'_2$ we can apply the induction hypothesis to obtain $[t/x]t'_1 \rightsquigarrow^* \hat{t}'_1$ and $[t/x]t'_2 \rightsquigarrow^* \hat{t}'_2$. Now we case split on whether or not $\hat{t}'_1$ is a λ-abstraction and t'_1 is not or $ctype_{\phi}(x, t'_1)$ is undefined. If $ctype_{\phi}(x, t'_1)$ is undefined or $\hat{t}'_1$ is not a λ-abstraction then $[t/x]^{\phi}t' = ([t/x]^{\phi}t'_1)\ ([t/x]^{\phi}t'_2) \equiv \hat{t}'_1\ \hat{t}'_2$. Thus, $[t/x]t' \rightsquigarrow^* [t/x]^{\phi}t'$, because $[t/x]t' = ([t/x]t'_1)\ ([t/x]t'_2)$. So suppose $\hat{t}'_1 \equiv \lambda y : \phi'.\hat{t}''_1$ and t'_1 is not a λ-abstraction. By Lemma 1 there exists a type ψ such that $ctype_{\phi}(x, t'_1) = \psi$, $\psi \equiv \phi'' \rightarrow \phi'$, and ψ is a subexpression of ϕ, where by inversion on $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$ there exists a type ϕ'' such that $\Gamma, x : \phi, \Gamma' \vdash t'_1 : \phi'' \rightarrow \phi'$. Then by the definiton of the hereditary substitution function $[t/x]^{\phi}(t'_1\ t'_2) = [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$. Now we know $\phi > \phi'$ so we can apply the induction hypothesis to obtain $[\hat{t}'_2/y]\hat{t}''_1 \rightsquigarrow^* [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$. Now by knowing that $(\lambda y : \phi'.\hat{t}''_1)\ t'_2 \rightsquigarrow [\hat{t}'_2/y]\hat{t}''_1$ and by the previous fact we know $(\lambda y : \phi'.\hat{t}''_1)\ t'_2 \rightsquigarrow^* [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$. We now make use of the well known result of full β-reduction. The result is stated as

$$\frac{\begin{array}{l} a \rightsquigarrow^* a' \\ b \rightsquigarrow^* b' \qquad a'\ b' \rightsquigarrow^* c \end{array}}{a\ b \rightsquigarrow^* c}$$

where a, a', b, b', and c are all terms. We apply this result by instantiating a, a', b, b', and c with $[t/x]t'_1$, $\hat{t}'_1$, $[t/x]t'_2$, $\hat{t}'_2$, and $[\hat{t}'_2/y]^{\phi'}\hat{t}''_1$ respectively. Therefore, $[t/x](t'_1\ t'_2) \rightsquigarrow^* [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$.

A.6 Proof of Type Soundness

This is a proof by induction on the structure of the typing derivation of t.

Case.

$$\frac{\Gamma(x) = \phi \qquad \Gamma\ Ok}{\Gamma \vdash x : \phi}$$

By regularity $\Gamma \vdash \phi : *_l$ for some l, hence $[\![\phi]\!]_\Gamma$ is nonempty. Clearly, $x \in [\![\phi]\!]_\Gamma$ by the definition of the interpretation of types.

Case.

$$\frac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x : \phi_1.t : \phi_1 \to \phi_2}$$

By the induction hypothesis $t \in [\![\phi_2]\!]_{\Gamma,x:\phi_1}$ and by the definition of the interpretation of types $t \rightsquigarrow^! n \in [\![\phi_2]\!]_{\Gamma,x:\phi_1}$ and $\Gamma, x : \phi_1 \vdash n : \phi_2$. Thus, by applying the λ-abstraction type-checking rule, $\Gamma \vdash \lambda x : \phi_1.n : \Pi x : \phi_1.\phi_2$ so by the definition of the interpretation of types $\lambda x : \phi_1.n \in [\![\phi_1 \to \phi_2]\!]_\Gamma$. Thus, according to the definition of the interpretation of types $\lambda x : \phi_1.t \rightsquigarrow^! \lambda x : \phi_1.n \in [\![\phi_1 \to \phi_2]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash t_1 : \phi_2 \to \phi_1 \qquad \Gamma \vdash t_2 : \phi_2}{\Gamma \vdash t_1\ t_2 : \phi_1}$$

By the induction hypothesis $t_1 \rightsquigarrow^! n_1 \in [\![\phi_2 \to \phi_1]\!]_\Gamma$, and $t_2 \rightsquigarrow^! n_2 \in [\![\phi_2]\!]_\Gamma$.

Now we know from above that $n_1 \in [\![\phi_2 \to \phi_1]\!]_\Gamma$ and $n_2 \in [\![\phi_2]\!]_\Gamma$, hence $\Gamma \vdash n_1 : \phi_2 \to \phi_1$ and $\Gamma \vdash n_2 : \phi_2$. It suffices to show that $n_1\ n_2 \in [\![\phi_2]\!]_\Gamma$. Clearly, $n_1\ n_2 = [n_1/z](z\ n_2)$ for some variable $z \notin FV(n_1, n_2)$. Lemma 2, Lemma 4, and Lemma 3 allow us to conclude that $[n_1/z](z\ n_2) \rightsquigarrow^* [n_1/z]^{\phi_2 \to \phi_1}(z\ n_2)$, $\Gamma \vdash [n_1/z]^{\phi_2 \to \phi_1}(z\ n_2) : \phi_2$, and $[n_1/z]^{\phi_2 \to \phi_1}(z\ n_2)$ is normal. Thus, $t_1\ t_2 \rightsquigarrow^* n_1\ n_2 = [n_1/z](z\ n_2) \rightsquigarrow^! [n_1/z]^{\phi_2 \to \phi_1}(z\ n_2) \in [\![\phi_2]\!]_\Gamma$.

B Proofs of Results Pertaining to SSF

B.1 Proof of Lemma 8

This is a proof by structural induction on the kinding derivation of $\Gamma \vdash \phi : *_p$.

Case.

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\ Ok}{\Gamma \vdash X : *_q}$$

By inversion of the kind-checking rule $\Gamma\ Ok$.

Case.

$$\frac{\Gamma \vdash \phi_1 : *_p \qquad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{max(p,q)}}$$

By the induction hypothesis, $\Gamma \vdash \phi_1 : *_p$ and $\Gamma \vdash \phi_2 : *_q$ both imply $\Gamma\ Ok$. Since the arrow-type kind-checking rule does not modify Γ in anyway Γ will remain Ok.

Case.

$$\frac{\Gamma, X : *_q \vdash \phi : *_p}{\Gamma \vdash \forall X : *_q.\phi : *_{max(p,q)+1}}$$

By the induction hypothesis $\Gamma, X : *_p\ Ok$, and by inversion of the type-variable well-formed contexts rule $\Gamma\ Ok$.

B.2 Proof of Lemma 9

We show level weakening for kinding by structural induction on the kinding derivation of $\phi : *_r$.

Case.

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\ Ok}{\Gamma \vdash X : *_q}$$

By assumption we know $q < s$, hence by reapplying the rule and transitivity we obtain $\Gamma \vdash X : *_s$.

Case.

$$\frac{\Gamma \vdash \phi_1 : *_p \qquad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{max(p,q)}}$$

By the induction hypothesis $\Gamma \vdash \phi_1 : *_s$ and $\Gamma \vdash \phi_2 : *_s$ for some arbitrary $s > max(p,q)$. Therefore, by reapplying the rule we obtain $\Gamma \vdash \phi_1 \to \phi_2 : *_s$.

Case.

$$\frac{\Gamma, X : *_q \vdash \phi' : *_p}{\Gamma \vdash \forall X : *_q.\phi' : *_{max(p,q)+1}}$$

We know by assumption that $max(p,q) + 1 < s$ which implies that $max(p,q) < s - 1$. Now by the induction hypothesis $\Gamma, X : *_q \vdash \phi' : *_{s-1}$. Lastly, we reapply the rule and obtain $\Gamma \vdash \forall X : *_q.\phi' : *_s$.

B.3 Proof of Substitution for Kinding, Context-Ok

This is a prove by induction on d. We prove the first implication first, and then the second, doing a case analysis for each implication on the form of the derivation whose depth is being considered.

Case.

$$\frac{(\Gamma, X : *_p, \Gamma')(Y) = *_r \qquad r \leq s \qquad \Gamma, X : *_p, \Gamma'\ Ok}{\Gamma, X : *_p, \Gamma' \vdash Y : *_s}$$

By assumption $\Gamma \vdash \phi' : *_p$. We must consider whether or not $X \equiv Y$. If $X \equiv Y$ then $[\phi'/X]Y \equiv \phi'$, $r = p$, and $q = s$; this conclusion is equivalent to $\Gamma, [\phi'/X]\Gamma' \vdash \phi' : *_q$ and by the induction hypothesis $\Gamma, [\phi'/X]\Gamma'\ Ok$. If $X \not\equiv Y$ then $[\phi'/X]Y \equiv Y$ and by the induction hypothesis $\Gamma, [\phi'/X]\Gamma'\ Ok$, hence, $\Gamma, [\phi'/X]\Gamma' \vdash Y : *_q$.

Case.

$$\frac{\Gamma, X : *_p, \Gamma' \vdash \phi_1 : *_r \qquad \Gamma, X : *_p, \Gamma' \vdash \phi_2 : *_s}{\Gamma, X : *_p, \Gamma' \vdash \phi_1 \to \phi_2 : *_{max(r,s)}}$$

Here $q = max(r, s)$ and by the induction hypothesis $\Gamma, [\phi'/X]\Gamma' \vdash [\phi'/X]\phi_1 : *_r$ and $\Gamma, [\phi'/X]\Gamma' \vdash [\phi'/X]\phi_2 : *_s$. We can now reapply the rule to get $\Gamma, [\phi'/X]\Gamma' \vdash [\phi'/X](\phi_1 \to \phi_2) : *_q$.

Case.

$$\frac{\Gamma, X : *_q, \Gamma', Y : *_r \vdash \phi : *_s}{\Gamma, X : *_p, \Gamma' \vdash \forall Y : *_r.\phi : *_{max(r,s)+1}}$$

Here $q = max(r, s) + 1$ and by the induction hypothesis $\Gamma, [\phi'/X]\Gamma', Y : *_r \vdash [\phi'/X]\phi : *_s$. We can reapply this rule to get $\Gamma, [\phi'/X]\Gamma' \vdash [\phi'/X]\forall Y : *_r.\phi : *_q$.

We now show the second implication. The case were $d = 0$ cannot arise, since it requires the context to be empty. Suppose $d = n + 1$. We do a case analysis on the last rule applied in the derivation of $\Gamma, X : *_p, \Gamma'$.

Case. Suppose $\Gamma' = \Gamma'', Y : *_q$.

$$\frac{\Gamma, X : *_p, \Gamma''\ Ok}{\Gamma, X : *_p, \Gamma'', Y : *_q\ Ok}$$

By the induction hypothesis, $\Gamma, [\phi'/X]\Gamma''\ Ok$. Now, by reapplying the rule above $\Gamma, [\phi'/X]\Gamma'', Y : *_q\ Ok$, hence $\Gamma, [\phi'/X]\Gamma'\ Ok$, since $X \not\equiv Y$.

Case. Suppose $\Gamma' = \Gamma'', y : \phi$.

$$\frac{\Gamma, X : *_p, \Gamma'' \vdash \phi : *_q \qquad \Gamma, X : *_p, \Gamma''\ Ok}{\Gamma, X : *_p, \Gamma'', y : \phi\ Ok}$$

By the induction hypothesis, $\Gamma', [\phi'/X]\Gamma'' \vdash [\phi'/X]\phi : *_q$ and $\Gamma', [\phi'/X]\Gamma''\ Ok$. Thus, by reapplying the rule above $\Gamma, [\phi'/X]\Gamma'', x : [\phi'/X]\phi\ Ok$, therefore,
$\Gamma, [\phi'/X]\Gamma'\ Ok$.

B.4 Proof of Context Strengthening for Kinding, Context-Ok

This is a prove by induction on d. We prove the first implication first, and then the second, doing a case analysis for each implication on the form of the derivation whose depth is being considered.

Case.

$$\frac{(\Gamma, x : \phi', \Gamma')(X) = *_p \qquad p \leq q \qquad \Gamma, x : \phi', \Gamma' \; Ok}{\Gamma, x : \phi', \Gamma' \vdash X : *_q}$$

By the second implication of the induction hypothesis, $\Gamma, \Gamma' \; Ok$. Also, $(\Gamma, \Gamma')(X) = *_p$. Now by reapplying the rule above, $\Gamma, \Gamma' \vdash X : *_q$.

Case.

$$\frac{\Gamma, x : \phi', \Gamma' \vdash \phi_1 : *_p \qquad \Gamma, x : \phi', \Gamma' \vdash \phi_2 : *_q}{\Gamma, x : \phi', \Gamma' \vdash \phi_1 \to \phi_2 : *_{max(p,q)}}$$

By the first implication of the induction hypothesis, $\Gamma, \Gamma' \vdash \phi_1 : *_p$ and $\Gamma, \Gamma' \vdash \phi_2 : *_q$. By reapplying the rule above we get, $\Gamma, \Gamma' \vdash \phi_1 \to \phi_2 : *_{max(p,q)}$.

Case.

$$\frac{\Gamma, x : \phi, \Gamma', Y : *_q \vdash \phi : *_p}{\Gamma, x : \phi', \Gamma' \vdash \forall Y : *_q.\phi : *_{max(p,q)+1}}$$

By the first implication of the induction hypothesis, $\Gamma, \Gamma', Y : *_q \vdash \phi : *_p$. By reapplying the rule we get, $\Gamma, \Gamma' \vdash \forall Y : *_q.\phi : *_{max(p,q)+1}$.

We now prove the second implication. The case where $d = 0$ cannot arise, since it requires the context to be empty. Suppose $d = n + 1$. We do a case analysis on the last rule applied in the derivation of $\Gamma, x : \phi, \Gamma' \; Ok$.

Case. Suppose $\Gamma' = \Gamma'', Y : *_l$. Then the last rule of the derivation of $\Gamma, x : \phi, \Gamma' \; Ok$ is as follows.

$$\frac{\Gamma, x : \phi, \Gamma'' \; Ok}{\Gamma, x : \phi, \Gamma'', Y : *_l \; Ok}$$

By the second implication of the induction hypothesis, $\Gamma, \Gamma'' \; Ok$. Now reapplying the rule we get, $\Gamma, \Gamma'', Y : *_l \; Ok$, which is equivalent to $\Gamma, \Gamma' \; Ok$.

Case. Suppose $\Gamma' = \Gamma'', y : \phi'$. Then the last rule of the derivation of $\Gamma, x : \phi, \Gamma' \; Ok$ is as follows.

$$\frac{\Gamma, x : \phi, \Gamma'' \vdash \phi' : *_p \qquad \Gamma, x : \phi, \Gamma'' \; Ok}{\Gamma, x : \phi, \Gamma'', y : \phi' \; Ok}$$

By the first implication of the induction hypothesis, $\Gamma, \Gamma'' \vdash \phi' : *_p$ and by the second, $\Gamma, \Gamma'' \; Ok$. Therefore, by reapplying the rule above, $\Gamma, \Gamma'', y : \phi' \; Ok$, which is equivalent to $\Gamma, \Gamma' \; Ok$.

B.5 Proof of Context Weakening for Kinding

This is a proof by structural induction on the kinding derivation of $\Gamma, \Gamma' \vdash \phi : *_p$.

Case.

$$\frac{(\Gamma, \Gamma')(X) = *_p \qquad p \leq q \qquad \Gamma, \Gamma' \; Ok}{\Gamma, \Gamma' \vdash X : *_q}$$

If $(\Gamma, \Gamma')(X) = *_p$ then $(\Gamma, \Gamma'', \Gamma')(X) = *_p$, hence, by reapplying the type-variable kind-checking rule, $\Gamma, \Gamma'', \Gamma' \vdash \phi : *_p$.

Case.

$$\frac{\Gamma, \Gamma' \vdash \phi_1 : *_p \qquad \Gamma, \Gamma' \vdash \phi_2 : *_q}{\Gamma, \Gamma' \vdash \phi_1 \rightarrow \phi_2 : *_{max(p,q)}}$$

By the induction hypothesis $\Gamma, \Gamma'', \Gamma' \vdash \phi_1 : *_p$ and $\Gamma, \Gamma'', \Gamma' \vdash \phi_2 : *_q$, hence, by reapplying the arrow-type kind-checking rule $\Gamma, \Gamma'', \Gamma'' \vdash \phi_1 \rightarrow \phi_2 : *_{max(p,q)}$.

Case.

$$\frac{\Gamma, \Gamma', X : *_q \vdash \phi' : *_p}{\Gamma, \Gamma' \vdash \forall X : *_q.\phi' : *_{max(p,q)+1}}$$

By the induction hypothesis $\Gamma, \Gamma'', \Gamma', X : *_p \vdash \phi : *_q$, hence, by reapplying the forall-type kind-checking rule $\Gamma, \Gamma'', \Gamma' \vdash \forall X : *_p.\phi : *_{max(p,q)+1}$.

B.6 Proof of Regularity

This proof is by structural induction on the derivation of $\Gamma \vdash t : \phi$.

Case.

$$\frac{\Gamma(x) = \phi \qquad \Gamma \; Ok}{\Gamma \vdash x : \phi}$$

By the definition of well-formedness contexts $\Gamma \vdash \phi : *_p$ for some p.

Case.

$$\frac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x : \phi_1.t : \phi_1 \rightarrow \phi_2}$$

By the induction hypothesis $\Gamma \vdash \phi_1 : *_p$, $\Gamma, x : \phi_1 \vdash \phi_2 : *_q$ and by Lemma **??**, $\Gamma \vdash \phi_2 : *_q$. By applying the arrow-type kind-checking rule we get $\Gamma \vdash \phi_1 \rightarrow \phi_2 : *_{max(p,q)}$.

Case.

$$\frac{\Gamma \vdash t_1 : \phi_1 \to \phi_2 \qquad \Gamma \vdash t_2 : \phi_1}{\Gamma \vdash t_1\ t_2 : \phi_2}$$

By the induction hypothesis $\Gamma \vdash \phi_1 \to \phi_2 : *r$ and $\Gamma \vdash \phi_1 : *_p$. By inversion of the arrow-type kind-checking rule $r = max(p, q)$, for some q, which implies $\Gamma \vdash \phi_2 : *_q$.

Case.

$$\frac{\Gamma, X : *_p \vdash t : \phi}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_q.\phi}$$

By the induction hypothesis $\Gamma, X : *_q \vdash \phi : *_p$. By applying the forall-type kind-checking rule $\Gamma \vdash \forall X.\phi : *_{max(p,q)+1}$.

Case.

$$\frac{\Gamma \vdash t : \forall X : *_p.\phi_1 \qquad \Gamma \vdash \phi_2 : *_p}{\Gamma \vdash t[\phi_2] : [\phi_2/X]\phi_1}$$

By assumption $\Gamma \vdash \phi_2 : *_r$. By the induction hypothesis $\Gamma \vdash \forall X : *_p.\phi_1 : *_s$ and by inversion of the forall-type kind-checking rule $r = max(p, q) + 1$, for some q, which implies $\Gamma, X : *_p \vdash \phi_1 : *_q$. Now, by Lemma **??**, $\Gamma \vdash [\phi_2/X]\phi_1 : *_q$.

B.7 Proof of Lemma 14

Suppose $\phi >_\Gamma \phi'$ and $\phi' >_\Gamma \phi''$. If $\phi \equiv \phi_1 \to \phi_2$ then, ϕ' must be a subexpression of ϕ. Now if $\phi' \equiv \phi'_1 \to \phi'_2$ then, ϕ'' must be a subexpression of ϕ', which implies that ϕ'' is a subexpression of ϕ. Thus, $\phi >_\Gamma \phi''$. If $\phi' \equiv \forall X : *_l.\phi'_1$ then, there exists a type ϕ'_2 where, $\Gamma \vdash \phi'_2 : *_l$, such that, $\phi'' \equiv [\phi'_2/X]\phi'_1$. The level of ϕ' is $max(l, l') + 1$, where l' is the level of ϕ'_1, the level of ϕ'' is $max(l, l')$, and the level of ϕ is $max(max(l, l') + 1, p)$, where p is the level of the type, which is, the second subexpression of ϕ. Clearly, $max(max(l, l') + 1, p) \geq max(l, l')$, thus, $\phi >_\Gamma \phi''$.

If $\phi \equiv \forall X : *_l.\phi_1$, then $\phi' \equiv [\phi_2/X]\phi_1$ for some type ϕ_2, where $\Gamma \vdash \phi_2 : *_l$. If $[\phi_2/X]\phi_1 \equiv \phi'_1 \to \phi'_2$ then the level of ϕ' is $max(p, q)$, where p is the level of ϕ'_1 and q is the level of ϕ'_2. Now ϕ'' must be a subexpression of ϕ', hence the level of ϕ'' is either p or q. Now, since the level of ϕ is greater than the level of ϕ' and we know, $max(p, q)$ is greater than both p and q then $\phi >_\Gamma \phi''$. If $[\phi_2/X]\phi_1 \equiv \forall Y : *_{l'}.\phi'_1$, then $\phi'' \equiv [\phi'_2/X]\phi'_1$. Now if p is the level of ϕ_1, then the level of ϕ is $max(l, p) + 1$ and the level of ϕ' must be $max(l, p)$ since we know the level of ϕ' is greater than the level of ϕ'' then clearly, the level of ϕ is greater than the level of ϕ''. Thus, $\phi >_\Gamma \phi''$.

B.8 Proof of Lemma 15

Assume $\phi >_\Gamma \phi'$ for some types ϕ and ϕ'. We case split on the form of ϕ. Clearly, ϕ is not a type variable.

Case. Suppose $\phi \equiv \phi_1 \to \phi_2$. Then ϕ' must be of the form ϕ_1 or ϕ_2. In both cases we have two cases to consider; either ϕ and ϕ' have the same level or they do not. Consider the first form and suppose they have the same level. Then it is clear that $depth(\phi) > depth(\phi')$. Now consider the latter form and suppose ϕ and ϕ' have the same level. Then, clearly, $depth(\phi) > depth(\phi')$. In either form if the level of ϕ and ϕ' are different, then the level of ϕ is larger than the level of ϕ'. In all cases $(l, d) > (l', d')$.

Case. Suppose $\phi \equiv \forall X : *_l.\phi_1$. Then ϕ' must be of the form $[\phi_2/X]\phi_1$ for some type $\Gamma \vdash \phi_2 : *_l$. It is obvious that the level of ϕ is always larger than the level of ϕ'. Hence, $(l, d) > (l', d')$.

B.9 Proof of Theroem 2

If there exists a infinite decreasing sequence using our ordering on types, then there is an infinite decreasing sequence using our measure by Lemma 15, but that is impossible.

B.10 Proof of Lemma 16

We prove part one first. This is a proof by induction on the structure of t.

Case. Suppose $t \equiv x$. Then $ctype_\phi(x, x) = \phi$. Clearly, $head(x) = x$ and $\phi \equiv \phi$.

Case. Suppose $t \equiv t_1\, t_2$. Then $ctype_\phi(x, t_1\, t_2) = \phi''$ when $ctype_\phi(x, t_1) = \phi' \to \phi''$. Now $t > t_1$ so by the induciton hypothesis $head(t_1) = x$ and $\phi' \to \phi'' \leq_{\Gamma,\Gamma'} \phi$. Therefore, $head(t_1\ t_2) = x$, and certainly, $\phi'' \leq_{\Gamma,\Gamma'} \phi$.

Next we prove part two. This is a proof by induction on the structure of $t_1\ t_2$.

The only possiblities for the form of t_1 is x, $\hat{t}_1\ \hat{t}_2$, or $\hat{t}[\phi'']$. All other forms would not result in $[t/x]^\phi t_1$ being a λ-abstraction and t_1 not. If $t_1 \equiv x$ then there exist a type ϕ'' such that $\phi \equiv \phi'' \to \phi'$ and $ctype_\phi(x, x\ t_2) = \phi'$ when $ctype_\phi(x, x) = \phi \equiv \phi'' \to \phi'$ in this case. We know ϕ'' to exist by inversion on $\Gamma, x : \phi, \Gamma' \vdash t_1\ t_2 : \phi'$.

Now suppose $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$. Now knowing t_1' to not be a λ-abstraction implies that $\hat{t}_1$ is also not a λ-abstraction or $[t/x]^\phi t_1$ would be an application instead of a λ-abstraction. So it must be the case that $[t/x]^\phi \hat{t}_1$ is a λ-abstraction and $\hat{t}_1$ is not. Since $t_1\ t_2 > t_1$ we can apply the induction hypothesis to obtain there exists a type ψ such that $ctype_\phi(x, \hat{t}_1) = \psi$. Now by inversion on $\Gamma, x : \phi, \Gamma' \vdash t_1\ t_2 : \phi'$ we know there exists a type ϕ'' such that $\Gamma, x : \phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$. We know $t_1 \equiv (\hat{t}_1\ \hat{t}_2)$ so by inversion on $\Gamma, x : \phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$ we know there exists a type ψ'' such that $\Gamma, x : \phi, \Gamma' \vdash \hat{t}_1 : \psi'' \to (\phi'' \to \phi')$. By part two of Lemma 16 we know $\psi \equiv \psi'' \to (\phi'' \to \phi')$ and $ctype_\phi(x, t_1) = ctype_\phi(x, \hat{t}_1\ \hat{t}_2) = \phi'' \to \phi'$ when $ctype_\phi(x, \hat{t}_1) = \psi'' \to (\phi'' \to \phi')$, because we know $ctype_\phi(x, \hat{t}_1) = \psi$.

The case where t_1 is a type application is similar to the previous case.

The remaining parts of the lemma are similar to part two.

B.11 Proof of Totality and Type Preservation

This is a proof by induction on the lexicorgraphic combination (ϕ, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on t'.

Case. Suppose t' is either x or a variable y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : \phi_1.t_1'$. By inversion on the typing judgement we know $\Gamma, x : \phi, \Gamma', y : \phi_1 \vdash t_1' : \phi_2$. We also know $t' > t_1'$, hence we can apply the induction hypothesis to obtain $[t/x]^\phi t_1' = \hat{t}_1'$ and $\Gamma, \Gamma', y : \phi_1 \vdash \hat{t} : \phi_2$ for some term $\hat{t}_1'$. By the definition of the hereditary substitution function $[t/x]^\phi t' = \lambda y : \phi_1.[t/x]^\phi t_1' = \lambda y : \phi_1.\hat{t}_1'$. It suffices to show that $\Gamma, \Gamma' \vdash \lambda y : \phi_1.\hat{t}_1' : \phi_1 \to \phi_2$. By simply applying the λ-abstraction typing rule using $\Gamma, \Gamma', y : \phi_1 \vdash \hat{t} : \phi_2$ we obtain $\Gamma, \Gamma' \vdash \lambda y : \phi_1.\hat{t}_1' : \phi_1 \to \phi_2$.

Case. Suppose $t' \equiv \Lambda X : *_l.t_1'$. Similar to the previous case.

Case. Suppose $t' \equiv t_1'\ t_2'$. By inversion we know $\Gamma, x : \phi, \Gamma' \vdash t_1' : \phi'' \to \phi'$ and $\Gamma, x : \phi, \Gamma' \vdash t_2' : \phi''$ for some types ϕ' and ϕ''. Clearly, $t' > t_i'$ for $i \in \{1, 2\}$. Thus, by the induction hypothesis there exists terms m_1 and m_2 such that $[t/x]^\phi t_i' = m_i$, $\Gamma, \Gamma' \vdash m_1 : \phi'' \to \phi'$ and $\Gamma, \Gamma' \vdash m_2 : \phi''$ for $i \in \{1, 2\}$. We case split on whether or not m_1 is a λ-abstraction and t_1' is not, or $ctype_\phi(x, t_1')$ is undefined. We only consider the non-trivial cases when $m_1 \equiv \lambda y : \phi''.m_1'$, t_1' is not a λ-abstraction, and $ctype_\phi(x, t_1') = \psi'' \to \psi'$. Suppose the former. Now by

Lemma 16 it is the case that there exists a ψ such that $ctype_\phi(x, t'_1) = \psi$, $\psi \equiv \phi'' \to \phi'$, and $\psi \leq_{\Gamma,\Gamma'} \phi$, hence $\phi >_{\Gamma,\Gamma'} \phi''$. Then $[t/x]^\phi(t'_1\ t'_2) = [m_2/y]^{\psi''} m'_1$. Therefore, by the induction hypothesis there exists a term m such that $[m_2/y]^{\phi''} m'_1 = m$ and $\Gamma, \Gamma' \vdash m : \phi''$.

Case. Suppose $t' \equiv t'_1[\phi'']$. Similar to the previous case.

B.12 Proof of Redex Preservation

This is a proof by induction on the lexicorgraphic combination (ϕ, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on the structure of t'.

Case. Let $t' \equiv x$ or $t' \equiv y$ where y is distinct from x. Trivial.

Case. Let $t' \equiv \lambda x : \phi_1.t''$. Then $[t/x]^\phi t' \equiv \lambda x : \phi_1.[t/x]^\phi t''$. Now

$$\begin{array}{rcl} rset(\lambda x : \phi_1.t'', t) & = & rset(\lambda x : \phi_1.t'') \cup rset(t) \\ & = & rset(t'') \cup rset(t) \\ & = & rset(t'', t). \end{array}$$

We know that $t' > t''$ by the strict subexpression ordering, hence by the induction hypothesis $|rset(t'', t)| \geq_{\Gamma,\Gamma'} |rset([t/x]^\phi t'')|$ which implies $|rset(t', t)| \geq |rset([t/x]^\phi t')|$.

Case. Let $t' \equiv \Lambda X : *_l.t''$. Similar to the previous case.

Case. Let $t' \equiv inl(t'')$. We know $rset(t', t) = rset(t'', t)$. Since $t' > t''$ we can apply the induction hypothesis to obtain $|rset(t'', t)| \geq |rset([t/x]^\phi t'')|$. This implies $|rset(t', t)| \geq_{\Gamma,\Gamma'} |rset([t/x]^\phi t')|$.

Case. Let $t' \equiv inr(t'')$. Similar to the previous case.

Case. Let $t' \equiv t'_1\ t'_2$. First consider when t'_1 is not a λ-abstraction. Then

$$rset(t'_1\ t'_2, t) = rset(t'_1, t'_2, t)$$

Clearly, $t' > t'_i$ for $i \in \{1, 2\}$, hence, by the induction hypothesis $|rset(t'_i, t)| \geq |rset([t/x]^\phi t'_i)|$. We have two cases to consider. That is whether or not $[t/x]^\phi t'_1$ is a λ-abstraction or not. Suppose so. Then by Lemma 16 $ctype_\phi(x.t'_1) = \psi$ and by inversion on $\Gamma, x : \phi, \Gamma' \vdash t'_1\ t'_2 : \phi'$ there exists a type ϕ'' such that $\Gamma, x : \phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$. Again, by Lemma 16 $\psi \equiv \phi'' \to \phi'$. Thus, $ctype_\phi(x, t'_1) = \phi'' \to \phi'$ and $\phi'' \to \phi'$ is a subexpression of ϕ. So by the definition of the hereditary substitution function $[t/x]^\phi t'_1\ t'_2 = [([t/x]^\phi t'_2)/y]^{\phi''} t''_1$, where $[t/x]^\phi t'_1 = \lambda y : \phi''.t''_1$. Hence,

$$|rset([t/x]^\phi t'_1\ t'_2)| = |rset([([t/x]^\phi t'_2)/y]^{\phi''} t''_1)|.$$

Now $\phi >_{\Gamma,\Gamma'} \phi''$ so by the induction hypothesis

$$\begin{array}{rcl} |rset([([t/x]^\phi t'_2)/y]^{\phi''} t''_1)| & \leq & |rset([t/x]^\phi t'_2, t''_1)| \\ & \leq & |rset(t'_2, t''_1, t)| \\ & = & |rset(t'_2, [t/x]^\phi t'_1, t)| \\ & \leq & |rset(t'_2, t'_1, t)| \\ & = & |rset(t'_1, t'_2, t)|. \end{array}$$

Suppose $[t/x]^\phi t'_1$ is not a λ-abstractions or $ctype_\phi(x, t'_1)$ is undefined. Then

$$\begin{array}{rcl} rset([t/x]^\phi(t'_1\ t'_2)) & = & rset([t/x]^\phi t'_1\ [t/x]^\phi t'_2) \\ & = & rset([t/x]^\phi t'_1, [t/x]^\phi t'_2). \\ & \geq & rset(t'_1, t'_2, t). \end{array}$$

Next suppose $t'_1 \equiv \lambda y : \phi_1.t''_1$. Then

$$rset((\lambda y : \phi_1.t''_1)\ t'_2, t) \quad = \quad \{(\lambda y : \phi_1.t''_1)\ t'_2\} \cup rset(t''_1, t'_2, t).$$

By the definition of the hereditary substitution function,

$$\begin{array}{lll} rset([t/x]^{\phi}(\lambda y : \phi_1.t''_1)\ t'_2) & = & rset([t/x]^{\phi}(\lambda y : \phi_1.t''_1)\ [t/x]^{\phi}t'_2) \\ & = & rset((\lambda y : \phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2) \\ & = & \{(\lambda y : \phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2\} \cup rset([t/x]^{\phi}t''_1) \cup rset([t/x]^{\phi}t'_2). \end{array}$$

Since $t' > t''_1$ and $t' > t'_2$ we can apply the induction hypothesis to obtain, $|rset(t''_1, t)| \geq |rset([t/x]^{\phi}t''_1)|$ and $|rset(t'_2, t)| \geq |rset([t/x]^{\phi}t'_2)|$. Therefore,
$|\{(\lambda y : \phi_1.t''_1)\ t'_2\} \cup rset(t''_1, t) \cup rset(t'_2, t)| \geq |\{(\lambda y : \phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2\} \cup rset([t/x]^{\phi}t''_1) \cup rset([t/x]^{\phi}t'_2)|$.

Case. Suppose $t' \equiv t'_1[\phi'']$. It suffices to show that $|rset(t, t')| \geq |rset([t/x]^{\phi}t')|$. Now

$$\begin{array}{lll} |rset(t, t')| & = & |rset(t, t'_1[\phi''])| \\ & = & |rset(t) \cup rset(t'_1[\phi''])| \\ & = & |rset(t) \cup rset(t'_1)| \\ & = & |rset(t, t'_1)|. \end{array}$$

and

$$|rset[t/x]^{\phi}t')| = |rset([t/x]^{\phi}(t'_1[\phi'']))|.$$

We have several cases to consider. Suppose t'_1 and $[t/x]^{\phi}t'_1$ are not type abstractions. Then

$$\begin{array}{lll} |rset([t/x]^{\phi}(t'_1[\phi'']))| & = & |rset(([t/x]^{\phi}t'_1)[\phi''])| \\ & = & |rset([t/x]^{\phi}t'_1)|. \end{array}$$

We can see that $t' > t'_1$ so by the induction hypothesis

$$\begin{array}{lll} |rset([t/x]^{\phi}t'_1)| & \leq & |rset(t, t'_1)| \\ & = & |rset(t, t')|. \end{array}$$

Suppose $t'_1 \equiv \Lambda X : *_l.t''_1$. Then

$$\begin{array}{lll} |rset(t, t')| & = & |rset(t, t'_1[\phi''])| \\ & = & |\{t'_1[\phi'']\} \cup rset(t, t''_1)| \end{array}$$

and

$$\begin{array}{lll} |rset([t/x]^{\phi}t')| & = & |rset([t/x]^{\phi}(t'_1[\phi''])| \\ & = & |rset((\Lambda X : *_l.[t/x]^{\phi}t''_1)[\phi''])| \\ & = & |\{(\Lambda X : *_l.[t/x]^{\phi}t''_1)[\phi'']\} \cup rset([t/x]^{\phi}t''_1)|. \end{array}$$

Again, $t' > t'_1$ so by the inducitom hypothesis $|rset([t/x]^{\phi}t''_1)| \leq |rset(t, t''_1)$. Thus, $|rset(t, t')| \geq |rset([t/x]^{\phi}t')|$.

Suppose t'_1 is not a type abstraction, but $[t/x]^{\phi}t'_1 \equiv \lambda X : *_l.t''_1$. Then

$$\begin{array}{lll} |rset([t/x]^{\phi}t')| & = & |rset([\phi''/X]t''_1)| \\ & = & |rset(t''_1)| \end{array}$$

and

$$\begin{array}{lll} |rset(t',t)| & = & |rset(t'_1[\phi''],t)| \\ & = & |rset(t'_1,t)|. \end{array}$$

Since $t' > t'_1$ we can apply the induction hypthothesis to obtain

$$\begin{array}{l} |rset([t/x]^{\phi}t'_1)| = |rset(t''_1) \\ \leq |rset(t'_1,t)|. \end{array}$$

Therefore, $|rset([t/x]^{\phi}t')| \leq |rset(t',t)|$.

Case. Let $t' \equiv t'_1\ t'_2$. First consider when t'_1 is not a λ-abstraction. Then

$$rset(t'_1\ t'_2,t) = rset(t'_1,t'_2,t)$$

Clearly, $t' > t'_i$ for $i \in \{1,2\}$, hence, by the induction hypothesis $|rset(t'_i,t)| \geq |rset([t/x]^{\phi}t'_i)|$. We have three cases to consider. That is whether or not $[t/x]^{\phi}t'_1$ is a λ-abstraction and t'_1 is not, or $ctype_{\phi}(x,t'_1)$ is undefined. Suppose t'_1 is a λ-abstraction. Then by Lemma 16 $ctype_{\phi}(x.t'_1) = \psi$ and by inversion on $\Gamma, x:\phi, \Gamma' \vdash t'_1\ t'_2 : \phi'$ there exists a type ϕ'' such that $\Gamma, x:\phi, \Gamma' \vdash t_1 : \phi'' \to \phi'$. Again, by Lemma 16 $\psi \equiv \phi'' \to \phi'$. Thus, $ctype_{\phi}(x,t'_1) = \phi'' \to \phi'$ and $\phi'' \to \phi'$ is a subexpression of ϕ. So by the definition of the hereditary substitution function $[t/x]^{\phi}t'_1\ t'_2 = [([t/x]^{\phi}t'_2)/y]^{\phi''}t''_1$, where $[t/x]^{\phi}t'_1 = \lambda y:\phi''.t''_1$. Hence,

$$|rset([t/x]^{\phi}t'_1\ t'_2)| = |rset([([t/x]^{\phi}t'_2)/y]^{\phi''}t''_1)|.$$

Now $\phi >_{\Gamma,\Gamma'} \phi''$ so by the induction hypothesis

$$\begin{array}{lll} |rset([([t/x]^{\phi}t'_2)/y]^{\phi''}t''_1)| & \leq & |rset([t/x]^{\phi}t'_2,t''_1)| \\ & \leq & |rset(t'_2,t''_1,t)| \\ & = & |rset(t'_2,[t/x]^{\phi}t'_1,t)| \\ & \leq & |rset(t'_2,t'_1,t)| \\ & = & |rset(t'_1,t'_2,t)|. \end{array}$$

Suppose $[t/x]^{\phi}t'_1$ is not a λ-abstractions or $ctype_{\phi}(x,t'_1)$ is undefined. Then

$$\begin{array}{lll} rset([t/x]^{\phi}(t'_1\ t'_2)) & = & rset([t/x]^{\phi}t'_1\ [t/x]^{\phi}t'_2) \\ & = & rset([t/x]^{\phi}t'_1,[t/x]^{\phi}t'_2). \\ & \leq & rset(t'_1,t'_2,t). \end{array}$$

Next suppose $t'_1 \equiv \lambda y:\phi_1.t''_1$. Then

$$rset((\lambda y:\phi_1.t''_1)\ t'_2,t) \quad = \quad \{(\lambda y:\phi_1.t''_1)\ t'_2\} \cup rset(t''_1,t'_2,t).$$

By the definition of the hereditary substitution function,

$$\begin{array}{lll} rset([t/x]^{\phi}(\lambda y:\phi_1.t''_1)\ t'_2) & = & rset([t/x]^{\phi}(\lambda y:\phi_1.t''_1)\ [t/x]^{\phi}t'_2) \\ & = & rset((\lambda y:\phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2) \\ & = & \{(\lambda y:\phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2\} \cup rset([t/x]^{\phi}t''_1) \cup rset([t/x]^{\phi}t'_2). \end{array}$$

Since $t' > t''_1$ and $t' > t'_2$ we can apply the induction hypothesis to obtain, $|rset(t''_1,t)| \geq |rset([t/x]^{\phi}t''_1)|$ and $|rset(t'_2,t)| \geq |rset([t/x]^{\phi}t'_2)|$. Therefore,
$|\{(\lambda y:\phi_1.t''_1)\ t'_2\} \cup rset(t''_1,t) \cup rset(t'_2,t)| \geq |\{(\lambda y:\phi_1.[t/x]^{\phi}t''_1)\ [t/x]^{\phi}t'_2\} \cup rset([t/x]^{\phi}t''_1) \cup rset([t/x]^{\phi}t'_2)|$.

B.13 Proof of Normality Preservation

By Lemma 17 we know there exists a term n'' such that $[n/x]^{\phi} n' = n''$ and by Lemma 18 $|rset(n', n)| \geq |rset([n/x]^{\phi} n')|$. Hence, $|rset(n', n)| \geq |rset(n'')|$, but $|rset(n', n)| = 0$. Therefore, $|rset(n'')| = 0$ which implies n'' has no redexes.

B.14 Proof of Soundness with Respect to Reduction

This is a proof by induction on the lexicorgraphic combination (ϕ, t') of $>_{\Gamma,\Gamma'}$ and the strict subexpression ordering. We case split on the structure of t'. When applying the induction hypothesis we must show that the input terms to the substitution and the hereditary substitution functions are typeable. We do not explicitly state typing results that are simple conseqences of inversion.

Case. Suppose t' is a variable x or y distinct from x. Trivial in both cases.

Case. Suppose $t' \equiv \lambda y : \phi'.\hat{t}$. Then $[t/x]^{\phi}(\lambda y : \phi'.\hat{t}) = \lambda y : \phi'.([t/x]^{\phi}\hat{t})$. Now $t' > \hat{t}$ so we can apply the induction hypothesis to obtain $[t/x]\hat{t} \rightsquigarrow^* [t/x]^{\phi}\hat{t}$. At this point we can see that since $\lambda y : \phi'.[t/x]\hat{t} \equiv [t/x](\lambda y : \phi'.\hat{t})$ and we may conclude that $\lambda y : \phi'.[t/x]\hat{t} \rightsquigarrow^* \lambda y : \phi'.[t/x]^{\phi}\hat{t}$.

Case. Suppose $t' \equiv \Lambda X : *_l.\hat{t}$. Similar to the previous case.

Case. Suppose $t' \equiv t'_1\ t'_2$. By Lemma 17 there exists terms $\hat{t}'_1$ and $\hat{t}'_2$ such that $[t/x]^{\phi}t'_1 = \hat{t}'_1$ and $[t/x]^{\phi}t'_2 = \hat{t}'_2$. Since $t' > t'_1$ and $t' > t'_2$ we can apply the induction hypothesis to obtain $[t/x]t'_1 \rightsquigarrow^* \hat{t}'_1$ and $[t/x]t'_2 \rightsquigarrow^* \hat{t}'_2$. Now we case split on whether or not $\hat{t}'_1$ is a λ-abstraction and t'_1 is not, or $ctype_{\phi}(x, t'_1)$ is undefined. If $ctype_{\phi}(x, t'_1)$ is undefined or $\hat{t}'_1$ is not a λ-abstraction then $[t/x]^{\phi}t' = ([t/x]^{\phi}t'_1)\ ([t/x]^{\phi}t'_2) \equiv \hat{t}'_1\ \hat{t}'_2$. Thus, $[t/x]t' \rightsquigarrow^* [t/x]^{\phi}t'$, because $[t/x]t' = ([t/x]t'_1)\ ([t/x]t'_2)$. So suppose $\hat{t}'_1 \equiv \lambda y : \phi'.\hat{t}''_1$ and t'_1 is not a λ-abstraction. By Lemma 16 there exists a type ψ such that $ctype_{\phi}(x, t'_1) = \psi$, $\psi \equiv \phi'' \to \phi'$, and ψ is a subexpression of ϕ, where by inversion on $\Gamma, x : \phi, \Gamma' \vdash t' : \phi'$ there exists a type ϕ'' such that $\Gamma, x : \phi, \Gamma' \vdash t'_1 : \phi'' \to \phi'$. Then by the definiton of the hereditary substitution function $[t/x]^{\phi}(t'_1\ t'_2) = [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$. Now we know $\phi >_{\Gamma,\Gamma'} \phi'$ so we can apply the induction hypothesis to obtain $[\hat{t}'_2/y]\hat{t}''_1 \rightsquigarrow^* [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$. Now by knowing that $(\lambda y : \phi'.\hat{t}''_1)\ t'_2 \rightsquigarrow [\hat{t}'_2/y]\hat{t}''_1$ and by the previous fact we know $(\lambda y : \phi'.\hat{t}''_1)\ t'_2 \rightsquigarrow^* [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$. We now make use of the well known result of full β-reduction. The result is stated as

$$\frac{\begin{array}{l} a \rightsquigarrow^* a' \\ b \rightsquigarrow^* b' \qquad a'\ b' \rightsquigarrow^* c \end{array}}{a\ b \rightsquigarrow^* c}$$

where a, a', b, b', and c are all terms. We apply this result by instantiating a, a', b, b', and c with $[t/x]t'_1$, $\hat{t}'_1$, $[t/x]t'_2$, $\hat{t}'_2$, and $[\hat{t}'_2/y]^{\phi'}\hat{t}''_1$ respectively. Therefore, $[t/x](t'_1\ t'_2) \rightsquigarrow^* [\hat{t}'_2/y]^{\phi'}\hat{t}''_1$.

Case. Suppose $t' \equiv t'_1[\phi'']$. Since $t' > t'_1$ we can apply the induction hypothesis to obtain $[t/x]t'_1 \rightsquigarrow^* [t'/x]^{\phi}t'_1$. We case split on whether or not $[t'/x]^{\phi}t'_1$ is a type abstraction and t'_1 is not. The case where it is not is trivial so we only consider the case where $[t'/x]^{\phi}t'_1 \equiv \Lambda X : *_l.s'$. Then $[t'/x]^{\phi}t' = [\phi'/X]s'$. Now we have $[t/x]t'_1 \rightsquigarrow^* [t'/x]^{\phi}t'_1$ and $[t/x](t'_1[\phi]) \equiv ([t/x]t'_1)[\phi] \rightsquigarrow^* ([t'/x]^{\phi}t'_1)[\phi] \rightsquigarrow [\phi/X]s'$. Thus, $[t/x]t' \rightsquigarrow^* [t'/x]^{\phi}t'$.

B.15 Proof of Lemma 22

This proof is by structural induction on n.

Case. Let $n \equiv x$. By the definition of the interpretation of types $\Gamma(x) = \phi$. Clearly, $(\Gamma, \Gamma')(x) = \phi$, and Lemma 12 gives us $\Gamma, \Gamma' \vdash \phi : *_p$ hence, $x \in [\![\phi]\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv \lambda x : \phi_1.n'$. By the definition of the interpretation of types, there exists a type ϕ_2, such that $\phi = \phi_1 \to \phi_2$, and $n' \in [\![\phi_2]\!]_{\Gamma,x:\phi_1}$. By the induction hypothesis, $n' \in [\![\phi_2]\!]_{\Gamma,\Gamma',x:\phi_1,}$, and by the definition of the interpretation of types $\lambda x : \phi_1.n' \in [\![\phi_1 \to \phi_2]\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv n_1\ n_2$. By the definition of the interpretation of types, there exists a type ϕ_1, such that $n_1 \in [\![\phi_1 \to \phi_2]\!]_\Gamma$, and $n_2 \in [\![\phi_2]\!]_\Gamma$. By the induction hypothesis, $n_1 \in [\![\phi_1 \to \phi_2]\!]_{\Gamma,\Gamma'}$, and $n_2 \in [\![\phi_2]\!]_{\Gamma,\Gamma'}$. Thus, by the definition of the interpretation of types $n_1 n_2 \in [\![\phi_2]\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv \Lambda X : *_p.n'$. By the definition of the interpretation of types, there exists a type ϕ', such that $n' \in [\![\phi']\!]_{\Gamma,X:*_p}$, and by the induction hypothesis $n' \in [\![\phi']\!]_{\Gamma,X:*_p,\Gamma'}$. By the definition of the interpretation of types $\Lambda X : *_p.n' \in [\![\forall X : *_p.\phi']\!]_{\Gamma,\Gamma'}$.

Case. Let $n \equiv n'[\phi']$. By the definition of the interpretation of types, there exists a type ϕ'' and l, such that $\phi = [\phi'/X]\phi''$, $\Gamma \vdash \phi' : *_l$, and $n' \in [\![\forall X : *_l.\phi'']\!]_\Gamma$. By the induction hypothesis $n' \in [\![\forall X : *_l.\phi'']\!]_{\Gamma,\Gamma'}$. We know, $\Gamma \vdash \phi' : *_k$, for some $k \leq l$, so by Lemma 12, $\Gamma, \Gamma' \vdash \phi' : *_k$, hence $\Gamma \vdash \phi' : *_l$. Thus, $n[\phi'] \in [\![[\phi'/X]\phi'']\!]_{\Gamma,\Gamma'}$.

B.16 Proof of Lemma 23

This proof is by structural induction on n.

Case. n is a variable y. Clearly, $[\phi/X]n \equiv [\phi/X]y = y \in [\![\phi']\!]_{\Gamma,X:*_l,\Gamma'}$, and
$(\Gamma, [\phi/X]\Gamma')(y) = [\phi/X]\phi'$. Also, we have $\Gamma, [\phi/X]\Gamma' \vdash [\phi/X]\phi' : *_p$ for some p, by Lemma 10. Hence, by the definition of the interpretation of types, $y \in [\![[\phi/X]\phi']\!]_{\Gamma,[\phi/X]\Gamma'}$.

Case. Let $n \equiv \lambda y : \psi.n'$. By the definition of the interpretation of types $\phi' \equiv \psi \to \psi'$. By the induction hypothesis $[\phi/X]n' \in [\![[\phi/X]\psi']\!]_{\Gamma,\Gamma',y:[\phi/X]\psi}$. Again by the definition of the interpretation of types $\lambda y : [\phi/X]\psi.[\phi/X]n' \equiv [\phi/X](\lambda y : \psi.n') \in [\![[\phi/X]\phi']\!]_{\Gamma,[\phi/X]\Gamma'}$.

Case. Let $n \equiv n_1\ n_2$. By the definition of the interpretation of types $\phi' \equiv \psi$, $n_1 \in [\![\psi' \to \psi]\!]_{\Gamma,X:*_q,\Gamma'}$, and $n_2 \in [\![\psi']\!]_{\Gamma,X:*_q,\Gamma'}$. By the induction hypothesis $[\phi/X]n_1 \in [\![[\phi/X](\psi' \to \psi)]\!]_{\Gamma,[\phi/X]\Gamma'}$ and $[\phi/X]n_2 \in [\![[\phi/X]\psi']\!]_{\Gamma,[\phi/X]\Gamma'}$. Now by the definition of the interpretation of types $([\phi/X]n_1)([\phi/X]n_2) \in [\![[\phi/X]\psi]\!]_{\Gamma,[\phi/X]\Gamma'}$, since $[\phi/X]n_1$, cannot be a λ-abstraction.

Case. Let $n \equiv \Lambda Y : *_q.n'$. By the definition of the interpretation of types $\phi' = \forall Y : *_q.\psi$ and $n' \in [\![\psi]\!]_{\Gamma,X:*_l,\Gamma',Y:*_q}$. By the induction hypothesis
$[\phi/X]n' \in [\![[\phi/X]\psi]\!]_{\Gamma,[\phi/X]\Gamma',Y:*_q}$ and by the definition of the interpretation of types $\Lambda Y : *_q.[\phi/X]n' \in [\![\forall Y : *_q.[\phi/X]\psi]\!]_{\Gamma,[\phi/X]\Gamma'}$ which is equivalent to $[\phi/X](\Lambda Y : *_q.n') \in [\![[\phi/X](\forall Y : *_q.\psi)]\!]_{\Gamma,[\phi/X]\Gamma'}$.

Case. Let $n \equiv n'[\psi]$. By the definition of the interpretation of types $\phi' = [\psi/Y]\psi'$, for some Y, ψ, and there exists a q such that $\Gamma, X : *_l, \Gamma' \vdash \psi : *_q$, and $n' \in [\![\forall Y : *_q.\psi']\!]_{\Gamma,X:*_l,\Gamma'}$. By the induction hypothesis $[\phi/X]n' \in [\![[\phi/X](\forall Y : *_q.\psi')]\!]_{\Gamma,[\phi/X]\Gamma'}$. Therefore, by the definition of the interpretation of types
$([\phi/X]n')[\psi] \in [\![[\psi/Y]([\phi/X]\psi')]\!]_{\Gamma,[\phi/X]\Gamma'}$, which is equivalent to
$[\phi/X](n'[\psi]) \in [\![[\phi/X]([\psi/Y]\psi')]\!]_{\Gamma,[\phi/X]\Gamma'}$.

B.17 Proof of Type Soundness

This is a proof by induction on the structure of the typing derivation of t.

Case.

$$\frac{\Gamma(x) = \phi \qquad \Gamma\ Ok}{\Gamma \vdash x : \phi}$$

By regularity $\Gamma \vdash \phi : *_l$ for some l, hence $[\![\phi]\!]_\Gamma$ is nonempty. Clearly, $x \in [\![\phi]\!]_\Gamma$ by the definition of the interpretation of types.

Case.

$$\frac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x : \phi_1.t : \phi_1 \to \phi_2}$$

By the induction hypothesis $t \in [\![\phi_2]\!]_{\Gamma,x:\phi_1}$ and by the definition of the interpretation of types $t \rightsquigarrow^! n \in [\![\phi_2]\!]_{\Gamma,x:\phi_1}$ and $\Gamma, x : \phi_1 \vdash n : \phi_2$. Thus, by applying the λ-abstraction type-checking rule, $\Gamma \vdash \lambda x : \phi_1.n : \Pi x : \phi_1.\phi_2$ so by the definition of the interpretation of types $\lambda x : \phi_1.n \in [\![\phi_1 \to \phi_2]\!]_\Gamma$. Thus, according to the definition of the interpretation of types $\lambda x : \phi_1.t \rightsquigarrow^! \lambda x : \phi_1.n \in [\![\phi_1 \to \phi_2]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash t_1 : \phi_2 \to \phi_1 \qquad \Gamma \vdash t_2 : \phi_2}{\Gamma \vdash t_1\ t_2 : \phi_1}$$

By the induction hypothesis $t_1 \rightsquigarrow^! n_1 \in [\![\phi_2 \to \phi_1]\!]_\Gamma$, $t_2 \rightsquigarrow^! n_2 \in [\![\phi_2]\!]_\Gamma$, $\Gamma \vdash \phi_2 \to \phi_1 : *_p$, and $\Gamma \vdash \phi_2 : *_q$. Inversion on the arrow-type kind-checking rule yields, $\Gamma \vdash \phi_1 : *_r$, and by Lemma 12, $\Gamma, x : \phi_2, \Gamma' \vdash \phi_1 : *_r$.

Now we know from above that $n_1 \in [\![\phi_2 \to \phi_1]\!]_\Gamma$ and $n_2 \in [\![\phi_2]\!]_\Gamma$, hence $\Gamma \vdash n_1 : \phi_2 \to \phi_1$ and $\Gamma \vdash n_2 : \phi_2$. It suffices to show that $n_1\ n_2 \in [\![\phi_2]\!]_\Gamma$. Clearly, $n_1\ n_2 = [n_1/z](z\ n_2)$ for some variable $z \notin FV(n_1, n_2)$. Lemma 17, Lemma 20, and Lemma 19 allow us to conclude that $[n_1/z](z\ n_2) \rightsquigarrow^* [n_1/z]^{\phi_2 \to \phi_1}(z\ n_2)$, $\Gamma \vdash [n_1/z]^{\phi_2 \to \phi_1}(z\ n_2) : \phi_2$, and $[n_1/z]^{\phi_2 \to \phi_1}(z\ n_2)$ is normal. Thus, $t_1\ t_2 \rightsquigarrow^* n_1\ n_2 = [n_1/z](z\ n_2) \rightsquigarrow^! [n_1/z]^{\phi_2 \to \phi_1}(z\ n_2) \in [\![\phi_2]\!]_\Gamma$.

Case.

$$\frac{\Gamma, X : *_p \vdash t : \phi}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.\phi}$$

By the induction hypothesis and definition of the interpretation of types $t \in [\![\phi]\!]_{\Gamma,X:*_p}$, $t \rightsquigarrow^! n \in [\![\phi]\!]_{\Gamma,X:*_p}$ and $\Lambda X : *_p.n \in [\![\phi]\!]_\Gamma$. Again, by definition of the interpretation of types $\Lambda X : *_p.t \rightsquigarrow^! \Lambda X : *_p.n \in [\![\phi]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash t : \forall X : *_l.\phi_1 \qquad \Gamma \vdash \phi_2 : *_l}{\Gamma \vdash t[\phi_2] : [\phi_2/X]\phi_1}$$

By the induction hypothesis $t \in [\![\forall X : *_l.\phi_1]\!]_\Gamma$ and by the definition of the interpretation of types we know $t \rightsquigarrow^! n \in [\![\forall X : *_l.\phi_1]\!]_\Gamma$. We case split on whether or not n is a type abstraction. If not then again, by the definition of the interpretation of types $n[\phi_2] \in [\![[\phi_2/X]\phi_1]\!]_\Gamma$, therefore $t \in [\![[\phi_2/X]\phi_1]\!]_\Gamma$. Suppose $n \equiv \Lambda X : *_l.n'$. Then $t[\phi_2] \rightsquigarrow^* (\Lambda X : *_l.n')[\phi_2] \rightsquigarrow [\phi_2/X]n'$. By the definition of the interpretation of types $n' \in [\![\phi_1]\!]_{\Gamma,X:*_l}$. Therefore, by Lemma 23 $[\phi_2/X]n' \in [\![[\phi_2/X]\phi_1]\!]_\Gamma$.

C Proofs Pretaining to SSF$^\omega$

C.1 Type Level Properties

We restate all the properties here with their proofs to make the proofs easier to understand. However, due to space constraints we did not include the statements in the main body of the paper.

Lemma 28 (Properties of $ctype_\phi$)**.**

i. If $ckind_K(X, \phi) = K'$ then $head(\phi) = X$ and K' is a subexpression of K.

ii. If $\Gamma, X : K, \Gamma' \vdash \phi : K'$ and $ckind_K(X, \phi) = K''$ then $K' \equiv K''$.

iii. If $\Gamma, X : K, \Gamma' \vdash \phi_1\ \phi_2 : K'$, $\Gamma \vdash \phi : K$, $[\phi/X]^K \phi_1 = \lambda Y : K_1.\psi$, and ϕ_1 is not then there exists a kind ψ' such that $ckind_K(X, \phi_1) = \psi'$.

Proof. We prove part one first. This is a proof by induction on the structure of ϕ.

Case. Suppose $\phi \equiv X$. Then $ckind_K(X, X) = K$. Clearly, $head(X) = X$ and K is a subexpression of itself.

Case. Suppose $\phi \equiv \phi_1\ \phi_2$. Then $ckind_K(X, \phi_1\ \phi_2) = K''$ when $ckind_K(X, \phi_1) = K' \to K''$. Now $\phi > \phi_1$ so by the induciton hypothesis $head(\phi_1) = X$ and $K' \to K''$ is a subexpression of K. Therefore, $head(\phi_1\ \phi_2) = X$ and certainly K'' is a subexpression of K.

We now prove part two. This is also a proof by induction on the structure of ϕ.

Case. Suppose $\phi \equiv X$. Then $ckind_K(X, X) = K$. Clearly, $K \equiv K$.

Case. Suppose $\phi \equiv \phi_1\ \phi_2$. Then $ckind_K(X, \phi_1\ \phi_2) = K_2$ when $ckind_K(X, \phi_1) = K_1 \to K_2$. By inversion on the assumed kinding derivation we know there exists a kind K'' such that $\Gamma, X : K, \Gamma' \vdash \phi_1 : K'' \to K'$. Now $\phi > \phi_1$ so by the induciton hypothesis $K_1 \to K_2 \equiv K'' \to K'$. Therefore, $K_1 \equiv K''$ and $K_2 \equiv K'$.

Next we prove part three. This is a proof by induction on the structure of $\phi_1\ \phi_2$.

The only possiblities for the form of ϕ_1 is X, $\hat{\phi}_1\ \hat{\phi}_2$. All other forms would not result in $\{\phi/X\}^K \phi_1$ being a λ-abstraction and ϕ_1 not. If $\phi_1 \equiv X$ then there exist a kind K'' such that $K \equiv K'' \to K'$ and $ckind_K(X, X\ \phi_2) = K'$ when $ckind_K(X, X) = K \equiv K'' \to K'$ in this case. We know K'' to exist by inversion on $\Gamma, X : K, \Gamma' \vdash \phi_1\ \phi_2 : K'$.

Now suppose $\phi_1 \equiv (\hat{\phi}_1\ \hat{\phi}_2)$. Now knowing ϕ_1' to not be a λ-abstraction implies that $\hat{\phi}_1$ is also not a λ-abstraction or $\{\phi/X\}^K \phi_1$ would be an application instead of a λ-abstraction. So it must be the case that $\{\phi/X\}^K \hat{\phi}_1$ is a λ-abstraction and $\hat{\phi}_1$ is not. Since $\phi_1\ \phi_2 > \phi_1$ we can apply the induction hypothesis to obtain there exists a kind K''' such that $ckind_K(X, \hat{\phi}_1) = K'''$. Now by inversion on $\Gamma, X : K, \Gamma' \vdash \phi_1\ \phi_2 : K'$ we know there exists a kind K'' such that $\Gamma, X : K, \Gamma' \vdash \phi_1 : K'' \to K'$. We know $\phi_1 \equiv (\hat{\phi}_1\ \hat{\phi}_2)$ so by inversion on $\Gamma, X : K, \Gamma' \vdash \phi_1 : K'' \to K'$ we know there exists a kind $\hat{K}$ such that $\Gamma, X : K, \Gamma' \vdash \hat{\phi}_1 : \hat{K} \to (K'' \to K')$. By part two of Lemma 28 we know $K''' \equiv \hat{K} \to (K'' \to K')$ and $ckind_K(X, \phi_1) = ckind_K(X, \hat{\phi}_1\ \hat{\phi}_2) = K'' \to K'$ when $ckind_K(X, \hat{\phi}_1) = \hat{K} \to (K'' \to K')$, because we know $ckind_K(X, \hat{\phi}_1) = K'''$.

□

Lemma 29 (Total and Type Preserving)**.** *Suppose $\Gamma \vdash \phi : K$ and $\Gamma, X : K, \Gamma' \vdash \phi' : K'$. Then there exists a type ϕ'' such that $\{\phi/X\}^K \phi' = \phi''$ and $\Gamma, \Gamma' \vdash \phi'' : K'$.*

Proof. This is a proof by induction on the lexicorgraphic combination (K, ϕ') of the strict subexpression ordering. We case split on ϕ'.

Case. Suppose ϕ' is either X or a variable Y distinct from X. Trivial in both cases.

Case. Suppose $\phi' \equiv \phi_1' \to \phi_2'$. Trivial.

Case. Suppse $\phi' \equiv \forall X : *_l.\phi''$. Trivial.

Case. Suppose $\phi' \equiv \lambda Y : K_1.\phi'_1$. By inversion on the kinding judgement we know $\Gamma, X : K, \Gamma', Y : K_1 \vdash \phi'_1 : K_2$. We also know $\phi' > \phi'_1$, hence we can apply the induction hypothesis to obtain $\{\phi/X\}^K\phi'_1 = \hat{\phi}'_1$ and $\Gamma, \Gamma', Y : K_1 \vdash \hat{\phi}'_1 : K_2$ for some type $\hat{\phi}'_1$. By the definition of the hereditary substitution function $\{\phi/X\}^K\phi' = \lambda Y : K_1.\{\phi/X\}^K\phi'_1 = \lambda Y : K_1.\hat{\phi}'_1$. It suffices to show that $\Gamma, \Gamma' \vdash \lambda Y : K_1.\hat{\phi}'_1 : K_1 \to K_2$. By simply applying the λ-abstraction kinding rule using $\Gamma, \Gamma', Y : K_1 \vdash \hat{\phi} : K_2$ we obtain $\Gamma, \Gamma' \vdash \lambda Y : K_1.\hat{\phi}'_1 : K_1 \to K_2$.

Case. Suppose $\phi' \equiv \phi'_1\ \phi'_2$. By inversion we know $\Gamma, X : K, \Gamma' \vdash \phi'_1 : K'' \to K'$ and $\Gamma, X : K, \Gamma' \vdash \phi'_2 : K''$ for some types K' and K''. Clearly, $\phi' > \phi'_i$ for $i \in \{1, 2\}$. Thus, by the induction hypothesis there exists types ψ_1 and ψ_2 such that $\{\phi/X\}^K\phi'_i = \psi_i$, $\Gamma, \Gamma' \vdash \psi_1 : K'' \to K'$ and $\Gamma, \Gamma' \vdash \psi_2 : K''$ for $i \in \{1, 2\}$. We case split on whether or not ψ_1 is a λ-abstraction, and ϕ'_1 is not, or $ckind_K(X, \phi'_1)$ is undefined. We only consider the non-trivial case when $\psi_1 \equiv \lambda Y : K''.\psi'_1$ and ϕ'_1 is not a λ-abstraction.

Now by Lemma 28 it is the case that there exists a $\hat{K}$ such that $ckind_K(X, \phi'_1) = \hat{K}$, $\hat{K} \equiv K'' \to K'$, and $\hat{K}$ is a subexpression of K, hence $K >_{\Gamma,\Gamma'} K''$. Then $\{\phi/X\}^K(\phi'_1\ \phi'_2) = \{\psi_2/Y\}^{K''}\psi'_1$. Therefore, by the induction hypothesis there exists a term ψ such that $\{\psi_2/Y\}^{K''}\psi'_1 = \psi$ and $\Gamma, \Gamma' \vdash \psi : K''$.

$\square$

Lemma 30 (Normality Preserving)**.** *If $\Gamma \vdash \phi : K$ and $\Gamma, X : K \vdash \phi' : \phi'$, where ϕ and ϕ' are normal, then there exists a normal type ϕ'' such that $\{\phi/X\}^K\phi' = \phi''$.*

Proof. By Lemma 29 we know there exists a type ϕ'' such that $\{\phi/X\}^K\phi' = \phi''$ and by Lemma 32 $|rset(\phi', \phi)| \geq |rset(\{\phi/X\}^K\phi')|$. Hence, $|rset(\phi', \phi)| \geq |rset(\phi'')|$, but $|rset(\phi', \phi)| = 0$. Therefore, $|rset(\phi'')| = 0$ which implies ϕ'' is normal. $\square$

Lemma 31 (Soundness with Respect to Reduction)**.** *If $\Gamma \vdash \phi : K$ and $\Gamma, X : K, \Gamma' \vdash \phi' : K'$ then $\{\phi/X\}\phi' \rightsquigarrow^* \{\phi/X\}^K\phi'$.*

Proof. This is a proof by induction on the lexicorgraphic combination (K, ϕ') of the strict subexpression ordering. We case split on the structure of ϕ'. When applying the induction hypothesis we must show that the input terms to the substitution and the hereditary substitution function are kindable. We do not explicitly state kinding results that are simple conseqences of inversion.

Case. Suppose ϕ' is a variable X or Y distinct from X. Trivial in both cases.

Case. Suppose $\phi' \equiv \lambda Y : K'.\hat{\phi}$. Then $\{\phi/X\}^K(\lambda Y : K'.\hat{\phi}) = \lambda Y : K'.(\{\phi/X\}^K\hat{\phi})$. Now $\phi' > \hat{\phi}$ so we can apply the induction hypothesis to obtain $\{\phi/X\}\hat{\phi} \rightsquigarrow^* \{\phi/X\}^K\hat{\phi}$. At this point we can see that since $\lambda Y : K'.\{\phi/X\}\hat{\phi} \equiv \{\phi/X\}(\lambda Y : K'.\hat{\phi})$ and we may conclude that $\lambda Y : K'.\{\phi/X\}\hat{\phi} \rightsquigarrow^* \lambda Y : K'.\{\phi/X\}^K\hat{\phi}$.

Case. Suppose $\phi' \equiv \phi'_1 \to \phi'_2$. Trivial.

Case. Suppse $\phi' \equiv \forall X : *_l.\phi''$. Trivial.

Case. Suppose $\phi' \equiv \phi'_1\ \phi'_2$. By Lemma 29 there exists terms $\hat{\phi}'_1$ and $\hat{\phi}'_2$ such that $\{\phi/X\}^K\phi'_1 = \hat{\phi}'_1$ and $\{\phi/X\}^K\phi'_2 = \hat{\phi}'_2$. Since $\phi' > \phi'_1$ and $\phi' > \phi'_2$ we can apply the induction hypothesis to obtain $\{\phi/X\}\phi'_1 \rightsquigarrow^* \hat{\phi}'_1$ and $\{\phi/X\}\phi'_2 \rightsquigarrow^* \hat{\phi}'_2$. Now we case split on whether or not $\hat{\phi}'_1$ is a λ-abstraction and ϕ'_1 is not or $ckind_K(X, \phi'_1)$ is undefined. If $ckind_K(X, \phi'_1)$ is undefined or $\hat{\phi}'_1$ is not a λ-abstraction then $\{\phi/X\}^K\phi' = (\{\phi/X\}^K\phi'_1)\ (\{\phi/X\}^\phi\phi'_2) \equiv \hat{\phi}'_1\ \hat{\phi}'_2$. Thus, $\{\phi/X\}\phi' \rightsquigarrow^* \{\phi/X\}^K\phi'$, because $\{\phi/X\}\phi' = (\{\phi/X\}\phi'_1)\ (\{\phi/X\}\phi'_2)$. So suppose $\hat{\phi}'_1 \equiv \lambda Y : K'.\hat{\phi}''_1$ and ϕ'_1 is not a λ-abstraction. By Lemma 28 there exists a kind $\hat{K}$ such that $ckind_K(X, \phi'_1) = \hat{K}$, $\hat{K} \equiv K'' \to K'$, and $\hat{K}$ is a subexpression of K, where by inversion on $\Gamma, X : K, \Gamma' \vdash \phi' : K'$ there exists a kind K'' such that $\Gamma, X : K, \Gamma' \vdash \phi'_1 : K'' \to K'$. Then by the definiton of the hereditary substitution function $\{\phi/X\}^K(\phi'_1\ \phi'_2) = \{\hat{\phi}'_2/Y\}^{K'}\hat{\phi}''_1$. Now we know $K > K'$ so we can apply the induction hypothesis to obtain $\{\hat{\phi}'_2/Y\}\hat{\phi}''_1 \rightsquigarrow^* \{\hat{\phi}'_2/Y\}^{K'}\hat{\phi}''_1$. Now by knowing that $(\lambda Y : K'.\hat{\phi}''_1)\ \phi'_2 \rightsquigarrow \{\hat{\phi}'_2/Y\}\hat{\phi}''_1$ and by the previous fact we know $(\lambda Y : K'.\hat{\phi}''_1)\ \phi'_2 \rightsquigarrow^* \{\hat{\phi}'_2/Y\}^{K'}\hat{\phi}''_1$. We now make use of the well known result of full β-reduction. The result is stated as

$$\frac{\begin{array}{l} a \leadsto^* a' \\ b \leadsto^* b' \qquad a'\, b' \leadsto^* c \end{array}}{a\, b \leadsto^* c}$$

where a, a', b, b', and c are all terms. We apply this result by instantiating a, a', b, b', and c with $[\phi/X]\phi'_1$, $\hat{\phi}'_1$, $[\phi/X]\phi'_2$, $\hat{\phi}'_2$, and $\{\hat{\phi}'_2/Y\}^{K'}\hat{\phi}''_1$ respectively. Therefore, $[\phi/X](\phi'_1\ \phi'_2) \leadsto^* \{\hat{\phi}'_2/Y\}^{K'}\hat{\phi}''_1$.

□

We redefine the $rset$ function for types as follows:

$$rset(X) = \emptyset$$

$$rset(\lambda X : K.\phi) = rset(\phi)$$

$$rset(\phi_1 \to \phi_2) = rset(\phi_1, \phi_2)$$

$$rset(\forall X : K.\phi) = rset(\phi)$$

$$\begin{array}{lll} rset(\phi_1\ \phi_2) & & \\ = & rset(\phi_1, \phi_2) & \text{if } \phi_1 \text{ is not a } \lambda\text{-abstraction.} \\ = & \{\phi_1\ \phi_2\} \cup rset(\phi'_1, \phi_2) & \text{if } \phi_1 \equiv \lambda X : K.\phi'_1. \end{array}$$

Lemma 32 (Redex Preserving)**.** *If* $\Gamma \vdash \phi : K$, $\Gamma, X : K, \Gamma' \vdash \phi' : K'$ *then* $|rset(\phi', \phi)| \geq |rset(\{\phi/X\}^K \phi')|$.

Proof. This is a proof by induction on the lexicorgraphic combination (K, ϕ') of the strict subexpression ordering. We case split on the structure of ϕ'.

Case. Let $\phi' \equiv X$ or $\phi' \equiv Y$ where Y is distinct from X. Trivial.

Case. Suppose $\phi' \equiv \phi'_1 \to \phi'_2$. Trivial.

Case. Suppse $\phi' \equiv \forall X : *_l.\phi''$. Trivial.

Case. Let $\phi' \equiv \lambda X : K_1.\phi''$. Then $\{\phi/X\}^K\phi' \equiv \lambda X : K_1.\{\phi/X\}^K\phi''$. Now

$$\begin{array}{rcl} rset(\lambda X : K_1.\phi'', \phi) & = & rset(\lambda X : K_1.\phi'') \cup rset(\phi) \\ & = & rset(\phi'') \cup rset(\phi) \\ & = & rset(\phi'', \phi). \end{array}$$

We know that $\phi' > \phi''$ by the strict subexpression ordering, hence by the induction hypothesis $|rset(\phi'', \phi)| \geq |rset(\{\phi/X\}^K\phi'')|$ which implies $|rset(\phi', \phi)| \geq |rset(\{\phi/X\}^K\phi')|$.

Case. Let $\phi' \equiv \phi'_1\ \phi'_2$. First consider when ϕ'_1 is not a λ-abstraction. Then

$$rset(\phi'_1\ \phi'_2, \phi) = rset(\phi'_1, \phi'_2, \phi)$$

Clearly, $\phi' > \phi'_i$ for $i \in \{1, 2\}$, hence, by the induction hypothesis $|rset(\phi'_i, \phi)| \geq |rset(\{\phi/X\}^K\phi'_i)|$. We have two cases to consider. That is whether or not $\{\phi/X\}^K\phi'_1$ is a λ-abstraction or $ckind_K(X, \phi'_1)$ is undefined. Suppose the former. Then by Lemma 28 $ctype_\phi(X, \phi'_1) = \hat{K}$ for some kind $\hat{K}$ and by inversion on $\Gamma, X : K, \Gamma' \vdash \phi'_1\ \phi'_2 : K'$ there exists a kind K'' such that $\Gamma, X : K, \Gamma' \vdash \phi_1 : K'' \to K'$. Again, by Lemma 28 $\hat{K} \equiv K'' \to K'$. Thus, $ckind_K(X, \phi'_1) = K'' \to K'$ and $K'' \to K'$ is a subexpression of K. So by the definition of the hereditary substitution function $\{\phi/X\}^K\phi'_1\ \phi'_2 = \{(\{\phi/X\}^K\phi'_2)/Y\}^{K''}\phi''_1$, where $\{\phi/X\}^K\phi'_1 = \lambda Y : K''.\phi''_1$. Hence,

$$|rset(\{\phi/X\}^K\phi'_1\ \phi'_2)| = |rset(\{(\{\phi/X\}^K\phi'_2)/Y\}^{K''}\phi''_1)|.$$

Now $K > K''$ so by the induction hypothesis

$$\begin{array}{rcl}
|rset(\{(\{\phi/X\}^K\phi_2')/Y\}^{K''}\phi_1'')| & \leq & |rset(\{\phi/X\}^K\phi_2', \phi_1'')| \\
& \leq & |rset(\phi_2', \phi_1'', \phi)| \\
& = & |rset(\phi_2', \{\phi/X\}^K\phi_1', \phi)| \\
& \leq & |rset(\phi_2', \phi_1', \phi)| \\
& = & |rset(\phi_1', \phi_2', \phi)|.
\end{array}$$

Suppose $\{\phi/X\}^K\phi_1'$ is not a λ-abstractions or $ckind_K(X, \phi_1')$ is undefined. Then

$$\begin{array}{rcl}
rset(\{\phi/X\}^K(\phi_1'\ \phi_2')) & = & rset(\{\phi/X\}^K\phi_1'\ \{\phi/X\}^K\phi_2') \\
& = & rset(\{\phi/X\}^K\phi_1', \{\phi/X\}^K\phi_2'). \\
& \leq & rset(\phi_1', \phi_2', \phi).
\end{array}$$

Next suppose $\phi_1' \equiv \lambda Y : K_1.\phi_1''$. Then

$$rset((\lambda Y : K_1.\phi_1'')\ \phi_2', \phi) \quad = \quad \{(\lambda Y : K_1.\phi_1'')\ \phi_2'\} \cup rset(\phi_1'', \phi_2', \phi).$$

By the definition of the hereditary substitution function,

$$\begin{array}{rcl}
rset(\{\phi/X\}^K(\lambda Y : K_1.\phi_1'')\ \phi_2') & = & rset(\{\phi/X\}^K(\lambda Y : K_1.\phi_1'')\ \{\phi/X\}^K\phi_2') \\
& = & rset((\lambda Y : K_1.\{\phi/X\}^K\phi_1'')\ \{\phi/X\}^K\phi_2') \\
& = & \{(\lambda Y : K_1.\{\phi/X\}^K\phi_1'')\ \{\phi/X\}^K\phi_2'\} \cup rset(\{\phi/X\}^K\phi_1'') \cup rset(\{\phi/X\}^K\phi_2').
\end{array}$$

Since $\phi' > \phi_1''$ and $\phi' > \phi_2'$ we can apply the induction hypothesis to obtain, $|rset(\phi_1'', \phi)| \geq |rset(\{\phi/X\}^K\phi_1'')|$ and $|rset(\phi_2', \phi)| \geq |rset(\{\phi/X\}^K\phi_2')|$. Therefore,
$|\{(\lambda Y : K_1.\phi_1'')\ \phi_2'\} \cup rset(\phi_1'', \phi) \cup rset(\phi_2', \phi)| \geq |\{(\lambda Y : K_1.\{\phi/X\}^K\phi_1'')\ \{\phi/X\}^K\phi_2'\} \cup rset(\{\phi/X\}^K\phi_1'') \cup rset(\{\phi/X\}^K\phi_2')|$.

□

Lemma 33 (Substitution for the Interpretation of Types)**.** *If* $\phi' \in [\![K']\!]_{\Gamma,X:K,\Gamma'}$, $\phi \in [\![K]\!]_\Gamma$, *then* $\{\phi/X\}^K\phi' \in [\![K']\!]_{\Gamma,\Gamma'}$.

Proof. By Lemma 29 we know there exists a type $\hat{\phi}$ such that $\{\phi/X\}^K\phi' = \hat{\phi}$ and $\Gamma, \Gamma' \vdash \hat{\phi} : K'$ and by Lemma 30 $\hat{\phi}$ is normal. Therefore, $\{\phi/X\}^K\phi' = \hat{\phi} \in [\![K']\!]_{\Gamma,\Gamma'}$. □

Theorem 7 (Type Soundness)**.** *If* $\Gamma \vdash \phi : K$ *then* $\phi \in [\![K]\!]_\Gamma$.

Proof. This is a proof by induction on the structure of the typing derivation of ϕ.

Case.

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\ Ok}{\Gamma \vdash X : *_p}$$

Clearly, $X \in [\![K]\!]_\Gamma$ by the definition of the interpretation of types.

Case.

$$\frac{\Gamma \vdash \phi_1 : *_p \qquad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{max(p,q)}}$$

Trivial.

Case.

$$\frac{\Gamma, X : *_q \vdash \phi : *_p}{\Gamma \vdash \forall X : *_q.\phi : *_{max(p,q)+1}}$$

Trivial.

Case.

$$\frac{\Gamma, X : K_1 \vdash \phi : K_2}{\Gamma \vdash \lambda X : K_1.\phi : K_1 \to K_2}$$

By the induction hypothesis $\phi \in [\![K_2]\!]_{\Gamma,X:K_1}$ and by the definition of the interpretation of types $\phi \rightsquigarrow^! \psi \in [\![\phi_2]\!]_{\Gamma,X:K_1}$ and $\Gamma, X : K_1 \vdash \psi : K_2$. Thus, by applying the λ-abstraction kind-checking rule, $\Gamma \vdash \lambda X : K_1.\psi : K_1 \to K_2$ so by the definition of the interpretation of types $\lambda X : K_1.\psi \in [\![K_1 \to K_2]\!]_\Gamma$. Thus, according to the definition of the interpretation of types $\lambda X : K_1.\phi \rightsquigarrow^! \lambda X : K_1.\psi \in [\![K_1 \to K_2]\!]_\Gamma$.

Case.

$$\frac{\Gamma \vdash \phi_1 : K_2 \to K_1 \qquad \Gamma \vdash \phi_2 : K_2}{\Gamma \vdash \phi_1\ \phi_2 : K_1}$$

By the induction hypothesis $\phi_1 \rightsquigarrow^! \psi_1 \in [\![K_2 \to K_1]\!]_\Gamma$, and $\phi_2 \rightsquigarrow^! \psi_2 \in [\![K_2]\!]_\Gamma$.

Now we know from above that $\psi_1 \in [\![K_2 \to K_1]\!]_\Gamma$ and $\psi_2 \in [\![K_2]\!]_\Gamma$, hence $\Gamma \vdash \psi_1 : K_2 \to K_1$ and $\Gamma \vdash \psi_2 : K_2$. It suffices to show that $\psi_1\ \psi_2 \in [\![K_2]\!]_\Gamma$. Clearly, $\psi_1\ \psi_2 = [\psi_1/Z](Z\ \psi_2)$ for some variable $Z \notin FV(\psi_1, \psi_2)$. Lemma 29, Lemma 31, and Lemma 30 allow us to conclude that $[\psi_1/Z](Z\ \psi_2) \rightsquigarrow^* \{\psi_1/Z\}^{K_2 \to K_1}(Z\ \psi_2)$, $\Gamma \vdash \{\psi_1/Z\}^{\phi_2 \to \phi_1}(Z\ \psi_2) : K_2$, and $\{\psi_1/Z\}^{K_2 \to K_1}(Z\ \psi_2)$ is normal. Thus, $\phi_1\ \phi_2 \rightsquigarrow^* \psi_1\ \psi_2 = [\psi_1/Z](Z\ \psi_2) \rightsquigarrow^! \{\psi_1/Z\}^{K_2 \to K_1}(Z\ \psi_2) \in [\![K_2]\!]_\Gamma$.

□

Corollary 5 (Normalization)**.** *If $\Gamma \vdash \phi : K$ then $\phi \rightsquigarrow^! \psi$ for some normal type ψ.*

C.2 Term Level Properties

C.2.1 Proof of Preservation of Types

We prove part one and two by mutual induction starting with part 1.

Case.

$$\frac{}{\cdot\ Ok}$$

Trivial.

Case.

$$\frac{(\Gamma, x : \phi, \Gamma')\ Ok}{(\Gamma, x : \phi, \Gamma', X : *_p)\ Ok}$$

A simple application of the inductive hypothesis yeilds our desired result.

Case.

$$\frac{\Gamma, x : \phi, \Gamma' \vdash \psi : *_p \qquad (\Gamma, x : \phi, \Gamma')\ Ok}{(\Gamma, x : \phi, \Gamma', y : \psi)\ Ok}$$

By weakening for kinding, $\Gamma, \Gamma' \vdash \psi : *_p$ and then by strengthening for kinding, $\Gamma, x : \phi', \Gamma' \vdash \psi : *_p$. The inductive hypothesis on $(\Gamma, x : \phi, \Gamma')\ Ok$ yields $(\Gamma, x : \phi', \Gamma')\ Ok$, thus by applying the above rule $(\Gamma, x : \phi', \Gamma', y : \psi)\ Ok$.

Next we prove part two.

Case.

$$\frac{\Gamma \vdash \phi_1 : *_p \qquad \Gamma \vdash \phi_2 : *_q}{\Gamma \vdash \phi_1 \to \phi_2 : *_{max(p,q)}}$$

By two applications of the inductive hypothesis $\Gamma \vdash \phi_2' : *_q$ and $\Gamma \vdash \phi_1' : *_p$, where $\phi_1 \rightsquigarrow \phi_1'$ and $\phi_2 \rightsquigarrow \phi_2'$, because $\phi \equiv (\phi_1 \to \phi_2) \rightsquigarrow \phi' \equiv (\phi_1' \to \phi_2')$. Now by applying the above rule $\Gamma \vdash \phi_1' \to \phi_2' : *_{max(p,q)}$.

Case.

$$\frac{\Gamma, X : *_q \vdash \phi : *_p}{\Gamma \vdash \forall X : *_q.\phi : *_{max(p,q)+1}}$$

We know by assumption that $\phi \equiv (\forall X : *_q.\phi) \rightsquigarrow \phi' \equiv (\forall X : *_q.\phi')$. Hence, $\phi \rightsquigarrow \phi'$ so by the inductive hypothesis $\Gamma, X : *_q \vdash \phi' : *_q$. Finally, by applying the above rule $\Gamma \vdash \forall X : *_q.\phi' : *_{max(p,q)+1}$.

Case.

$$\frac{\Gamma(X) = *_p \qquad p \leq q \qquad \Gamma\ Ok}{\Gamma \vdash X : *_q}$$

Trivial, because X is a normal form.

C.2.2 Proof of Lemma 27

This is a proof by induction on the assumed typing derivation.

Case.

$$\frac{\Gamma(x) = \phi \qquad \Gamma\ Ok}{\Gamma \vdash x : \phi}$$

Suppose there exists a type ϕ' such that $\phi \rightsquigarrow \phi'$. It suffcies to show that $\Gamma \vdash x : \phi'$. We know it must be the case that $\Gamma \equiv (\Gamma'', x : \phi, \Gamma''')$ and by Lemma 26 $(\Gamma'', x : \phi', \Gamma''')\ Ok$. Now clearly, $\Gamma'', x : \phi', \Gamma''' \vdash x : \phi'$. Take, $\Gamma'', x : \phi', \Gamma'''$ for Γ'.

Case.

$$\frac{\Gamma, x : \phi_1 \vdash t : \phi_2}{\Gamma \vdash \lambda x : \phi_1.t : \phi_1 \to \phi_2}$$

First, by assumption $\phi \equiv \phi_1 \to \phi_2$ and $\phi_1 \to \phi_2 \rightsquigarrow \phi'$. It must be the case that $\phi' \rightsquigarrow \phi'_1 \to \phi'_2$ some some types ϕ_1 and ϕ_2. By the inductive hyothesis there exists a Γ' such that $\Gamma' \vdash t : \phi'_2$. Lemma 8 yeilds $(\Gamma, x : \phi_1)\ Ok$ which implies that $\Gamma, x : \phi \vdash \phi_1 : K$ for some kind K. Now by Lemma 26 $\Gamma, x : \phi_1 \vdash \phi'_1 : K$. We obtain $\Gamma \vdash \phi'_1$ by strengthing for kinding and by weaking for kinding $\Gamma', x : \phi'_1 \vdash t : \phi'_2$. Thus, by applying the λ-abstraction typing rule $\Gamma' \vdash \lambda x : \phi'_1.t : \phi'_1 \to \phi'_2$.

Case.

$$\frac{\Gamma \vdash t_1 : \phi_1 \to \phi_2 \qquad \Gamma \vdash t_2 : \phi_1}{\Gamma \vdash t_1\ t_2 : \phi_2}$$

We know by assumption that $\phi \equiv \phi_2 \rightsquigarrow \phi'$ which implies that $\phi_1 \to \phi_2 \rightsquigarrow \phi_1 \to \phi'_2$. By the inductive hypothesis there exists a $\Gamma' \vdash t_1 : \phi_1 \to \phi_2$. Let $\Gamma'' \subseteq \Gamma$ contain only the free variables necessary to type t_2. So $\Gamma \equiv \Gamma_1, \Gamma'', \Gamma_2$ and we know $\Gamma_1, \Gamma'', \Gamma_2 \vdash t_2 : \phi_1$. Clearly, if Γ'' contains only the variables needed to type t_2 then $\Gamma'' \vdash t_2 : \phi_1$, because it must be the case that Γ_1 and Γ_2 are not needed for the typing of t_2. Now by weakening for typing $\Gamma'', \Gamma' \vdash t_2 : \phi_2$, which is equivalent to $\Gamma', \Gamma'' \vdash t_2 : \phi_2$. Again by weakening for typing, $\Gamma', \Gamma'' \vdash t_1 : \phi_1 \to \phi'_2$. Thus, by applying the above typing rule $\Gamma', \Gamma'' \vdash t_1\ t_2 : \phi'_2$.

Case.

$$\frac{\Gamma, X : *_p \vdash t : \phi}{\Gamma \vdash \Lambda X : *_p.t : \forall X : *_p.\phi}$$

Similar to the λ-abstraction case.

Case.

$$\frac{\Gamma \vdash t : \forall X : *_l.\phi_1 \qquad \Gamma \vdash \phi_2 : *_l}{\Gamma \vdash t[\phi_2] : [\phi_2/X]\phi_1}$$

Similar to the application case.