substitution would have the following result:

$$[t/y]t' = (\lambda x : X.x)z.$$

However, hereditary substitution has the following result:

$$[t/y]^{X \to X} t' = z,$$

because hereditary substitution first capture avoidingly substitutes t for y in t' and examines the result. It then sees that a new redex $(\lambda x : T.x)z$ has been created. Then it recursively reduces this redex as follows: $[z/x]^X x$.

Hereditary substitution is important for a number of reasons. It was first used as a means to conduct the metatheory of the type theory LF which is the core of the proof assistant Twelf. LF is based on canonical forms. That is, the language itself does not allow any non-normal forms to be defined. That is $(\lambda x : T.t)\, t'$ is not a valid term in LF. Thus, their operational semantics cannot use ordinary capture avoiding substitution, because as we saw in the above example, we can substitute normal forms into a normal form and end up with a non-normal form. So Watkins used hereditary substitution instead of capture avoiding substitution in their operational semantics [142]. Adams extended this work to dependent types in his thesis [6]. We will show how to use hereditary substitution to show weak normalization. Let's consider how to define hereditary substitution for STLC.

6.2 Hereditary Substitution for STLC

The definition of the hereditary substitution depends on a partial function called *ctype*. This function is equivalent to the *treduce* function used in [142]. It is defined by the following definition.

Definition 6.2.0.1.

The partial function is defined with respect to a fixed type T and has two arguments, a free variable x, and a term t, where x may be free in t. We define ctype by induction on the form of t.

$$ctype_T(x, x) = T$$

$$ctype_T(x, t_1\ t_2) = T''$$

$$\text{Where } ctype_T(x, t_1) = T' \to T''.$$

The *ctype* function simply computes the type of a term in weak-head normal form. The following lemma states two very important properties of *ctype*. We do not include any proofs here, but they can be found in [50].

Lemma 6.2.0.2.

i. If $ctype_T(x, t) = T'$ then $head(t) = x$ and T' is a subexpression of T.

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

The purpose of *ctype* is to detect when a new redex will be created in the definition of the hereditary substitution function. We define the hereditary substitution function next.

Definition 6.2.0.3.

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 T.

$$[t/x]^T x = t$$

$[t/x]^T y = y$

Where y is a variable distinct from x.

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

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

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

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

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

We can see that every case of the previous definition except the application cases are identical to the definition of capture-avoiding substitution. 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.

One 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. This was first done by Harley Eades and Aaron Stump in [49] 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 of application that prevents totality from being trivial. Now if this case was used we know that $ctype_T(x, t_1) = T'' \to T'$, and by Lemma 6.2.0.2, $T'' \to T'$ is a subexpression of T. This implies that T'' is a strict subexpression on T. 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 (T, t) of the strict subexpression ordering. This shows that *ctype* reveals information about the types of the input terms to the hereditary substitution function, which allows us to use the well-founded ordering to prove properties of the hereditary substitution function.

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 *ctype* brings this ordering into the definition of the hereditary substitution function.

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* is defined. So by Lemma 6.2.0.2 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 hereditarily substituting $[t/x]^T t_2'$ for for y of type T'' 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.

In STLC 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 one contracted by 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.

The hereditary substitution function has several properties. First it is a total and type preserving function.

Lemma 6.2.0.4. *Suppose* $\Gamma \vdash t : T$ *and* $\Gamma, x : T, \Gamma' \vdash t' : T'$. *Then there exists a term* t'' *such that* $[t/x]^T t' = t''$ *and* $\Gamma, \Gamma' \vdash t'' : T'$.

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 6.2.0.5. *If $\Gamma \vdash n : T$ and $\Gamma, x : T \vdash n' : T'$ then there exists a normal term n'' such that $[n/x]^T n' = n''$.*

The final property is soundness with respect to reduction.

Lemma 6.2.0.6. *If $\Gamma \vdash t : T$ and $\Gamma, x : T, \Gamma' \vdash t' : T'$ then $[t/x]t' \rightsquigarrow^* [t/x]^T t'$.*

Soundness with respect to reduction shows that the hereditary substitution function does nothing more than what we can do with the operational semantics and ordinary capture avoiding substitution. All of these properties should hold for any hereditary substitution function, not just for STLC. They are correctness properties that must hold in order to use the hereditary substitution function to show normalization.

We can now prove normalization of STLC using the hereditary substitution function. We first define a semantics for the types of STLC.

Definition 6.2.0.7.

First we define when a normal form is a member of the interpretation of type T in context Γ

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

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

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

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

The interpretation of types was inspired by the work of Prawitz in [110] although we use open terms here where he used closed terms. Next we show that the definition of the interpretation of types is closed under hereditary substitutions.

Lemma 6.2.0.8. *If $n' \in [\![T']\!]_{\Gamma,x:T,\Gamma'}$, $n \in [\![T]\!]_{\Gamma}$, then $[n/x]^T n' \in [\![T']\!]_{\Gamma,\Gamma'}$.*

Proof. By Lemma 6.2.0.4 we know there exists a term $\hat{n}$ such that $[n/x]^T n' = \hat{n}$ and $\Gamma, \Gamma' \vdash \hat{n} : T'$ and by Lemma 6.2.0.5 $\hat{n}$ is normal. Therefore, $[n/x]^T n' = \hat{n} \in [\![T']\!]_{\Gamma,\Gamma'}$. $\square$

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

Theorem 6.2.0.9. *If $\Gamma \vdash t : T$ then $t \in [\![T]\!]_{\Gamma}$.*

Proof. This holds by a straightforward proof by induction on the structure of the assumed typing derivation where the application case uses the previous lemma. $\square$

Corollary 6.2.0.10. *If $\Gamma \vdash t : T$, then there exists a normal form n, such that $t \rightsquigarrow^! n$.*

This proof method has been applied to a number of different type theories. Eades and Stump show that SSF is weakly normalizing using this proof technique in [49]. The advantage of hereditary substitution is that it shows promise of being less complex than other normalization techniques. For example, when proving normalization using the Tait-Girard reducibility method – discussed in the next section – the main soundness theorem must universally quantify over the domain of well-formed substitutions, but this is not needed in the proof using hereditary substitution. In addition, the interpretation of types do not require advanced mathematical machinerary like the reducibility method where they require quantification over types as well as recursion over types (large eliminations). This implies that it would be easier to formalize in hereditary substitution proofs in proof assistants. However, it is currently unknown which type theories can be proven normalizing using hereditary substitution.

The type theories currently known to be proven normalizing using the hereditary substitution proof technique are relatively simple. For example, the simply-typed λ-calculus, predicative polymorphic versions of system F, and the LF family of dependent types have all been shown to be weakly-normalizing using hereditary substitution. However, it is currently unknown how to prove normalization of system T using hereditary substitution, because the usual ordering on types is not proof theoretically strong enough. The termination ordinal of the ordering on types would have to be at least ϵ_0, but such an ordering is non-trivial to construct. Furthermore, impredicative theories like system F pose a similar problem. It is currently unknown how to construct the ordering for second order types. The hereditary substitution technique heavily depends on the ordering on types and the lack of one prevent its use. As the type theory becomes more expressive through extensions the harder the ordering on types becomes to construct.

Hereditary substitution can be used to maintain canonical forms and even prove weak normalization of predicative simple type theories. It can also be used as a normalization function. A normalization function is a function that when given a term it returns the normal form of the input term. Andreas Abel and Dulma Rodriguez used hereditary substitution in this manner in [4]. They used it to normalize types in a type theory with type-level computation much like system F^{ω}. In that paper the authors were investigating subtyping in the presence of type level computation. They found that hereditary substitution could be used to normalize types and then do subtyping. This allowed them to only define subtyping on normal types. Similar to their work Chantal Keller and Thorsten Altenkirch use hereditary substitution to define a normalizer for STLC and formalize their work in Agda [72]. As we mentioned above the drawback of hereditary substitution is that it does not scale to richer type theories. Thus, to prove consistency of more advanced type theories we need another technique that does scale.

6.3 Tait-Girard Reducibility

The Tait-Girard reducibility method is a technique for showing weak and strong normalization of type theories. It originated from the work of William Tait. He showed strong normalization of system Tusing an interpretation of types based on set theory with comprehension. He called this interpretation saturated sets. Later, John Yves Girard, against popular belief[1], extended Tait's method to be able to prove system F strongly normalizing. He called his method reducibility candidates. The reducibility candidates method is based on second order set theory with comprehension. It turns out that the genius work of Girard extends to a large class of type theories. The standard reference on all the topics of this section is Girard's wonderful book [59]. We will summarize how to show strong normalization of STLC using Tait's

[1]It has been said that while Girard was working on extending Tait's method other researchers, notably Stephen Kleene, criticized him for trying. They thought it was an impossible endeavor.