Alpha Conversion in Simply Typed Lambda Calculus

ANA BOVE, *Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, E-mail:* bove@cs.chalmers.se

PAULA SEVERI, *Centro de Matemática, Fac. de Ciencias, Universidad de la República, E-mail:* severi@cmat.edu.uy

Abstract

In the usual presentations of simply typed λ-calculus, it is usual to identify α-convertible terms. However, this is not at all a practise in most (typed) functional languages, for which simply typed λ-calculus is a theoretical foundation. Here, five well known variants of the type system for simply λ-calculus which work with variable names are presented. Essentially, these formulations differ in the way variable declarations are handled and all of them are equivalent if α-convertible terms are identified. However, if α-convertible terms are not identified, some of the systems turn out to type less terms than the others. The main aim of this paper is to relate these systems by comparing their set of typable terms and study the property of closure under α-conversion for each system.

1 Introduction

Consider presentations of λ-calculus as in [1, 2] or any extension of it. In these kind of presentations, it is usual to identify terms that are α-convertible and then, work with λ-terms in a naive way.

In the presentation of simply typed λ-calculus given in [2], we can type the term $(\lambda u.\lambda x.\lambda y.u\ y\ x)\ (\lambda z.\lambda x.z)$. However, if we β-reduce it, we obtain $\lambda x.\lambda y.(\lambda z.\lambda x.z)\ y\ x$ which is not typable. The problem here is that to type the latter term, we have to type the subterm $\lambda z.\lambda x.z$ in a context where the variable x is already declared. As this is not possible in the type system presented in [2], we cannot derive a type for the term. Since α-convertible terms are identified in this presentation of λ-calculus, we can think that β-reducing the first term gives $\lambda x.\lambda y.(\lambda z.\lambda w.z)\ y\ x$ as a result, which is typable in the system.

However, identifying terms that are α-convertible is not at all a practise in most (typed) functional languages, for which simply typed λ-calculus is a theoretical foundation. Then, `\x -> \x -> x` and `fn x => fn x => x` are legal expressions of type `a -> b -> b` in Haskell and ML respectively. Hence, the type systems behind functional languages can formally deal with terms that are α-convertible.

A natural way to deal with terms that are α-convertible is to add an extra rule to the type system that expresses that α-convertible terms have the same type, similar to the one presented in [9]. Then, if $=_\alpha$ denotes α-conversion and $\vdash$ represents the type derivability judgement, given Γ a context, M and M' terms and τ a type, the extra rule would look like:

$$\frac{\Gamma \vdash M : \tau \qquad M =_\alpha M'}{\Gamma \vdash M' : \tau}$$

The disadvantage of adding this extra rule is that now, when reasoning about type derivations, we should not only consider the structure of the term but also the rules that define α-conversion.

Another solution is to work with indices instead of with variable names, as suggested in [8]. Then, both the terms $\lambda x.\lambda x.x$ and $\lambda x.\lambda y.y$ would be written in a way similar to $\lambda.\lambda.0$. Note that this solution naturally identifies terms that are α-convertible. However, already with relatively small terms, the indices-versions of the terms become unreadable for human eyes.

Then, the main aim of this work is to study type systems for simply typed λ-calculus that work with variable names and that are closed under α-conversion. Besides, we would like to have clear and simple induction principles associated to the type systems that make proofs easy to perform and understand.

We present here five variants of the type system for simply typed λ-calculus. Essentially, these formulations differ in the way they handle variable declarations. All of these formulations are equivalent if we identify terms that are α-convertible, that is, they have the same set of typable terms up to α-conversion. However, if we distinguish terms that differ in their bound variables, they are not longer equivalent.

Although these systems appear often in theoretical research about λ-calculus or functional languages, it is not common to see a formal study of their properties with respect to α-conversion. In order to close this gap, we present here a formal definition of α-conversion and we prove which of the systems we present are closed under α-conversion and which are not. In addition, we show how the systems are related by comparing the set of typable terms in each system.

The organisation of this paper is as follows:

In section 2, we introduce formal definitions of one step α-reduction and α-conversion.

In section 3, we present five variants of the type system for simply typed λ-calculus, namely *Us*, *Wk*, *Th*, *Lt* and *St*.

In section 4, we prove some properties of *Lt*, namely the strengthening, weakening, thinning and substitution lemmas. We conclude this section with the proof that *Lt* is closed under α-conversion.

In section 5, we relate the type systems introduced in section 3 by comparing the set of typable terms. Here, we prove that *Us* is included in *Wk* which is, in turn, included in *Lt*, and we show that neither *Us* nor *Wk* are closed under α-conversion. In addition, we prove that the systems *Lt*, *St* and *Th* are equivalent. As a result of this equivalence, we have that *Th* and *St* are also closed under α-conversion.

In section 6, we briefly discuss the problem of α-conversion for simply typed λ-calculus extended with polymorphic definitions and fix point operators, and λ-calculus with dependent types.

2 Alpha Conversion

In this section, we introduce a formal definition of α-conversion. In order to do this, we first define the notions of substitution and one step α-reduction.

We assume the usual definition for λ-calculus and the convention for parentheses as in [1]. Then, $\lambda x.M\ N$ denotes $\lambda x.(M\ N)$. Given M a λ-term and S a set, we assume the following notations: $\mathsf{fv}(M)$ and $\mathsf{bv}(M)$ denote the sets of free and bound variables of M respectively, $\mathsf{vars}(M)$ the set of all variables of M, $\#\lambda(M)$ denotes the number of λ's that occurs in M and $\#(S)$ the cardinality of S.

Now, we introduce the notion of *substitution of N for x in M*, denoted by $M\,[N/x]$.

Definition 1. (Substitution of N for x in M).

$$
\begin{array}{lcll}
y\,[N/x] & \stackrel{def}{=} & \begin{cases} y & \text{if } y \neq x \\ N & \text{if } y = x \end{cases} \\[2ex]
(\lambda y.P)\,[N/x] & \stackrel{def}{=} & \begin{cases} \lambda y.(P\,[N/x]) & \text{if } y \neq x \\ \lambda y.P & \text{if } y = x \end{cases} \\[2ex]
(P\ Q)\,[N/x] & \stackrel{def}{=} & P\,[N/x]\ \ Q\,[N/x]
\end{array}
$$

Note that when we substitute N for x in $\lambda y.M$ with $y \neq x$ and y free in N, we do not perform any rename of bound variable as in [6]. However, whenever substitution is used in this paper we explicitly add those requirements needed to avoid binding free variables.

Now, we define the notion of α-reduction. In one step of α-reduction, we perform the renaming of one bound variable, that is, the variable x in an abstraction of the form $\lambda x.M$ is changed for a variable y not occurring in M.

Definition 2. (α-reduction).

$$
\mathsf{Ren}_\alpha\ \ \frac{y \notin \mathsf{vars}(M)}{\lambda x.M \to_\alpha \lambda y.M\,[y/x]} \qquad \mathsf{Abs}_\alpha\ \ \frac{M \to_\alpha M'}{\lambda x.M \to_\alpha \lambda x.M'}
$$

$$
\mathsf{App\_L}_\alpha\ \ \frac{M \to_\alpha M'}{M\ N \to_\alpha M'\ N} \qquad \mathsf{App\_R}_\alpha\ \ \frac{N \to_\alpha N'}{M\ N \to_\alpha M\ N'}
$$

We denote *α-conversion* by $=_\alpha$ and we define it as the reflexive, symmetric and transitive closure of $\to_\alpha$.

With the next lemma, we show that any term is α-convertible to a term whose bound variables are all different from each other and also different from the free ones.

Lemma 3. Let S be a set of variables such that $\mathsf{vars}(M) \cap S = \emptyset$ and $\#(S) = \#\lambda(M)$. Then, there exists a term M' such that $M =_\alpha M'$ and $\mathsf{bv}(M') = S$.

The proof is by induction on the structure of M.

3 Variants for Simply Typed Lambda Calculus

In this section, we present five variants of the type system for simply typed λ-calculus. In order to present them, we first introduce the notions of types and contexts.

We consider the usual set of types as presented in [2]. We assume that σ, τ (possibly primed or subscripted) range over types and $\sigma \to \tau$ represents a function type.

A context is a (possibly empty) sequence of declarations of the form $x\!:\!\sigma$ and we assume that Γ, Δ (possibly primed or subscripted) range over contexts. We denote the empty context as $[\,]$, by $\Gamma, x\!:\!\sigma$ we mean a context where the declaration $x\!:\!\sigma$ has been added to the declarations in Γ, and Γ, Δ denotes the concatenation of the contexts Γ and Δ. By Γ valid we mean a context Γ in which each variable is declared at most once, $\mathsf{dom}(\Gamma)$ denotes the domain of Γ, that is, the set of variables declared in

$$\mathsf{var}_{\mathsf{Us}} \qquad \frac{\Gamma\ \mathsf{valid} \qquad x{:}\sigma \in \Gamma}{\Gamma \vdash_{Us} x : \sigma}$$

$$\mathsf{abs}_{\mathsf{Us}} \qquad \frac{\Gamma, x{:}\sigma \vdash_{Us} M : \tau}{\Gamma \vdash_{Us} \lambda x.M : \sigma \to \tau}$$

$$\mathsf{app}_{\mathsf{Us}} \qquad \frac{\Gamma \vdash_{Us} M : \sigma \to \tau \qquad \Gamma \vdash_{Us} N : \sigma}{\Gamma \vdash_{Us} M\ N : \tau}$$

$$\mathsf{var}_{\mathsf{Wk}} \qquad \frac{\Gamma, x{:}\sigma\ \mathsf{valid}}{\Gamma, x{:}\sigma \vdash_{Wk} x : \sigma}$$

$$\mathsf{weak}_{\mathsf{Wk}} \qquad \frac{\Gamma \vdash_{Wk} M : \tau \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x{:}\sigma \vdash_{Wk} M : \tau}$$

$$\mathsf{abs}_{\mathsf{Wk}} \qquad \frac{\Gamma, x{:}\sigma \vdash_{Wk} M : \tau}{\Gamma \vdash_{Wk} \lambda x.M : \sigma \to \tau}$$

$$\mathsf{app}_{\mathsf{Wk}} \qquad \frac{\Gamma \vdash_{Wk} M : \sigma \to \tau \qquad \Gamma \vdash_{Wk} N : \sigma}{\Gamma \vdash_{Wk} M\ N : \tau}$$

$$\mathsf{var}_{\mathsf{Th}} \qquad \frac{\Gamma\ \mathsf{valid} \qquad x{:}\sigma \in \Gamma}{\Gamma \vdash_{Th} x : \sigma}$$

$$\mathsf{thinn}_{\mathsf{Th}} \qquad \frac{\Gamma \vdash_{Th} M : \tau \qquad \Gamma \leqslant_{Th} \Delta \qquad \Delta\ \mathsf{valid}}{\Delta \vdash_{Th} M : \tau}$$

$$\mathsf{abs}_{\mathsf{Th}} \qquad \frac{\Gamma, x{:}\sigma \vdash_{Th} M : \tau}{\Gamma \vdash_{Th} \lambda x.M : \sigma \to \tau}$$

$$\mathsf{app}_{\mathsf{Th}} \qquad \frac{\Gamma \vdash_{Th} M : \sigma \to \tau \qquad \Gamma \vdash_{Th} N : \sigma}{\Gamma \vdash_{Th} M\ N : \tau}$$

$$\mathsf{var}_{\mathsf{Lt}} \qquad \frac{x \notin \mathsf{dom}(\Delta)}{\Gamma, x{:}\sigma, \Delta \vdash_{Lt} x : \sigma}$$

$$\mathsf{abs}_{\mathsf{Lt}} \qquad \frac{\Gamma, x{:}\sigma \vdash_{Lt} M : \tau}{\Gamma \vdash_{Lt} \lambda x.M : \sigma \to \tau}$$

$$\mathsf{app}_{\mathsf{Lt}} \qquad \frac{\Gamma \vdash_{Lt} M : \sigma \to \tau \qquad \Gamma \vdash_{Lt} N : \sigma}{\Gamma \vdash_{Lt} M\ N : \tau}$$

$$\mathsf{var}_{\mathsf{St}} \qquad \frac{\Gamma\ \mathsf{valid} \qquad x{:}\sigma \in \Gamma}{\Gamma \vdash_{St} x : \sigma}$$

$$\mathsf{abs}_{\mathsf{St}} \qquad \frac{\Gamma_x, x{:}\sigma \vdash_{St} M : \tau}{\Gamma \vdash_{St} \lambda x.M : \sigma \to \tau}$$

$$\mathsf{app}_{\mathsf{St}} \qquad \frac{\Gamma \vdash_{St} M : \sigma \to \tau \qquad \Gamma \vdash_{St} N : \sigma}{\Gamma \vdash_{St} M\ N : \tau}$$

FIG. 1. Five Type Systems for Simply Typed λ-Calculus

Γ, and Γ_x denotes Γ where all the declarations for the variable x have been removed. The relation $x{:}\sigma \in \Gamma$ is satisfied if $x{:}\sigma$ is one of the declarations in Γ.

In figure 1 we present five well-known variants of the type system for simply typed λ-calculus. Essentially, they differ in the way they handle variable declarations. In section 5, we relate these systems by comparing the set of typable terms.

Let $ts \in \{Us, Wk, Th, Lt, St\}$. The relation $\Gamma \vdash_{ts} M : \tau$ means that we can derive that the term M has type τ under the context Γ in the type system ts. If Γ is empty, we might write $\vdash_{ts} M : \tau$ instead of $[\,] \vdash_{ts} M : \tau$. For each type system ts, $\Gamma \leqslant_{ts} \Delta$ means that the context Δ extends Γ in the sense that for any variable x, $\Gamma \vdash_{ts} x : \sigma$ implies $\Delta \vdash_{ts} x : \sigma$. If ts is a type system where the context is required to be valid, then Γ valid and Δ valid should also be satisfied as part of the definition of $\Gamma \leqslant_{ts} \Delta$. Note that if $\Gamma \leqslant_{ts} \Delta$, the order of the declarations in Δ might differ from the order of the declarations in Γ.

The system Us is the usual presentation of the simply typed λ-calculus à la Curry as in [2]. This system is not closed under α-conversion.

Using the weakening rule for the system Wk, we can derive a type for certain terms where some variables are bound more than once, such as $\lambda x.\lambda x.x$. Although one might believe that this system is closed under α-conversion, we show with a counterexample in section 5 that it is not. This system is usually given in its general formulation for pure type systems (see [3, 15, 2]) and it has also been used as the type system for a small typed functional language in [4] and [14].

The thinning rule for system *Th* is often seen as a generalisation of the weakening rule for *Wk*, where we have the possibility of adding several declarations at the same time instead of one at a time. However, due to the definition of $\leqslant_{ls}$, the thinning rule also allows us to change the order of the declarations in the context. It is in fact this possibility that makes the system *Th* closed under α-conversion.

The system *Lt* is the only one that allows multiple declarations for the same variable. In *Lt*, the contexts are considered as lists, and in order to give the type of a variable x we look up the variable in the context from right to left until we find a declaration for x. This, in turn, is the last declaration added to the context for the variable x. This makes the system closed under α-conversion. Although one seldom finds this system in theoretical research, this system is often used when implementing a type system for a functional language. The difference between *Lt* and the system LT in [12] is that Pollack gives two rules for typing variables while we give only one. However, the meaning of his two rules is exactly the meaning of the rule $\mathsf{var}_{\mathsf{Lt}}$ in our system.

Finally, in the system *St* we consider the contexts as sets. Here, in order to derive $\Gamma \vdash_{St} \lambda x.M : \sigma \to \tau$, we take away all declarations for the variable x from the context and we add a declaration of the form $x{:}\sigma$. Note that if $x \notin \mathsf{dom}(\Gamma)$ then $\Gamma = \Gamma_x$. On the other hand, if x is declared in Γ, and thus $x \in \mathsf{dom}(\Gamma)$, we can see the abstraction rule for the system *St* as having an implicit thinning rule. The power of this abstraction rule makes this system closed under α-conversion. Extensions of this system for types-schemes for functional languages and qualified types can be found in [7] and [10] respectively.

4 Properties of *Lt*

In this section, we prove some properties of *Lt* concluding with the property that shows that *Lt* is closed under α-conversion.

When typing a term, we only use the last declarations of its free variables. In particular, in the derivation of $\Gamma, x{:}\sigma, \Delta \vdash_{Lt} M : \tau$, the declaration $x{:}\sigma$ is used only if x occurs free in M and if this is the last declaration for x in the context. Otherwise, we can think that the declaration $x{:}\sigma$ is *unnecessary*.

Definition 4. (Unnecessary Declaration). We say that the declaration $x{:}\sigma$ is unnecessary for M in Δ, if $x \in \mathsf{fv}(M)$ implies $x \in \mathsf{dom}(\Delta)$.

Below, we show the strengthening lemma that says that we can remove those declarations that are unnecessary for typing a term from the context. For example, if we have a derivation of $x{:}\sigma, x{:}\tau \vdash_{Lt} x : \tau$, then the first declaration of x is unnecessary and can be removed from the context.

Lemma 5. (Strengthening). Assume that $x{:}\sigma$ is unnecessary for M in Δ. Then, $\Gamma, x{:}\sigma, \Delta \vdash_{Lt} M : \tau$ implies $\Gamma, \Delta \vdash_{Lt} M : \tau$.

Proof: The proof is by induction on the structure of the term M. We show here the case where M is an abstraction of the form $\lambda x.N$. All other cases are straightforward.

When M is $\lambda x.N$, as $\Gamma, x{:}\sigma, \Delta \vdash_{Lt} \lambda x.N : \tau_1 \to \tau_2$ then $\Gamma, x{:}\sigma, \Delta, x{:}\tau_1 \vdash_{Lt} N : \tau_2$. Thus, $x{:}\sigma$ is unnecessary for N in $\Delta, x{:}\tau_1$. By the induction hypothesis, we have that

$\Gamma, \Delta, x:\tau_1 \vdash_{\mathit{Lt}} N : \tau_2$. Then, by applying the abstraction rule we obtain the desired result, that is, $\Gamma, \Delta \vdash_{\mathit{Lt}} \lambda x.N : \tau_1 \rightarrow \tau_2$. ∎

As a consequence of the previous lemma, we have that if a term M is typable under a context Γ, then M is also typable under a context Γ' that only contains the last declaration for each of the free variables in M.

Corollary 6. If $\Gamma \vdash_{\mathit{Lt}} M : \tau$ then $\Gamma' \vdash_{\mathit{Lt}} M : \tau$, where Γ' is obtained from Γ by removing all unnecessary declarations.

The next lemma we present is the converse of the strengthening lemma. It says that we can add a declaration to a context, if the new declaration does not interfere with typing a term. Note that a declaration $x:\sigma$ does not interfere with the derivation of $\Gamma, \Delta \vdash_{\mathit{Lt}} M : \tau$, when $x:\sigma$ is unnecessary for M in Δ.

Lemma 7. (Weakening). Let $x:\sigma$ be unnecessary for M in Δ. If $\Gamma, \Delta \vdash_{\mathit{Lt}} M : \tau$ then $\Gamma, x:\sigma, \Delta \vdash_{\mathit{Lt}} M : \tau$.

The proof of the lemma is by a straightforward induction on the structure of M.

Although the next lemma might seem a generalisation of the previous one, it is not. If $x \notin \mathsf{fv}(M) \cup \mathsf{dom}(\Delta)$ and $x:\tau$ is the last declaration for x in Γ with $\tau \neq \sigma$, then we have that $\Gamma \leqslant_{\mathit{Lt}} \Gamma, x:\sigma, \Delta$ does not hold. Hence, there is a case where the weakening lemma applies but the thinning lemma does not.

Lemma 8. (Thinning). Let $\Gamma \leqslant_{\mathit{Lt}} \Delta$. If $\Gamma \vdash_{\mathit{Lt}} M : \tau$ then $\Delta \vdash_{\mathit{Lt}} M : \tau$.

The lemma is proved by induction on the structure of the term M.

Now we introduce the substitution lemma for Lt. We need the hypothesis $x \notin \mathsf{dom}(\Delta)$ to ensure that $x:\sigma$ is the last declaration for the variable x in the context. The hypothesis $\mathsf{fv}(N) \cap (\mathsf{bv}(M) \cup \mathsf{dom}(\Delta)) = \emptyset$ is necessary to avoid binding free variables of N when substituting.

Lemma 9. (Substitution). Let $x \notin \mathsf{dom}(\Delta)$ and $\mathsf{fv}(N) \cap (\mathsf{bv}(M) \cup \mathsf{dom}(\Delta)) = \emptyset$. If $\Gamma, x:\sigma, \Delta \vdash_{\mathit{Lt}} M : \tau$ and $\Gamma \vdash_{\mathit{Lt}} N : \sigma$, then $\Gamma, \Delta \vdash_{\mathit{Lt}} M\,[N/x] : \tau$.

Proof: The proof is by induction on the structure of M. We show here the case where M is the variable x. All other cases are straightforward.

If $M = x$, since $x \notin \mathsf{dom}(\Delta)$ then $\sigma = \tau$ and $\Gamma \vdash_{\mathit{Lt}} N : \tau$. As $\mathsf{fv}(N) \cap \mathsf{dom}(\Delta) = \emptyset$, all declarations in Δ are unnecessary for typing N. Then, we can apply the weakening lemma several times to enlarge the context and obtain $\Gamma, \Delta \vdash_{\mathit{Lt}} N : \tau$. ∎

Note that the converse of the substitution lemma does not hold. If $I = \lambda x.x$, then the term $I\ I$ is typable in Lt but the term $x\ x$ is not, no matter which type we assume for x. However, the former is the result of substituting I for x in the latter.

We need the following proposition to prove that Lt is closed under α-reduction and α-expansion.

Proposition 10. Let $x \notin \mathsf{dom}(\Delta)$ and $y \notin \mathsf{vars}(M) \cup \mathsf{dom}(\Delta)$. Then, we have $\Gamma, x:\sigma, \Delta \vdash_{\mathit{Lt}} M : \tau$ if and only if $\Gamma, y:\sigma, \Delta \vdash_{\mathit{Lt}} M\,[y/x] : \tau$.

Proof: ($\Rightarrow$). Assume that $\Gamma, x:\sigma, \Delta \vdash_{\mathit{Lt}} M : \tau$. Since $y \notin \mathsf{vars}(M)$, $y:\sigma$ is unnecessary for M in any context. Then, by the weakening lemma, we have $\Gamma, y:\sigma, x:\sigma, \Delta \vdash_{\mathit{Lt}} M : \tau$ and by the substitution lemma, we obtain that $\Gamma, y:\sigma, \Delta \vdash_{\mathit{Lt}} M\,[y/x] : \tau$.

($\Leftarrow$). The proof is made by induction on the structure of M. Note that even though $M\,[y/x]\,[x/y] \;=\; M$, we cannot use ($\Rightarrow$) to prove ($\Leftarrow$) since the variable x might occur bound in M. ∎

Theorem 11. (Closure under α-reduction and α-expansion). If $M \rightarrow_\alpha M'$, then $\Gamma \vdash_{Lt} M : \tau$ if and only if $\Gamma \vdash_{Lt} M' : \tau$.

Proof: ($\Rightarrow$) The proof is by induction on the derivation of $M \rightarrow_\alpha M'$. We show here the case for the renaming rule. All other cases are straightforward.

In the case where the derivation is obtained by applying the renaming rule, we have that M is of the form $\lambda x.N$, M' of the form $\lambda y.N\,[y/x]$ and τ of the form $\tau_1 \rightarrow \tau_2$. Then, we have that $\Gamma \vdash_{Lt} \lambda x.N : \tau_1 \rightarrow \tau_2$, and thus $\Gamma, x{:}\tau_1 \vdash_{Lt} N : \tau_2$. Since $y \notin \mathsf{vars}(N)$, by applying the proposition 10 with $\Delta = [\,]$, we obtain that $\Gamma, y{:}\tau_1 \vdash_{Lt} N\,[y/x] : \tau_2$, and hence $\Gamma \vdash_{Lt} \lambda y.N\,[y/x] : \tau_1 \rightarrow \tau_2$ as desired.

($\Leftarrow$) The proof is by induction on the derivation of $M \rightarrow_\alpha M'$ and it is symmetric to the previous one. ∎

As a consequence of the this theorem we have that *Lt* is closed under α-conversion.

Corollary 12. (Closure under α-conversion for *Lt*). If $M =_\alpha M'$ and $\Gamma \vdash_{Lt} M : \tau$, then $\Gamma \vdash_{Lt} M' : \tau$.

5 Relation between the Different Type Systems

In this section, we relate the type systems introduced in section 3 by comparing the set of typable terms.

We say that two systems *ts* and *ts'* are *equivalent*, denoted by $ts = ts'$, if for every term M, whenever we can derive that M has type τ in *ts*, we can also derive that M has type τ in *ts'* and vice versa. We say that system *ts* is *included* in system *ts'*, denoted by $ts \subset ts'$, if for every term M, whenever we can derive that M has type τ in *ts*, we can also derive that M has type τ in *ts'*, but the two systems are not equivalent.

Lemma 13. If $\Gamma \vdash_{Us} M : \tau$ then $\Gamma \vdash_{Wk} M : \tau$.

The proof is by induction on the structure of M. The converse of this lemma does not hold since for example, $\lambda x.\lambda x.x$ is typable in *Wk* but not in *Us*. Hence, $Us \subset Wk$.

Lemma 14. If $\Gamma \vdash_{Wk} M : \tau$ then $\Gamma \vdash_{Lt} M : \tau$.

The proof is by induction on the derivation of $\Gamma \vdash_{Wk} M : \tau$.

Now, we show that *Wk* and *Lt* are not equivalent. Following the structure of the term we can easily derive that $\vdash_{Lt} \lambda x.\lambda y.(\lambda x.x\ y)\ x : (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau$. However, as it is shown below $\not\vdash_{Wk} \lambda x.\lambda y.(\lambda x.x\ y)\ x : (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau$. Then, $Wt \subset Lt$.

To show that $\vdash_{Wk} \lambda x.\lambda y.(\lambda x.x\ y)\ x : (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau$ is not derivable in *Wk*, we give a bottom-up argument. To begin with, as the context is empty, the only possibility we have is to apply the abstraction rule. Afterwards, we can either apply the weakening or the abstraction rule. If we use the weakening rule, we need to derive that $\lambda y.(\lambda x.x\ y)\ x$ has a certain type under the empty context. This is, of course, not possible because the term is not closed (the last occurrence of the variable x is free) and the context is empty. Thus, the only possibility is to apply the abstraction rule and then, we have to give a derivation of $x{:}\sigma \to \tau, y{:}\sigma \vdash_{Wk} (\lambda x.x\ y)\ x : \tau$. Once more, we have two possible choices: the weakening or the application rule. If we apply the weakening rule, we should derive that $x{:}\sigma \to \tau \vdash_{Wk} (\lambda x.x\ y)\ x : \tau$. Here, the variable y is free in the term but there is no declaration in the context for the variable. Hence, the weakening rule is discarded and the only possible choice is to use the application rule. So far, the derivation looks as follows:

$$\dfrac{\dfrac{\dfrac{x{:}\sigma \to \tau, y{:}\sigma \vdash_{Wk} \lambda x.x\ y : (\sigma \to \tau) \to \tau \qquad x{:}\sigma \to \tau, y{:}\sigma \vdash_{Wk} x : \sigma \to \tau}{x{:}\sigma \to \tau, y{:}\sigma \vdash_{Wk} (\lambda x.x\ y)\ x : \tau}}{x{:}\sigma \to \tau \vdash_{Wk} \lambda y.(\lambda x.x\ y)\ x : \sigma \to \tau}}{\vdash_{Wk} \lambda x.\lambda y.(\lambda x.x\ y)\ x : (\sigma \to \tau) \to \sigma \to \tau}$$

It is easy to see that the derivation of $x{:}\sigma \to \tau, y{:}\sigma \vdash_{Wk} x : \sigma \to \tau$ can be carried out with no problem. However, $x{:}\sigma \to \tau, y{:}\sigma \vdash_{Wk} \lambda x.x\ y : (\sigma \to \tau) \to \tau$ is not derivable. Assume that we can derive it. Then, we have two possibilities: either we apply the weakening rule or the abstraction rule. If we apply the weakening rule, the variable y occurs free in the term $\lambda x.x\ y$ but there would be no declaration in the context for that variable. On the other hand, we cannot apply the abstraction rule because we would have a context of the form $x{:}\sigma \to \tau, y{:}\sigma, x{:}\sigma \to \tau$ which is not valid. Thus, we have shown that $\nvdash_{Wk} \lambda x.\lambda y.(\lambda x.x\ y)\ x : (\sigma \to \tau) \to \sigma \to \tau$.

Another consequence of the above result is that *Wk* is not closed under α-conversion. This is immediate since $\vdash_{Wk} \lambda x.\lambda y.(\lambda z.z\ y)\ x : (\sigma \to \tau) \to \sigma \to \tau$ is derivable in *Wk* and $\lambda x.\lambda y.(\lambda x.x\ y)\ x =_\alpha \lambda x.\lambda y.(\lambda z.z\ y)\ x$.

With the following three lemmas, we show that the systems *Lt*, *St* and *Th* are equivalent.

Lemma 15. If Γ valid and $\Gamma \vdash_{Lt} M : \tau$, then $\Gamma \vdash_{St} M : \tau$.

Proof: The proof is by induction on the term M. We show here the case where M is an abstraction of the form $\lambda x.N$. All other cases are straightforward.

To type the term $\lambda x.N$ in *Lt* we use the abstraction rule which has $\Gamma, x{:}\tau_1 \vdash_{Lt} N : \tau_2$ as a premise.

If $x \notin \mathsf{dom}(\Gamma)$ then $\Gamma = \Gamma_x$. Thus, the desired result is obtained by applying the abstraction rule for *St* to the induction hypothesis for the premise.

If $x \in \mathsf{dom}(\Gamma)$ then, there exist Γ_0, Γ_1 and σ such that $\Gamma = \Gamma_0, x{:}\sigma, \Gamma_1$, and $x{:}\sigma$ is unnecessary for N in $\Gamma_1, x{:}\tau_1$. Thus, by applying the strengthening lemma for *Lt* (lemma 5), we have that $\Gamma_0, \Gamma_1, x{:}\tau_1 \vdash_{Lt} N : \tau_2$. Now we can apply the induction hypothesis to obtain the same derivation but in *St*. Finally, we apply the abstraction rule for *St*, which allows us to introduce the declaration $x{:}\sigma$ again in the context, and obtain the desired result. ■

Lemma 16. If $\Gamma \vdash_{St} M : \tau$ then $\Gamma \vdash_{Th} M : \tau$.

Proof: The proof is by induction on the term M.

In the case where M is an abstraction of the form $\lambda x.N$ and $x \in \mathsf{dom}(\Gamma)$, by applying the abstraction rule for *Th* to the induction hypothesis we have $\Gamma_x \vdash_{Th} \lambda x.N : \tau_1 \to \tau_2$. As we have that $\Gamma_x \leqslant_{Th} \Gamma$, by applying the thinning rule for system *Th* we obtain the desired result. ■

Lemma 17. If $\Gamma \vdash_{Th} M : \tau$ then $\Gamma \vdash_{Lt} M : \tau$.

Proof: The proof is by induction on the derivation of $\Gamma \vdash_{Th} M : \tau$. We show here the case for the thinning rule. All other cases are straightforward.

In the case where we obtain $\Gamma \vdash_{Th} M : \tau$ by applying the thinning rule, we have as premises that $\Delta \vdash_{Th} M : \tau$ and $\Delta \leqslant_{Th} \Gamma$. By the induction hypothesis we have that $\Delta \vdash_{Lt} M : \tau$. It can easily be shown that if $\Delta \leqslant_{Th} \Gamma$ then $\Delta \leqslant_{Lt} \Gamma$ and thus, we can apply the thinning lemma for *Lt* (lemma 8) to obtain the desired result. ∎

Theorem 18. (Equivalence of *Lt*, *St* and *Th*). For valid contexts, we have the following equivalences: $Lt = St = Th$.

Proof: Immediately from the lemmas 15, 16 and 17. ∎

As a consequence of the previous theorem and of corollary 12, we have that *St* and *Th* are closed under α-conversion.

Corollary 19. (Closure under α-conversion for *St* and *Th*). Let $ts \in \{St, Th\}$. If $\Gamma \vdash_{ts} M : \tau$ and $M =_\alpha M'$, then $\Gamma \vdash_{ts} M' : \tau$.

To finish, we summarise the relations we have proven.

$$Us \subset Wk \subset Lt = St = Th$$

6 Conclusions

In this section, we briefly discuss the problem of α-conversion for two extensions of simply typed λ-calculus.

We believe that our study of α-conversion for simply typed λ-calculus can be extended to include polymorphic definitions as in [7] and fix point operators. It seems to us that the type systems *Lt*, *St* and *Th* extended with these two new constructors are still closed under α-conversion.

It is more interesting to consider the problem of α-conversion for λ-calculus with dependent types.

The formulations of *St* and *Th* for dependent types are not closed under α-conversion. For A type and P a family of types over A, we can derive that $\lambda x{:}A.\lambda y{:}Px.y$ has type $\Pi x{:}A.\Pi y{:}Px.Px$ but we cannot derive a type for $\lambda x{:}A.\lambda x{:}Px.x$ since the context $x{:}A, x{:}Px$ is not valid.

The straightforward extension of *Lt* with dependent types is not correct as it is shown in [12]. In this system, we derive that $\lambda x{:}A.\lambda x{:}Px.x$ has type $\Pi x{:}A.\Pi x{:}Px.Px$, which is not the correct type for the term.

Some work has been done towards the formulation of type systems for λ-calculus with dependent types that are closed under α-conversion.

The type systems with dependent types which use de Bruijn indices are trivially closed under α-conversion. However, as already pointed out in section 1, terms using these indices are unreadable for human eyes.

In [5], Coquand suggests a type system with dependent types that is closed under α-conversion. A simplification of this system for simple typed λ-calculus has been presented by Pollack in his paper [12]. This solution uses two disjoint sets: the usual set of variables and a set of parameters. Here, to type an abstraction of the form $\lambda x.M$, we type $M\,[p/x]$ in a context extended with a declaration for p, being p a fresh parameter. Then, we can see this abstraction rule as having an implicit renaming rule. A similar solution for dependent types has been presented and proved correct by Pollack in his thesis [13].

From the logical point of view, it is interesting to study the inhabitation of types. To study the inhabitation of types in the simply typed λ-calculus, it does not make any difference if we add α-conversion or not. However in some extensions of the λ-calculus, α-conversion can increase the set of inhabited types. In [11], it is shown that there is a type inhabited in system F_ω with α-conversion on types which is not inhabited in F_ω without α-conversion.

Acknowledgement. We are grateful to Bjórn von Sydow for listening and discussing our ideas and for his comments on earlier versions of this paper. We also want to thank Verónica Gaspes and Alvaro Tasistro for useful discussions on this subject, and Johan Jeuring for reading and commenting on previous versions of this paper.

References

[1] H. Barendregt. *The Lambda Calculus.* North Holland, 1981.

[2] H. P. Barendregt. Lambda calculi with types. In Abramsky Gabbai and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[3] S. Berardi. Towards a mathematical analysis of the coquand-huet calculus of constructions and the other systems in barendregt's cube. Technical report, Dept. of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, 1988.

[4] A. Bove. A machine-assisted proof of the subject reduction property for a small typed functional language. Master's thesis, Department of Computer Science, University of the Republic, Uruguay, November 1995. Technical Report INCO-95-06. Available on the WWW http://md.chalmers.se/~bove/Papers/master_thesis.ps.gz.

[5] T. Coquand. An algorithm for testing conversion in type theory. In *Logical Frameworks.* Cambridge University Press, 1991.

[6] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[7] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th. ACM Symposium on Principles of Programming Languages*, Albuquerque NM, January 1982.

[8] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In J. H. Geuvers, R. P. Nederpelt, and R. C. de Vrijer, editors, *Selected Papers on Automath*, volume 133, pages 375–388. North Holland, 1994.

[9] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[10] M. P. Jones. *Qualified Types. Theory and Practice.* Cambridge University Press, 1994.

[11] A. J. Kfoury, S. Ronchi della Rocca, J. Tiuryn, and P. Urzyczyn. Alpha-conversion and typability. *Information and Computation*, 150(1), April 1999.

[12] R. Pollack. Closure under Alpha conversion. In *The Informal Proceeding of the 1993 Workshop on Types for Proofs and Programs*, May 1993.

[13] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[14] A. Tasistro. Machine-assisted application of constructive type theory to the theory of functional programming languages. a first experiment using alf. in PhD thesis, 1997. Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden.

[15] J. Terlouw. Een nadere bewijstheoretische analyse van gstt's. Technical report, Department of Computer Science, University of Nijmegen, 1989.

Received May 10th, 1999