A Machine-assisted Proof of the *Subject Reduction* Property for a Small Typed Functional Language

Ana Bove [1]

bove@fing.edu.uy

November, 1995

[1]Computer Science Department, Engineering School, University of the Republic, Montevideo, Uruguay. Partially supported by a Bid-Conicyt grant and by a Swedish Institute scholarship.

Abstract

We present an experiment in formally describing a programming language and its properties in constructive type theory. By constructive type theory we understand primarily the formulation of Martin-Löf's set theory. Constructive type theory can also be seen as a programming language where we write types, and objects of these types that can be view as functional programs. Thus, practical applicability of type theory depends on the availability of programming environments or proof assistants. The language we analyze is a small typed functional language. We present its syntax, its dynamic semantics and its type system. Among other properties, we present a formalization of the Subject Reduction property for the language. The proof assistant we use is ALF.

Contents

List of Figures

1 Introduction

We are interested in the formalization of the theory of programming languages in constructive type theory. By constructive type theory we understand primarily Martin-Löf's theory of logical types (or logical framework [Nor 90]). This is conceived as a formal language in which to carry out constructive mathematics. So what we want to do in the first place is to investigate constructive formalizations of the mathematics of programs.

Constructive type theory can also be viewed as a programming language. More precisely, in type theory we write types and objects of these types, and the objects are functional programs. Proofs of propositions are objects (programs) in type theory. Following the constructive interpretations, a proof of a theorem is in general a function. Given proofs of the assumptions of the theorem this function computes the proof of the conclusion. In particular, when the theorem states the existence of an object with certain properties, the proof of the theorem computes such an object from any given proofs of the assumptions of the theorem. In this way, many important algorithms used in the process of interpretation or compilation of programming languages arise naturally as proofs of properties of the languages. In other words, the formalization of the relevant parts of the theory of programming languages gives implementations of these languages that are correct by construction. So another motivation of our work is to actually carry out this idea in practice, that is, to investigate the production of verified implementations of programming languages.

The experiments we want to perform will test type theory as a formal language and the implementation of type theory. We hope to obtain ideas about how to design better programming environments for type theory.

In this paper we use the proof editor ALF ([Alt 94, Mag 92, Mag 94]) for Martin-Löf's monomorphic type theory. We use the pattern matching facility in our proofs, which in fact is not a part of Martin-Löf's framework but which makes the proofs easier to carry out. In the proofs we also use the formalization of Martin-Löf's monomorphic set theory provided by the ALF library.

To begin with, we consider the theory of functional programming languages. In this paper, we consider a small polymorphic functional language and among others, the property that relates its type system with the evaluation of its expressions, more precisely the *Subject Reduction Property*. A natural next step is given by the property of existence of most general type schemes, i.e., the algorithm of type inference.

This paper is intended for readers who have some basic knowledge of operational semantics, type systems and ALF.

1.1 About this Paper

This paper is organized as follows :

In section 2 we briefly introduce type theory, and we present the language to be analyzed; that is, we present its syntax, its dynamic semantics and its type system. Furthermore, we describe informally the proofs of a number of properties of the language, concluding with the *Subject Reduction* property.

In section 3 we describe how we formalize the results presented in section 2 in ALF. We finish this section with some conclusions about ALF.

In section 4 we present related work and further work.

Finally, in appendix A we present the ALF code of all the definitions and theorems described in this paper. In this code we use the formalization of Martin-Löf's monomorphic set type theory provided by the ALF library.

2 Informal Presentation

In this section we briefly introduce type theory, and we present the language to be analyzed; that is, we present its syntax, its dynamic semantics and its type system. Furthermore, we describe informally the proofs of a number of properties of the language, concluding with the *Subject Reduction* property.

We use an informal mathematical language. We briefly explain the basic notions we use, which are essentially those of Martin-Löf's type theory. For an introduction to Martin-Löf's type theory see [Coq 94] or [Nor 90].

We have objects of various types, namely :

- **Sets** : A set is inductively defined, that is, a set is determined by the rules that construct its elements. We write *Set* to refer to the type of sets.
- **Elements of Sets** : For each set S, the elements of S are objects of type S.
- **Dependent Functions** : A dependent function is a function in which the type of the output depends on the value of the input. To form the type of a dependent function we first need a type α as domain and then a family of types over α. The concept of *family of types* over a type α is explained as follows : if β is a family of types over α, then to every object a of type α there is a corresponding type $\beta\, a$.

  Given a type α and a family of types β over α, we write $\alpha \to \beta$ for the type of dependent functions from α to β. If f is a function of type $\alpha \to \beta$, then by applying f to any object a of type α we obtain an object of type $\beta\, a$. We write fa for such an application.

  A non-dependent function is regarded as a special case of a dependent function. So, if α and β are types we also write $\alpha \to \beta$ for the type of functions mapping objects of type α to objects of type β.

In each case we write $a : \alpha$ for "a is (an object) of type α".

We want to consider predicates and relations on sets, and arbitrary complex propositions formed from these predicates and relations. Predicates and relations are naturally viewed in type theory as functions yielding propositions as output. So, if S is a set then, the predicate P over S is a function of type $S \to Prop$ where *Prop* is the type of propositions. The type of propositions is explained as follows : a proposition is determined by the rules that construct its proofs, i.e., it is inductively defined. This explanation allows us to identify propositions with sets, which is actually done in type theory. This identification has many consequences that are exploited in the formal language of type theory. However, for the informal presentation we prefer to follow the classical mathematical practice and distinguish sets and elements of sets from propositions about them and proofs of the propositions.

As mentioned before, propositions are inductively defined. So when introducing for instance a relation R on the sets S and S', that is, $R : S \to S' \to Prop$, we give the rules that construct the proofs of $R(a, b)$ for some $a : S$ and $b : S'$. We understand these rules as constituting the

inductive definition of the propositions $R(a, b)$ for each $a : S$ and $b : S'$. We write them as usual in the form :

$$\frac{P_1 \qquad \cdots \qquad P_n}{C}$$

If a and b are elements of the same set S, $a \equiv b$ denotes the definitional equality between these elements. So this is the symbol we use when introducing definitions. We write $a = b$ for the proposition that says that a and b are definitionally equal. In both cases, it is possible to subscript the set in the equality as in $a \equiv_S b$. But in most of the cases it is very easy to deduce the set S, so there is no need for writing it explicitly.

2.1 The Syntax of the Language

We define the set of expressions *Exp* by means of its abstract syntax in figure 1. For the definition of *Exp* we assume a (possibly infinite) set of variables *Var*. For each pair of variables in *Var* it is decidable whether or not they are equal.

The expressions of the language are those of a small functional language. There are variables x ; $\lambda\, x\, .\, e$ denotes an abstraction with respect to a variable in an expression, $(d\ \ e)$ the application of an expression to another one, **fix** $x\, .\, e$ the least fixed point of $\lambda\, x\, .\, e$, $true$ and $false$ the two boolean values and **if** d **then** e **else** f a conditional expression. In contrast to the usual functional languages, we do not allow defining local constants. Expressions of the form **let** $x\ =\ d$ **in** e are interesting for our investigation if we can derive a polymorphic type for them using the type system. The most convenient way of doing this uses *type schemes*. In this paper we are not interested in considering type schemes or type inference, so we work with a simple set of types and do not consider local definitions. We discuss the introduction of the **let** expressions in section 4.2.

Notation : In concrete expressions, we use parentheses to avoid ambiguity. From now on x, y, z (possibly primed) denote elements in the set *Var* and d, e, f, g denote elements in the set *Exp*.

We define the set **FV** of *free* variables of an expression as usual (see [Bar 92]). A *closed* expression is an expression with no free variables.

$Var : Set$
$Exp : Set$

$x \quad : Var$
$d\ ,\ e\ ,\ f \quad : Exp$

$e ::= x \ \mid \lambda\, x\, .\, e \ \mid (d\ \ e) \ \mid$ **fix** $x\, .\, e \ \mid true \mid false \mid$ **if** d **then** e **else** f

Figure 1: Abstract Syntax of *Exp*.

$$\_\,[\_\,/\_\,] : Exp \to Var \to Exp \to Exp$$

$$
\begin{array}{lcll}
y\,[d/x] & \equiv & y & \text{if } y \neq x \\
 & \equiv & d & \text{if } y = x \\
(\lambda\, y\,.\, e\,)[d/x] & \equiv & \lambda\, y\,.\,(e\,[d/x]) & \text{if } y \neq x \\
 & \equiv & \lambda\, y\,.\, e & \text{if } y = x \\
(e\ \ f)\,[d/x] & \equiv & (e\,[d/x]\ \ f\,[d/x]) & \\
(\mathbf{fix}\ y\,.\, e\,)[d/x] & \equiv & \mathbf{fix}\ y\,.\,(e\,[d/x]) & \text{if } y \neq x \\
 & \equiv & \mathbf{fix}\ y\,.\, e & \text{if } y = x \\
true\,[d/x] & \equiv & true & \\
false\,[d/x] & \equiv & false & \\
(\mathbf{if}\ e\ \mathbf{then}\ f\ \mathbf{else}\ g\,)[d/x] & \equiv & \mathbf{if}\ e\,[d/x]\ \mathbf{then}\ f\,[d/x]\ \mathbf{else}\ g\,[d/x] &
\end{array}
$$

Figure 2: Definition of the function $\_\,[\_\,/\_\,]$ over *Exp*.

For the definition of the dynamic semantics of the language we define the *substitution* of an expression for a variable in an expression. We write the substitution of the expression d for the (free occurrences of the) variable x in the expression e as $e\,[d/x]$. We present the definition of the function $\_\,[\_\,/\_\,]$ in figure 2.

Notice that we do not care about capturing variables in the definition of the function $\_\,[\_\,/\_\,]$. This is because we only consider evaluation of closed expressions and no evaluation takes place under a binding operator in the definition of the dynamic semantics (section 2.2). Then, variable capture cannot occur during evaluation. This choice is usual when implementing functional programming languages because it gives a powerful notion of evaluation from the users point of view and allows a great simplification of the substitution function.

2.2 The Dynamic Semantics of the Language

In this section we present the dynamic semantics of the language; that is, we describe how to evaluate an expression of the language. We also present some properties related to the evaluation of expressions.

If it exists, the result of evaluating an expression e , the *value* of e , is also an element in the set *Exp*. We show in section 2.2.1 that the dynamic semantics we present here is a partial function.

We give the dynamic semantics of the language in an operational style. There exist several approaches to operational semantics (for an introduction to operational semantics see [Plo 81]); we use *Evaluation Semantics* ([Hen 90]), also known as *Natural Semantics* ([Kah 87], [Nie 92]). Evaluation Semantics describes how to obtain the overall results of executions. It describes the relationship between the initial and the final state of the execution. In our case, the evaluation

semantics consists of a set of rules that capture the essence of how we calculate the value of an expression without excessive detail.

The dynamic semantics of the language is intended for closed expressions and we define it in figure 3. The rule **DSApp** implies that the dynamic semantics is *lazy*.

Expressions of the form $true$, $false$ and $\lambda\, x\, .\, e$ are themselves values. These expressions are the *canonical* expressions of the language.

There exist also closed expressions without a value. These are :

- Type incorrect expressions, such as for example the expression **if** $\lambda\, x\, .x$ **then** $true$ **else** $false$.
- Looping expressions, such as for example **fix** $x\, .\, x$.

2.2.1 Some Properties of the Dynamic Semantics

In this section we present some properties of the dynamic semantics of the language.

Proposition 1 *If d and e are two expressions such that $d \Rightarrow e$, then the expression e is a canonical expression.*

Proof. The proof is by straightforward induction on the derivation of $d \Rightarrow e$.

■

Proposition 2 (Unicity) *Every expression has at most one value.*

Proof. In general, given an expression e there is at most one rule for evaluating it. Except in the case of an **if** _ **then** _ **else** _ expression, the form of the expression determines the rule to apply. For an expression of the form **if** d **then** f **else** g, it is the value of expression d which determines the applicable rule.

■

This proposition tells us that the relation _ $\Rightarrow$ _ is a partial function.

$\Rightarrow : Exp \rightarrow Exp \rightarrow Prop$

$x \quad : Var$
$d\,,\, e\,,\, f \; : Exp$

DSAbs Rule

$$\frac{}{\lambda\, x\,.\, e \;\Rightarrow\; \lambda\, x\,.\, e}$$

DSApp Rule

$$\frac{d \;\Rightarrow\; \lambda\, x\,.\, f \qquad f\,[e/x] \;\Rightarrow\; g}{(d\;\; e) \;\Rightarrow\; g}$$

DSFix Rule

$$\frac{e\,[\textbf{fix}\; x\,.\, e\,/x] \;\Rightarrow\; g}{\textbf{fix}\; x\,.\, e \;\Rightarrow\; g}$$

DSTrue Rule

$$\frac{}{true \;\Rightarrow\; true}$$

DSFalse Rule

$$\frac{}{false \;\Rightarrow\; false}$$

DSIf_True Rule

$$\frac{d \;\Rightarrow\; true \qquad e \;\Rightarrow\; g}{\textbf{if}\; d\; \textbf{then}\; e\; \textbf{else}\; f \;\Rightarrow\; g}$$

DSIf_False Rule

$$\frac{d \;\Rightarrow\; false \qquad f \;\Rightarrow\; g}{\textbf{if}\; d\; \textbf{then}\; e\; \textbf{else}\; f \;\Rightarrow\; g}$$

Figure 3: Inductive Definition of the Dynamic Semantics of the Language.

2.3 The Type System

In this section we present the type system for the language. Before presenting it, we should introduce the types themselves and the notion of *context*.

2.3.1 The Set of Types

We define the set of types *Types* by means of its abstract syntax in figure 4. We do not consider type variables or type schemes in this paper.

Notation : In concrete types, we use parentheses to avoid ambiguity. From now on α , β , γ , δ denote elements in the set *Type*.

2.3.2 The Contexts for the Type System

The contexts we use in the type system are (in principle) lists of associations of the form $x : \alpha$. We call these associations *declarations*. *Decl* is the set of declarations.

We are only interested in those contexts where each variable is declared at most once. Therefore, we restrict the notion of contexts as lists of declarations given above. For that, we use the relation **fresh** : $Var \rightarrow Ctxt \rightarrow Set$. By x **fresh** Γ we mean that "Γ contains no declarations for the variable x ". The definition of **fresh** is mutually recursive with the definition of contexts. We define the set of contexts *Ctxt* by giving the rules that construct its elements in figure 5.

Notation : In concrete contexts, we use parentheses to avoid ambiguity. In this section Γ , Δ (possibly primed) denote elements in the set *Ctxt*.

In order to present the *Substitution Lemma* for the language, we define the concatenation of two *disjoint* contexts. By disjoint contexts we mean that there is no variable x which is declared in both contexts. We denote the function that concatenates two disjoint contexts by $\_ \,\&\, \_$. We define it as usual, by induction on the second argument.

$Type : Set$

$\alpha , \beta : Type$

$\alpha ::= \mathbf{Bool} \mid \alpha \rightarrow \beta$

Figure 4: Abstract Syntax of the Set of Types.

$Ctxt : Set$

$x \quad : Var$
$\alpha \quad : Type$
$\Gamma \quad : Ctxt$

Empty_Ctxt $\dfrac{}{[\,] : Ctxt}$

Cons_Ctxt $\dfrac{x \ \textbf{fresh}\ \Gamma}{[\Gamma\ ,\ x\ :\alpha\ ]\ :\ Ctxt}$

Figure 5: Inductive Definition of Contexts.

2.3.3 Presentation of the Type System

In this section we present the type system. The type system tells us when an expression e has type α under a context Γ. We define the type system in figure 6.

For our purpose we adapt a type system presented in [Geu 90] for Pure Type Systems. The type system we present in this paper has one rule for each expression form plus a thinning rule. The rules are rather straightforward. Notice that the rules **TSVar** and **TSThinn** look up a variable in a context and that in all the rules in which we add a new declaration $x : \alpha$ to the context Γ in the conclusion of the rule, we check that the variable x is fresh for Γ, so that $[\Gamma\ ,\ x\ :\alpha\ ]$ is a context.

The thinning rule (**TSThinn Rule**) plays an important role when typing expressions where the same variable is abstracted more than once, such as for example the expression $\lambda x\,.\,\lambda x\,.\,e$. For typing this expression we cannot use simply the rule **TSAbs** twice because the second time the variable x will not be fresh in the context. Instead, we first type the expression $\lambda x\,.\,e$ with the rule **TSAbs**, then add the variable x to the context with the rule **TSThinn** and then apply the rule **TSAbs** again to obtain the type of the complete expression.

Instead of this type system we could have presented an equally expressive system with no thinning rule and in which the context is a simple list of declarations. Then, we can declare a variable more than once in a context. We look up a variable in a context from right to left. Because we can declare a variable more than once in a context, in the rule **TSVar** we do not need to check if the variable added is fresh. The rest of the rules are the same as here. Although the Subject Reduction property also holds in this system, its proof is more complex than the one we present in this paper.

$\vdash : \mathit{Ctxt} \to \mathit{Exp} \to \mathit{Type} \to \mathit{Prop}$

$x : \mathit{Var}$
$d\,,\, e\,,\, f : \mathit{Exp}$
$\alpha\,,\, \beta : \mathit{Type}$
$\Gamma : \mathit{Ctxt}$

TSVar Rule

$$\frac{x \ \textbf{fresh}\ \Gamma}{[\Gamma\,,\, x\ :\alpha\ ] \vdash x\ :\alpha}$$

TSThinn Rule

$$\frac{\Gamma \vdash e\ :\alpha \qquad x\ \textbf{fresh}\ \Gamma}{[\Gamma\,,\, x\ :\beta\ ] \vdash e\ :\alpha}$$

TSAbs Rule

$$\frac{[\Gamma\,,\, x\ :\alpha\ ] \vdash e\ :\beta}{\Gamma \vdash \lambda\, x\,.\, e\ :\alpha \to \beta}$$

TSApp Rule

$$\frac{\Gamma \vdash e\ :\alpha \to \beta \qquad \Gamma \vdash f\ :\alpha}{\Gamma \vdash (e\ \ f)\ :\beta}$$

TSFix Rule

$$\frac{[\Gamma\,,\, x\ :\alpha\ ] \vdash e\ :\alpha}{\Gamma \vdash \textbf{fix}\ x\,.\, e\ :\alpha}$$

TSTrue Rule

$$\frac{}{\Gamma \vdash \mathit{true}\ :\textbf{Bool}}$$

TSFalse Rule

$$\frac{}{\Gamma \vdash \mathit{false}\ :\textbf{Bool}}$$

TSIf Rule

$$\frac{\Gamma \vdash d\ :\textbf{Bool} \qquad \Gamma \vdash e\ :\alpha \qquad \Gamma \vdash f\ :\alpha}{\Gamma \vdash \textbf{if}\ d\ \textbf{then}\ e\ \textbf{else}\ f\ :\alpha}$$

Figure 6: Inductive Definition of the Type System for the Language.

The discussion on the choice of the type system is not exhausted by the above observations but we will not discuss this topic further here.

2.4 The *Subject Reduction* Theorem

In this section we present some properties related to the type system of the language. We end up this section with a proof of the *Subject Reduction* property. In our case, this property says that if a closed expression e of type α has a value, this value is also a closed expression of type α.

For the proof of this property we need to prove the *Substitution Lemma* which states (informally) that if an expression e has a type under a context where the variable x (that possible occurs free in e) is declared to have type α, the result of substituting an expression d that also has type α for x in e is an expression of the same type as the original one. We present it formally in lemma 4.

We need an auxiliary lemma in order to prove the *Substitution Lemma*. This lemma says that if a variable x is not free in an expression e, then the result of substituting an expression d for x in e is equal to e. We express the fact : "the variable x is not free in an expression e" as "we can derive that e has a type α under a context Γ that does not contain any declaration of the variable x".

Lemma 3 *Let* Γ *be a context,* e *an expression and* α *a type. If we can derive* $\Gamma \vdash e : \alpha$ *and the variable* x *is fresh for* Γ, *then for any expression* d, $e = e\,[d/x]$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash e : \alpha$. We discuss two cases here. The other cases are straightforward.

- **Case TSVar Rule** : The rule we apply is $\dfrac{y \ \textbf{fresh}\ \Gamma}{[\Gamma\,,\, y : \alpha\,] \vdash y : \alpha}$

  Notice that $x \neq y$ because x **fresh** $[\Gamma\,,\, y : \alpha\,]$ and $x = y$ cannot hold at the same time.

  Then, the result follows from the fact that $y\,[d/x] \equiv y$ by definition of the function $\_\,[\_\,/\_\,]$.

- **Case TSAbs Rule** : The rule we apply is $\dfrac{[\Gamma\,,\, y : \beta\,] \vdash f : \gamma}{\Gamma \vdash \lambda\, y\,.\, f : \beta \rightarrow \gamma}$

  We consider whether or not the variables x and y are the same.

  - $x = y$: The result follows immediately from the fact that $(\lambda\, y\,.\, f\,)[d/x] \equiv \lambda\, y\,.\, f$ by definition of the function $\_\,[\_\,/\_\,]$.

– $x \neq y$: By definition of the function $\_[\_/\_]$ we have $(\lambda\, y\,.\, f\,)[d/x] \equiv \lambda\, y\,.\,(\mathrm{f}[d/x])$. Since x **fresh** $[\Gamma\,,\, y\ :\beta\ ]$ because x **fresh** Γ and $x \neq y$, we get $f\ =\ f\,[d/x]$ by the induction hypothesis. Then, we c

■

Usually, in informal presentations we find the *Substitution Lemma* formulated as follows :

Let Γ *be a context,* d *and* e *expressions,* x *a variable and* α *and* β *types. If we can derive* $[\Gamma\,,\, x\ :\alpha\ ] \vdash e\ :\beta$ *and* $\Gamma \vdash d\ :\alpha$ *, then we can also derive* $\Gamma \vdash e\,[d/x] : \beta$.

Although this formulation is slightly restrictive because the variable x must be the last variable added in the context, it is enough for our purposes in the proof of the *Subject Reduction* property. However, this formulation of the lemma cannot be proved directly by induction on derivations in our type system. Therefore, we formulate and prove the lemma in a more general way, allowing the variable x to occur at any position in the context.

Theorem 4 (Substitution Lemma) *Let* $[\Gamma\,,\, x\ :\alpha\ ]$ *and* Δ *be two disjoint contexts,* d *and* e *expressions,* x *a variable and* α *and* β *types. If we can derive* $[\Gamma\,,\, x\ :\alpha\ ]\ \&\ \Delta\ \vdash e\ :\beta$ *and* $\Gamma \vdash d\ :\alpha$ *, then we can also derive* $\Gamma\ \&\ \Delta\ \vdash e\,[d/x] : \beta$.

Proof. The proof is by induction on the derivation of $[\Gamma\,,\, x : \alpha\,]\ \&\ \Delta \vdash e : \beta$. Notice that if $[\Gamma, x : \alpha]$ and Δ are disjoint contexts, then so are Γ and Δ. Thus, in this case, $\Gamma\ \&\ \Delta$ is a context.

- **Case TSVar Rule** : We consider the two cases for context Δ .

  – $\Delta\ \equiv\ [\,]$: The rule we apply is $\dfrac{x\ \textbf{fresh}\ \Gamma}{[\Gamma\,,\, x\ :\alpha\ ] \vdash x\ :\alpha}$

  The result follows from the facts that $x\,[d/x] \equiv d$ by definition of the function $\_[\_/\_]$ and that $\Gamma \vdash d\ :\alpha$ by assumption.

  – $\Delta\ \equiv\ [\Delta'\,,\, y\ :\beta\ ]$: The rule we apply is $\dfrac{y\ \textbf{fresh}\ [\Gamma\,,\, x\ :\alpha\ ]\ \&\ \Delta'}{[[\Gamma\,,\, x\ :\alpha\ ]\ \&\ \Delta'\,,\, y\ :\beta\ ] \vdash y\ :\beta}$

  Notice that $x \neq y$ because y **fresh** $[\Gamma\,,\, x\ :\alpha\ ]\ \&\ \Delta'$ and $x\ =\ y$ cannot hold at the same time.
  Since y **fresh** $[\Gamma\,,\, x\ :\alpha\ ]\ \&\ \Delta'$, we have y **fresh** $\Gamma\ \&\ \Delta'$. Now, we can apply the variable rule to get $\Gamma\ \&\ \Delta' \vdash y\ :\beta$. Then, we conclude $\Gamma\ \&\ \Delta' \vdash y\,[d/x] : \beta$ as desired because $y\,[d/x] \equiv\ y$ by definition of the function $\_[\_/\_]$.

- **Case TSThinn Rule** : Again, we consider the two cases for context Δ .

  - $\Delta \equiv [\,]$: The rule we apply is $\dfrac{\Gamma \vdash e : \beta \qquad x \ \textbf{fresh}\ \Gamma}{[\Gamma\,, x : \alpha\,] \vdash e : \beta}$

    From the assumptions x **fresh** Γ and $\Gamma \vdash e : \beta$, using lemma 3, we get $e = e[d/x]$. Thus , as we have $\Gamma \vdash e : \beta$ by assumption, we can conclude $\Gamma \vdash e[d/x] : \beta$ as desired.

  - $\Delta \equiv [\Delta'\,, y : \gamma\,]$: The rule we apply is

    $$\frac{[\Gamma\,, x : \alpha\,] \ \&\ \Delta' \vdash e : \beta \qquad y \ \textbf{fresh}\ [\Gamma\,, x : \alpha\,] \ \&\ \Delta'}{[[\Gamma\,, x : \alpha\,] \ \&\ \Delta'\,, y : \gamma\,] \vdash e : \beta}$$

    We get $\Gamma \ \&\ \Delta' \vdash e[d/x] : \beta$ by the induction hypothesis. Since y **fresh** $[\Gamma, x : \alpha] \ \&\ \Delta'$, we have y **fresh** $\Gamma \ \&\ \Delta'$. Then, we can apply the thinning rule to conclude $\Gamma \ \&\ [\Delta'\,, y : \gamma\,] \vdash e\,[d/x] : \beta$ as desired.

- **Case TSAbs Rule** : The rule we apply is $\dfrac{[[\Gamma\,, x : \alpha\,] \ \&\ \Delta\,, y : \gamma\,] \vdash f : \delta}{[\Gamma\,, x : \alpha\,] \ \&\ \Delta \vdash \lambda y\,.\,f : \gamma \to \delta}$

  Again, notice that $x \neq y$ for the same reason mentioned before.

  We get $\Gamma \ \&\ [\Delta\,, y : \gamma\,] \vdash f\,[d/x] : \delta$ by the induction hypothesis. Then, we can apply the abstraction rule to conclude $\Gamma \ \&\ \Delta \vdash \lambda y\,.\,(f\,[d/x]) : \gamma \to \delta$ as desired, because $(\lambda y\,.\,f\,)[d/x] \equiv \lambda y\,.\,(f\,[d/x])$ by definition of the function $\_\,[\_\,/\_\,]$.

- **Case TSApp Rule** : The rule we apply is

  $$\frac{[\Gamma\,, x : \alpha\,] \ \&\ \Delta \vdash f : \gamma \to \beta \qquad [\Gamma\,, x : \alpha\,] \ \&\ \Delta \vdash g : \gamma}{[\Gamma\,, x : \alpha\,] \ \&\ \Delta \vdash (f\ \ g) : \beta}$$

  We get both $\Gamma \ \&\ \Delta \vdash f[d/x] : \gamma \to \beta$ and $\Gamma \ \&\ \Delta \vdash g[d/x] : \gamma$ by the induction hypothesis. Then, we can apply the application rule to conclude $\Gamma \ \&\ \Delta \vdash (f\,[d/x]\ \ g\,[d/x]) : \beta$ as desired, because $(f\ \ g)\,[d/x] \equiv (f\,[d/x]\ \ g\,[d/x])$ by definition of the function $\_\,[\_\,/\_\,]$.

- **Case TSFix Rule** : Analogous to the abstraction case.

- **Case TSTrue Rule** : The rule we apply is $\dfrac{}{[\Gamma\,, x : \alpha\,] \ \&\ \Delta \vdash \mathit{true} : \textbf{Bool}}$

  By definition of the function $\_\,[\_\,/\_\,]$ we have $\mathit{true}\,[d/x] \equiv \mathit{true}$. Then, the result follows from applying the rule for the true expressions.

- **Case TSFalse Rule** : Analogous to the previous case.

- **Case TSIf Rule** : Analogous to the application case.

■

In our case, the *Subject Reduction* property only applies to closed expressions. To see that in our case this property does not hold for open expressions we present the following example. Consider the context $[[\ ]\ ,\ y\ :\alpha\ ]$ and the expression $((\lambda x\,.\,\lambda y\,.\,x\ )\ \ y\ )$. Notice that this expression should have the expression $\lambda y\,.\,z$ as value. It is easy to see that both $[[\ ]\,,y\ :\alpha\ ] \vdash ((\lambda x\,.\,\lambda y\,.\,x\ )\ \ y\ ) : \beta \rightarrow \alpha$ and $((\lambda x\,.\,\lambda y\,.\,x\ )\ \ y) \Rightarrow \lambda y\,.\,y$ are derivable in our type system but not $[[\ ]\,,y\ :\alpha\ ] \vdash \lambda y\,.\,y : \beta \rightarrow \alpha$. Instead $[[\ ]\,,y\ :\alpha\ ] \vdash \lambda y\,.\,y : \beta \rightarrow \beta$ is derivable. The problem arises here because we evaluate an open expression and in the evaluation process we capture a variable. This capturing changes the meaning of the resulting expression and thus its type. Therefore, if we want the *Subject Reduction* property to also hold for open expressions we have to change the substitution function in order to avoid the capturing of variables.

Theorem 5 (Subject Reduction) *Let d and e be expressions and α a type. If $d \Rightarrow e$ and $[\ ] \vdash d\ :\alpha$, then $[\ ] \vdash e\ :\alpha$.*

Proof. The proof is by induction on the derivation of $d\ \Rightarrow\ e$. For each case in this derivation we consider the possible cases in the derivation of $[\ ] \vdash d\ :\alpha$. Notice that neither the variable rule nor the thinning rule can be applied in the derivation of $[\ ] \vdash d\ :\alpha$ because in the conclusion of these rules the context must contain at least one declaration.

- **Case DSAbs Rule** : The rule we apply is $\dfrac{}{\lambda x\,.\,f\ \Rightarrow\ \lambda x\,.\,f}$

  The result follows immediately from the assumption $[\ ] \vdash \lambda x\,.\,f\ :\alpha$.

- **Case DSApp Rule** : The rule we apply is $\dfrac{f\ \Rightarrow\ \lambda x\,.\,f' \qquad f'\,[g/x] \Rightarrow\ e}{(f\ \ g)\ \Rightarrow\ e}$

  The only applicable rule for $[\ ] \vdash (f\ \ g)\ :\alpha$ is the application rule, so the rule we apply is

  $$\frac{[\ ] \vdash f\ :\beta\ \rightarrow\ \alpha \qquad [\ ] \vdash g\ :\beta}{[\ ] \vdash (f\ \ g)\ :\alpha}$$

  Since both $f \Rightarrow \lambda x\,.f'$ and $[\ ] \vdash f : \beta \rightarrow \alpha$ hold by assumption, we have by the induction hypothesis that $[\ ] \vdash \lambda x\,.\,f'\ :\beta\ \rightarrow\ \alpha$. Then, it follows that $[[\ ]\,,x\ :\beta\ ]\ \&\ [\ ] \vdash f'\ :\alpha$ holds. This is because the only way to obtain $[\ ] \vdash \lambda x\,.\,f' : \beta \rightarrow \alpha$ from the type system is by applying the abstraction rule to $[[\ ]\,,x\ :\beta\ ] \vdash f'\ :\alpha$. The other rules cannot be applied either because the form of the expression does not match or because the context must be inhabited (for the variable or the thinning rule to be applicable). Now, we can apply the *Substitution Lemma* to $[[\ ]\,,x\ :\beta\ ]\ \&\ [\ ] \vdash f'\ :\alpha$ and to the assumption $[\ ] \vdash g\ :\beta$ to obtain $[\ ] \vdash f'\,[g/x] : \alpha$. Since both $f'\,[g/x] \Rightarrow\ e$ and $[\ ] \vdash f'\,[g/x] : \alpha$ hold, then we conclude by the induction hypothesis that $[\ ] \vdash e\ :\alpha$ also hods, as desired.

- **Case DSFix Rule** : Analogous to the previous case.
- **Case DSTrue Rule** : Analogous to the abstraction case.
- **Case DSFalse Rule** : Analogous to the abstraction case.

- **Case DSIf_True Rule** : The rule we apply is $\dfrac{f \Rightarrow true \qquad g \Rightarrow e}{\textbf{if } f \textbf{ then } g \textbf{ else } g' \Rightarrow e}$

  The only applicable rule for $[\,] \vdash \textbf{if } f \textbf{ then } g \textbf{ else } g' : \alpha$ is the rule for the **if then else** expressions, so the rule we apply is $\dfrac{[\,] \vdash f : \textbf{Bool} \qquad [\,] \vdash g : \alpha \qquad [\,] \vdash g' : \alpha}{[\,] \vdash \textbf{if } f \textbf{ then } g \textbf{ else } g' : \alpha}$

  Since $g \Rightarrow e$ and $[\,] \vdash g : \alpha$ hold, we have by the induction hypothesis that $[\,] \vdash e : \alpha$, as desired.

- **Case DSIf_False Rule** : Analogous to the previous case.

■

3 Working with ALF

In this section we describe how we formalize the results presented in section 2 in ALF. We finish this section with some conclusions about ALF.

ALF (Another Logical Framework) is an interactive editor for Martin-Löf's monomorphic type theory. In Martin-Löf's type theory theorems are identified with types and a proof is an object of the type, generally a function mapping proofs of the assumptions into proofs of its conclusion. ALF ensures that the objects we construct are well-formed and well-typed. Since proofs are objects, checking well-typing of objects amounts to checking correctness of proofs. For an introduction to ALF see [Alt 94], [Aug 90], [Mag 92] and [Mag 94].

In nearly all cases, the formalization of the definitions and theorems presented in the last section is direct. There are two points that need further explanation : the formalization of contexts (and subsequently, the formalization of the type system) and the formalization of the *Substitution Lemma*. Although the formalization of variables and in particular the formalization of the decidability of their equality is straightforward, we explain it because nearly all the proofs rely on the fact that equality between variables is decidable. We discuss these formalizations in section 3.1.

In appendix A we present the ALF code of all the definitions and theorems introduced in this paper.

3.1 Formalization in ALF

3.1.1 Formalization of the Set of Variables

A variable is encoded as a natural number. Thus, we formally define a (constructor) function $var : (n : N)VAR$ that given a natural number returns a variable, where N is the set of natural numbers and VAR the set of variables. Two variables $var(n)$ and $var(m)$ are equal if their "indices" n and m are equal.

Since the equality between natural numbers is decidable, so is the equality between variables. Thus, we define the function $IdVarDec : (x : VAR; y : VAR)Or(Id(x,y),Not(Id(x,y)))$ (see A.1) that tells us whether or not the variables x and y are equal. The two cases $x = y$ and $x \neq y$ in the informal proofs are translated to the different cases for the proof of $IdVarDec(x,y)$ in the formal proofs.

3.1.2 Formalization of Contexts

With the presentation of contexts as in section 2.3.2, that Γ is a context implies that Γ does not declare a variable twice, because when adding a new declaration $x : \alpha$ to the context Γ we check that the variable x is **fresh** for Γ. So with each context we not only have the list of the declarations of the context, but also the proofs of the side conditions about the freshness of the variables added to the list. A direct way of formalizing the contexts in ALF from the definition

presented in section 2.3.2 is as follows :

$$Ctxt : Set$$

$$\begin{array}{lcl} EmptyCtxt & : & Ctxt \\ ConsCtxt & : & (\Gamma : Ctxt)(\mathrm{dl} : Decl)(\mathrm{p} : Fresh(VarDecl(\mathrm{dl}),\Gamma))Ctxt \end{array}$$

where *VarDecl* is the function that returns the variable of a declaration and *Fresh* is the formalization of the predicate **fresh**.

For the formalization of the *Substitution Lemma* in section 3.1.3 we have to deal with equality of contexts. With the present definition, the contexts $ConsCtxt(\Gamma, \mathrm{d}, \mathrm{p})$ and $ConsCtxt(\Gamma', \mathrm{d}', \mathrm{p}')$ are equal if Γ , Γ' and d, d′ are equal and the proofs p and p′ are equal. But in practice, two contexts are equal if they contain the same declarations in the same order, and the proofs about the freshness of the variables do not matter.

We have a similar problem with the definition of the function that concatenates two contexts. The direct formalization of such a function is as follows :

$$Concat : (\Gamma , \Delta : Ctxt)(\mathrm{p} : Disjoint(\Gamma ,\Delta))Ctxt$$

where *Disjoint* is a predicate that formalizes the notion that two contexts are disjoint.

In order to work with the type system we need to know that the contexts we use are valid, in the sense that they declare each variable at most once. So we need to be sure that when we use a list of declarations Γ for typing an expression e , Γ is a valid context. However, we do not need to have the proofs about the freshness of the variables we add to a context and the proofs that two contexts are disjoint when concatenating them.

Our solution is to formalize a context as a list of declarations and to define a predicate **Context** that tells us whether or not a list of declarations is a valid context. We present the definition of this predicate in figure 7. Because the notions of contexts and lists of declarations are similar we abuse notation and use the same abstract syntax for both notions.

Notation : In this section Γ , Δ , Σ (possibly primed) denote elements in the set *ListDecl*.

We use lists of declarations instead of contexts in the rules of the type system. To ensure that each list of declarations Γ used for typing an expression e is valid we change the type system. We present a modified type system for the language in figure 8. We can easily see that the definition of the (previous) type system presented in figure 6 is equivalent to the one we present here in the sense that each time we can derive $\Gamma \vdash e : \alpha$ with the former system we can also derive $\Gamma' \vdash e : \alpha$ with the latter one, where Γ' is the list that contains the same declarations as the context Γ and in the same order.

The following lemma states that with these new definitions, each time we use a list Γ for typing an expression e in the type system, the list is a valid context.

x : *Var*
α : *Type*
Γ : *ListDecl*

Empty_Ctxt $\dfrac{}{[\,]\ \textbf{Context}}$

Cons_Ctxt $\dfrac{\Gamma\ \textbf{Context} \qquad x\ \textbf{fresh}\ \Gamma}{[\Gamma\ ,\ x\ :\alpha\ ]\ \textbf{Context}}$

Figure 7: Inductive Definition of the Predicate Context.

Lemma 6 *Let* Γ *be a list of declarations,* e *an expression and* α *a type. If we can derive* $\Gamma \vdash e : \alpha$, *then the list* Γ *is a valid context, that is, we can derive* Γ **Context**.

Proof. The proof is by straightforward induction on the derivation of $\Gamma \vdash e : \alpha$. Remember that if a list $[\Gamma\ ,\ x\ :\alpha\ ]$ is a context, then so is the list Γ.

■

Now, we can formalize contexts and the type system directly using these modifications.

Notice that after the modifications, we define the predicates **fresh** and **Context** over lists of declarations. Furthermore, the definitions of the predicates **fresh** and **Context** are no more mutually recursive. Finally, notice that since we work with lists of declarations we do not have to check whether or not the lists Γ and Δ are disjoint to define the concatenation $\Gamma\ \&\ \Delta$. The list $\Gamma\ \&\ \Delta$ is a valid context if $\Gamma\ \&\ \Delta$ **Context** holds.

3.1.3 Formalization of the Substitution Lemma

In section 2.4 we formulated the *Substitution Lemma* as follows :

Let $[\Gamma\ ,\ x\ :\alpha\ ]$ *and* Δ *be two disjoint contexts,* d *and* e *expressions,* x *a variable and* α *and* β *types. If we can derive* $[\Gamma\ ,\ x\ :\alpha\ ]\ \&\ \Delta \vdash e : \beta$ *and* $\Gamma \vdash d : \alpha$, *then we can also derive* $\Gamma\ \&\ \Delta \vdash e\,[d/x] : \beta$.

After the modifications of the previous section $[\Gamma\ ,\ x\ :\alpha\ ]$ and Δ are lists of declarations. So, the requirement that the two contexts are disjoint is irrelevant.

Our informal proof is by induction on the derivation of $[\Gamma\ ,\ x\ :\alpha\ ]\ \&\ \Delta \vdash e : \beta$. The problem now is the formalization of this induction in ALF.

$\vdash : \mathit{ListDecl} \to \mathit{Exp} \to \mathit{Type} \to \mathit{Set}$

$x \quad : \mathit{Var}$
$d\,,\,e\,,\,f \;: \mathit{Exp}$
$\alpha\,,\,\beta \quad : \mathit{Type}$
$\Gamma \quad : \mathit{ListDecl}$

TSVar Rule

$$\frac{\Gamma\ \textbf{Context} \qquad x\ \textbf{fresh}\ \Gamma}{[\Gamma\,,\,x\ :\alpha\ ]\ \vdash x\ :\alpha}$$

TSThinn Rule

$$\frac{\Gamma\ \vdash e\ :\alpha \qquad x\ \textbf{fresh}\ \Gamma}{[\Gamma\,,\,x\ :\beta\ ]\ \vdash e\ :\alpha}$$

TSAbs Rule

$$\frac{[\Gamma\,,\,x\ :\alpha\ ]\ \vdash e\ :\beta}{\Gamma\ \vdash \lambda\, x\,.\,e\ :\alpha\ \to\ \beta}$$

TSApp Rule

$$\frac{\Gamma\ \vdash e\ :\alpha\ \to\ \beta \qquad \Gamma\ \vdash f\ :\alpha}{\Gamma\ \vdash (e\ \ f)\ :\beta}$$

TSFix Rule

$$\frac{[\Gamma\,,\,x\ :\alpha\ ]\ \vdash e\ :\alpha}{\Gamma\ \vdash \textbf{fix}\ x\,.\,e\ :\alpha}$$

TSTrue Rule

$$\frac{\Gamma\ \textbf{Context}}{\Gamma\ \vdash \mathit{true}\ :\textbf{Bool}}$$

TSFalse Rule

$$\frac{\Gamma\ \textbf{Context}}{\Gamma\ \vdash \mathit{false}\ :\textbf{Bool}}$$

TSIf Rule

$$\frac{\Gamma\ \vdash d\ :\textbf{Bool} \qquad \Gamma\ \vdash e\ :\alpha \qquad \Gamma\ \vdash f\ :\alpha}{\Gamma\ \vdash \textbf{if}\ d\ \textbf{then}\ e\ \textbf{else}\ f\ :\alpha}$$

Figure 8: Modified Inductive Definition of the Type System for the Language.

Given the definition of the type system, we have a natural induction principle on derivations of judgements of the form $\Gamma \vdash e : \alpha$, for an arbitrary expression e, an arbitrary type α and an arbitrary context Γ; that is, not necessarily of the form $\Gamma' \ \& \ \Delta$. In the informal proof, we implicitly inferred the cases of the derivations of $[\Gamma , x : \alpha] \ \& \ \Delta \vdash e : \beta$ from the definition of the type system.

If we formulate the theorem in this way in ALF and we try to perform the induction mentioned before, we get the following message :

> *Non trivial unification problem*
> $[\Gamma' , y : \beta] \equiv [\Gamma , x : \alpha] \ \& \ \Delta$

This happens because ALF tries to unify $[\Gamma , x : \alpha] \ \& \ \Delta \vdash e : \beta$ with the conclusion of each rule in the type system. In the cases of the rules **TSVar** and **TSThinn** the conclusion is of the form $[\Gamma' , y : \beta] \vdash e' : \gamma$ (in the rule **TSVar** $y \equiv e'$ and $\beta \equiv \alpha$). So ALF has to unify $[\Gamma , x : \alpha] \ \& \ \Delta$ with $[\Gamma' , y : \beta] , e$ with e' and α with γ. In the last two cases the unification is easy, but in the first one the unification is non trivial. For unifying the lists ALF needs to know whether the list Δ is empty or inhabited, because we defined the concatenation function by induction in the second argument. In this case the list Δ can be any kind of list, so ALF has no information about whether it is empty or not; so it cannot unify.

There are at least two possible solutions for the problem :

- The first solution is to formalize the lemma as presented before. When constructing the proof, before making the induction we consider the cases for the list Δ, that is, whether it is empty or inhabited. This gives us two equations in the construction of the proof of the lemma. In each equation we make the desired induction.

  This process gives us two equations for each rule in the type system. For each rule we discuss the lemma considering the cases whether or not the list Δ is empty. However, the informal presentation of the proof presented in section 2.4 tells us that we only need this distinction in the cases of the rules **TSVar** and **TSThinn**. So, in a way, this solution implies more work than necessary.

- The second solution is to reformulate the theorem. We can formulate the *Substitution Lemma* as follows :

  Let Σ, Γ *and* Δ *be lists of declarations,* d *and* e *expressions,* x *a variable and* α *and* β *types. If we can derive* $\Sigma \vdash e : \beta$ *and* $\Gamma \vdash d : \alpha$ *when* $\Sigma = [\Gamma , x : \alpha] \ \& \ \Delta$ *, then we can also derive* $\Gamma \ \& \ \Delta \vdash e\,[d/x] : \beta$.

  It is easy to see that both formulations of the lemma are equivalent. But now we can perform the induction on the definition of $\Sigma \vdash e : \beta$ without problems. This gives us one equation for each rule in the type system. With this solution, like in the informal

presentation of the proof, we only have to consider cases for the list Δ in the rules **TSVar** and **TSThinn**. We can now analyze (in the equations where this is needed) the proof that requires $\Sigma = [\Gamma , x : \alpha] \ \& \ \Delta$. The only proof of a proposition of the form $a = b$ (with a and b elements of a set S) is $id(a)$. So, this analysis gives us the list Σ that makes the proposition true. Of course, the form of Σ depends on the form of the list $[\Gamma , x : \alpha]$ and on the form of the list Δ .

Notice that in spite of the fact that we use an extra list Σ and the requirement that Σ is equal to $[\Gamma , x : \alpha] \ \& \ \Delta$, the complete proof with this solution has the same shape as the proof presented in section 2.4.

In our formalization in ALF of all of the results, we formulate the *Substitution Lemma* as in the second solution presented above.

3.2 Some Conclusions About ALF

The interactive proof editor ALF has changed a lot since its first version in 1990. The possibility of working with *pattern matching* instead of with the *elimination rules* for the sets defined makes the programs easier to write and to read, although it is known that these two disciplines are not proof-theoretically equivalent. However, in our opinion there are still two important improvements to make.

The first improvement is related to the inductive proofs. When making proofs by induction, ALF checks the types but it does not check whether the induction is well founded or not. This condition is easy to check manually when the theory we are working with is small and there is just one induction involved in the proof. But, if the theory is big and we are constructing a proof that involves more than one induction, the control of whether or not these inductions are well founded is not so trivial.

The second improvement is related to extending the theory. Frequently, when we want to prove some results for a language, we start studying just a small part of the language and after having proved the desired results for that small part we study the rest of it. A clear example of this is the theory presented here. We started with a small language and after making the proof of the *Subject Reduction* property for this language, we would like to extend it with for example the **natural numbers** and the **let** expression. And, very likely, after that we would like to extend it again with new expressions. But each time we want to extend the language in ALF instead of completing the proofs for the new cases, we have to define the language again and to redo the proofs from the beginning.

4 Conclusions

4.1 Related Work

The Subject Reduction property for languages similar to ours is studied in several papers. However, hardly any of these treatments is completely formal.

In [Bar 92], Barendregt studies the Subject Reduction property for the λ_calculus where the dynamic semantics is given by the β_reduction rules. In the proofs Barendregt relies on what he calls *variable convention* that lets him deal with variables. This convention says that bound and free variables are chosen such that they differ from each other. This allows him to prove a thinning rule needed for the Substitution Lemma, as a derived rule. Without this convention it is not possible to prove the thinning rule and hence the Substitution Lemma.

Of course this variable convention is not formal. We think that there are (at least) two ways to formalize it. One way is taking bound and free variables as two syntactically different categories, as is done by Pollack in his PhD thesis ([Pol 94], see discussion below). The other way is defining a predicate over expressions that tells us whether or not the variables that occur in an expression are such that the convention mentioned above holds. We believe that it is quite hard to work with this kind of predicate (if not impossible).

Holmström also proves the Subject Reduction property for a language of expressions similar to ours (see [Hol 83]). In some of his proofs (as for example in the proof of the Substitution Lemma), he shows that the conclusion of a theorem holds by informally manipulating the derivations in the type system.

In his PhD thesis, Pollack studies and formalizes the Subject Reduction property for Pure Type Systems (PTS) in the proof checker LEGO. This language is more complex than ours because of the dependent types. In his thesis he distinguishes between bound variables that he calls *variables* and free variables that he calls *parameters*. Parameters and variables are disjoint sets. Having these two sets leads him to define two substitutions : one replacing an expression for a parameter in an expression and other replacing an expression for a variable in an expression. The idea is to replace the usual rule for typing an abstraction by the following rule :

$$\frac{[\Gamma\ ,\ p\ :\alpha\ ]\ \vdash\ e\,[p/x]\ :\beta}{\Gamma\ \vdash\ \lambda\, x\,.\,e\ :\alpha\ \rightarrow\ \beta}$$

where p is a completely fresh parameter. In this way he captures the essence of α_conversion where the name of bound variables does not matter, and abstracting a variable more than once in an expression is no more a problem here (see the discussion at the end of section 2.3.3). Pollack also present this method in his paper [Pol 93] where he discusses how to deal with α_conversion in the λ_calculus.

Another difference between our presentation and Pollack's is the validity of contexts. As an optimization he takes the validity of contexts out of the type system and each time he wants to prove a result he adds an extra assumption to the theorem requiring the context to be valid. Instead, we prefer to leave this validity condition as part of the system. Although leaving the condition as part of the system implies that it is tested more often, this also ensures that

the type system is closed in the sense that we do not need to add extra requirements in theorems.

In [Mic 91] there is a formalization of the dynamic semantics and the type system of Mini-ML in the logic programming language Elf ([Pfe 91]) which is founded upon the logical framework LF ([Har 93]). This language is bigger than ours, but it is not more complex. Although the informal presentation of the type system presented in the paper is similar to ours, the Elf formalization is quite different to the ALF formalization we present here. The formal counterpart of a set of Martin-Löf's type theory is in LF called a type. Thus, for instance, our set *Exp* would be declared as a type. But, unlike Martin-Löf's type theory's sets, LF's types are not inductively defined. This allows the use of a so-called "higher-order abstract syntax" for coding expressions into LF. For instance, the ML abstraction is formalized as a function with type $(exp \to exp) \to exp$ where exp is the type that represents expressions of ML. So variable binding in ML is represented with the help of the λ_abstraction in Elf and then substitutions in ML are implemented using the β_reduction of Elf which avoids explicit α_conversion to prevent capturing bound variables. Moreover, the formalization of the type system contexts in the informal presentation become contexts in the meta-language Elf, so there is no need to formalize the notion of contexts. Notice that if we formalize the abstractions in this way we obtain more expressions than expected. One advantage of this formalization is that, because of the features of Elf, this formalization can be used as an interpreter for ML. However, because types are not inductively defined in Elf, there is no way of formalizing properties such as for example the *Subject Reduction*. Instead, the paper present a set of rules that describe the relation between the assumptions and the conclusion of the theorem. This set of rules is called in the paper a *partial internalization* of the proof of the property.

4.2 Further Work

A natural extension of this work is considering the addition of *type schemes* to the language of types. Then, we can show that if an expression e has type under a context then e has a *most general type scheme* from which all types we can derive for the expression are *instances*. The proof of such a theorem in type theory gives us the *type inference* algorithm. That is, the proof of the theorem is a method that infers the most general type scheme for an expression, provided that the expression has a type.

Another extension is adding to the language expressions of the form **let** $x = d$ **in** e. We can type a **let** expression in the same way as an application or we can do it in a polymorphic way. The difference between the expressions $(\lambda x\,.\,e\ \ \mathrm{d})$ and **let** $x = d$ **in** e is explained very clearly in [Hol 83]. There are two ways of typing the **let** expression in a polymorphic way. In [Cle 86] and [Dam 82] we find a type system that uses *type schemes* to type the **let** expressions. There is another method that involves the use of substitutions in the type system (see [Hol 83]).

There are several type systems for a language like the one we presented here. We chose just one and discussed the results for that type system. However, we are not sure that the system we chose is the most convenient one for the formal treatment we are interested in. The study of other systems has also to include the verification of whether they are equally expressive in the sense that an expression e has type α in one system if and only if e has type α in the other system.

There is a well know notion introduced by Milner in [Mil 78] that we did not study here. This is the notion of *well typed programs cannot go wrong*, that can be formalized with the *Computation Semantics* (see [Mil 78, Hol 83]). Instead of defining the overall evaluation relation $\_ \Rightarrow \_$, the computation semantics (see [Hen 90]) defines a one step relation $\_ \rightarrow \_$. The relation $\_ \Rightarrow \_$ can be seen as the *reflexive transitive closure* of $\_ \rightarrow \_$.

Finally, there are other properties of a compilers we want to study. For example, it will be interesting to show that when evaluating an expression of the language, the dynamic semantics does not introduce variables. To prove this, we define a relation $\Lambda \rhd e$ meaning that the expression e depends on the list of variables Λ. Furthermore, an expression is *closed* if it depends on the empty list of variables. Then we can show that if $\rightsquigarrow$ is the dynamic semantics, d and e are two expressions such that $d \rightsquigarrow e$ and Λ is a list of variables, $\Lambda \rhd d$ implies $\Lambda \rhd e$. In particular, if $d \rightsquigarrow e$ for a closed expression d , then the expression e is also closed.

Acknowledgements

First, I want to thank the Computer Science Department at the Engeneering School, Uruguay, where I work and the Department of Computer Sciences at the Chalmers University of Technology, Sweden, where I spent more than half a year as a guest, for providing a friendly working environment.

I want to thank Bengt Nordström for his help in arranging my stay in Sweden.

I especially want to thank my supervisor Alvaro Tasistro which whom I spent many hours discussing the results of this work and who corrected the different versions of the thesis, and who also encouraged me each time things went wrong.

Several people spent part of their time discussing ideas about this work or reading draft versions of this thesis. I want to thank all of them for their effort and helpful comments. Especially, I want to thank Gustavo Betarte, Verónica Gaspes, Johan Jeuring, Randy Pollack and Björn von Sydow.

Finally, I want to thank my family and friends for all the support they gave to me in these years.

A ALF Code

In this section we present the ALF code of the definitions and theorems described in this paper. In the code we use the formalization of Martin-Löf's monomorphic set theory provided by the ALF library.

In the following code we can find three forms of definitions :

- $c : \alpha \ [\Gamma\,]\ C$, where the letter C at the end of the definition indicates that c is introduced as a constructor.
- $c : \alpha \ [\Gamma\,]\ I$, where the letter I at the end of the definition indicates that c is an implicit constant defined using pattern matching or case analysis over (some of) its arguments. Immediately after such a definition, the equations associated to each case of the pattern or the corresponding case expressions are displayed.
- $c = e : \alpha \ [\Gamma\,]$, where this case indicates that c is just an abbreviation (explicit constant).

In each case $[\Gamma\,]$ stands for the set of assumptions of the definitions. In the following code, it is always the case that $[\Gamma\,]$ is empty ([]).

To make the code more readable, we introduce some abbreviations in the code of the *Substitution Lemma*. Each abbreviation has the mark (*) on its left. The codes of all the abbreviations are displayed immediately after the lemma.

A.1 Definition of the Set of Variables

```
VAR : Set                                                    [] C

  var : (n:N)VAR                                             [] C



Value : (v:VAR)N                                             [] I

  Value(var(n)) = n



EqualToAbsurdity1 : (n:N; i:Id(N,0,s(n)))Absurdity           [] I

  EqualToAbsurdity1(n,i) = case i : Id(N,0,s(n)) of
                           end
```

```
EqualToAbsurdity2 : (n:N; i:Id(N,s(n),0))Absurdity                          [] I

  EqualToAbsurdity2(n,i) = case i : Id(N,s(n),0) of
                           end



EqAndNotEq1ToAbsurdity : (n:N; m:N;
                          ni:Not(Id(N,n,m));
                          i:Id(N,s(n),s(m))
                         )Absurdity                                          [] I

  EqAndNotEq1ToAbsurdity(n,m,ni,i) =
    case ni : Not(Id(N,n,m)) of
      Imply_intro(_,_,f) => case f(idcongr(N,N,pred,s(n),s(m),i)) : Absurdity of
                            end
    end



EqAndNotEq2ToAbsurdity : (n:N; m:N;
                          ni:Not(Id(N,n,m));
                          i:Id(VAR,var(n),var(m))
                         )Absurdity                                          [] I

  EqAndNotEq2ToAbsurdity(n,m,ni,i) =
    case ni : Not(Id(N,n,m)) of
      Imply_intro(_,_,f) => case f(idcongr(VAR,N,Value,var(n),var(m),i)) :
                                                                  Absurdity of
                            end
    end
```

```
FromNatEq : (n:N; m:N)Or(Id(N,n,m),Not(Id(N,n,m)))                    [] I

  FromNatEq(0,0)         = Or_intro1(Id(N,0,0),Not(Id(N,0,0)),id(N,0))
  FromNatEq(0,s(m1))     =
    Or_intro2(Id(N,0,s(m1)),
              Not(Id(N,0,s(m1))),
              Imply_intro(Id(N,0,s(m1)),
                          Absurdity,
                          [i]EqualToAbsurdity1(m1,i)))
  FromNatEq(s(n1),0)     =
    Or_intro2(Id(N,s(n1),0),
              Not(Id(N,s(n1),0)),
              Imply_intro(Id(N,s(n1),0),
                          Absurdity,
                          [i]EqualToAbsurdity2(n1,i)))
  FromNatEq(s(n1),s(m1)) =
    case FromNatEq(n1,m1) : Or(Id(N,n1,m1),Not(Id(N,n1,m1))) of
      Or_intro1(_,_,i)  => Or_intro1(Id(N,s(n1),s(m1)),
                                     Not(Id(N,s(n1),s(m1))),
                                     idcongr(N,N,s,n1,m1,i))
      Or_intro2(_,_,ni) =>
        Or_intro2(Id(N,s(n1),s(m1)),
                  Imply(Id(N,s(n1),s(m1)),Absurdity),
                  Imply_intro(Id(N,s(n1),s(m1)),
                              Absurdity,
                              [i]EqAndNotEq1ToAbsurdity(n1,m1,ni,i)))
    end


IdVarDec : (x:VAR; y:VAR)Or(Id(VAR,x,y),Not(Id(VAR,x,y)))              [] I

  IdVarDec(var(n),var(m)) =
    case FromNatEq(n,m) : Or(Id(N,n,m),Not(Id(N,n,m))) of
      Or_intro1(_,_,i)  => Or_intro1(Id(VAR,var(n),var(m)),
                                     Not(Id(VAR,var(n),var(m))),
                                     idcongr(N,VAR,var,n,m,i))
      Or_intro2(_,_,ni) =>
        Or_intro2(Id(VAR,var(n),var(m)),
                  Not(Id(VAR,var(n),var(m))),
                  Imply_intro(Id(VAR,var(n),var(m)),
                              Absurdity,
                              [i]EqAndNotEq2ToAbsurdity(n,m,ni,i)))
    end
```

A.2 Definition of the Set of Expressions

```
Exp : Set                                                          [] C

  Var   : (x:VAR)Exp                                               [] C
  Abs   : (x:VAR; e:Exp)Exp                                        [] C
  App   : (d:Exp; e:Exp)Exp                                        [] C
  Fix   : (x:VAR; e:Exp)Exp                                        [] C
  True  : Exp                                                      [] C
  False : Exp                                                      [] C
  If    : (d:Exp; e:Exp; f:Exp)Exp                                 [] C



Subst : (x:VAR; d:Exp; e:Exp)Exp                                   [] I

  Subst(x,d,Var(y))    = Or_elim(Id(VAR,x,y),
                                 Not(Id(VAR,x,y)),
                                 [e]Exp,
                                 [i]d,
                                 [ni]Var(y),
                                 IdVarDec(x,y))
  Subst(x,d,Abs(y,e))  = Or_elim(Id(VAR,x,y),
                                 Not(Id(VAR,x,y)),
                                 [e']Exp,
                                 [i]Abs(y,e),
                                 [ni]Abs(y,Subst(x,d,e)),
                                 IdVarDec(x,y))
  Subst(x,d,App(e,f))  = App(Subst(x,d,e),Subst(x,d,f))
  Subst(x,d,Fix(y,e))  = Or_elim(Id(VAR,x,y),
                                 Not(Id(VAR,x,y)),
                                 [e']Exp,
                                 [i]Fix(y,e),
                                 [ni]Fix(y,Subst(x,d,e)),
                                 IdVarDec(x,y))
  Subst(x,d,True)      = True
  Subst(x,d,False)     = False
  Subst(x,d,If(e,f,g)) = If(Subst(x,d,e),Subst(x,d,f),Subst(x,d,g))
```

A.3 Definition of Canonical Expressions

```
Canon : (Exp)Set                                                    [] C

  CanonTrue  : Canon(True)                                          [] C
  CanonFalse : Canon(False)                                         [] C
  CanonAbs   : (x:VAR; e:Exp)Canon(Abs(x,e))                        [] C
```

A.4 Definition of the Dynamic Semantics

```
DynSem : (d:Exp; e:Exp)Set                                          [] C

   DSAbs   : (x:VAR; e:Exp)DynSem(Abs(x,e),Abs(x,e))                [] C
   DSApp   : (x:VAR;
              d:Exp; f:Exp; e:Exp; g:Exp;
              p:DynSem(f,Abs(x,d));
              p1:DynSem(Subst(x,e,d),g)
             )DynSem(App(f,e),g)                                    [] C
   DSFix   : (x:VAR;
              e:Exp; f:Exp;
              p:DynSem(Subst(x,Fix(x,e),e),f)
             )DynSem(Fix(x,e),f)                                    [] C
   DSTrue  : DynSem(True,True)                                      [] C
   DSFalse : DynSem(False,False)                                    [] C
   DSIfT   : (d:Exp; e:Exp; f:Exp; g:Exp;
              p:DynSem(d,True);
              p1:DynSem(e,g)
             )DynSem(If(d,e,f),g)                                   [] C
   DSIfF   : (d:Exp; e:Exp; f:Exp; g:Exp;
              p:DynSem(d,False);
              p1:DynSem(f,g)
             )DynSem(If(d,e,f),g)                                   [] C
```

A.5 Formalization of the Proofs Related with the Dynamic Semantics

```
CanonEvaluation : (d:Exp; e:Exp; p:DynSem(d,e))Canon(e)                [] I

  CanonEvaluation(_,_,DSAbs(x,e1))              = CanonAbs(x,e1)
  CanonEvaluation(_,e,DSApp(x,d1,f,e1,_,p1,p2)) =
    CanonEvaluation(Subst(x,e1,d1),e,p2)
  CanonEvaluation(_,e,DSFix(x,e1,_,p1))         =
    CanonEvaluation(Subst(x,Fix(x,e1),e1),e,p1)
  CanonEvaluation(_,_,DSTrue)                   = CanonTrue
  CanonEvaluation(_,_,DSFalse)                  = CanonFalse
  CanonEvaluation(_,e,DSIfT(d1,e1,f,_,p1,p2))   = CanonEvaluation(e1,e,p2)
  CanonEvaluation(_,e,DSIfF(d1,e1,f,_,p1,p2))   = CanonEvaluation(f,e,p2)



Unicity : (d:Exp; e:Exp; e1:Exp;
           p:DynSem(d,e); p1:DynSem(d,e1)
          )Id(Exp,e,e1)                                                [] I

  Unicity(_,_,_,DSAbs(x,e2),DSAbs(_,_))                          =
    id(Exp,Abs(x,e2))
  Unicity(_,e,e1,DSApp(x,d1,f,e2,_,p2,p3),DSApp(x1,d,_,_,_,p,p4)) =
    case Unicity(f,Abs(x,d1),Abs(x1,d),p2,p) : Id(Exp,Abs(x,d1),Abs(x1,d)) of
      id(_,_) => case Unicity(Subst(x1,e2,d),e,e1,p3,p4) : Id(Exp,e,e1) of
                   id(_,_) => id(Exp,e1)
                 end
    end
  Unicity(_,e,e1,DSFix(x,e2,_,p2),DSFix(_,_,_,p))                =
    Unicity(Subst(x,Fix(x,e2),e2),e,e1,p2,p)
  Unicity(_,_,_,DSTrue,DSTrue)                                   =
    id(Exp,True)
  Unicity(_,_,_,DSFalse,DSFalse)                                 =
    id(Exp,False)
  Unicity(_,e,e1,DSIfT(d1,e2,f,_,p2,p3),DSIfT(_,_,_,_,p,p4))     =
    Unicity(e2,e,e1,p3,p4)
  Unicity(_,e,e1,DSIfT(d1,e2,f,_,p2,p3),DSIfF(_,_,_,_,p,p4))     =
    case Unicity(d1,True,False,p2,p) : Id(Exp,True,False) of
    end
  Unicity(_,e,e1,DSIfF(d1,e2,f,_,p2,p3),DSIfT(_,_,_,_,p,p4))     =
    case Unicity(d1,True,False,p,p2) : Id(Exp,True,False) of
    end
  Unicity(_,e,e1,DSIfF(d1,e2,f,_,p2,p3),DSIfF(_,_,_,_,p,p4))     =
    Unicity(f,e,e1,p3,p4)
```

A.6 Definition of the Set of Types

```
Type : Set                                                    [] C

  TBool : Type                                                [] C
  TFunc : (a:Type; b:Type)Type                                [] C
```

A.7 Definition of the Set of Declarations

```
Decl : Set                                                    [] C

  decl : (x:VAR; a:Type)Decl                                  [] C


VarDecl : (dl:Decl)VAR                                        [] I

  VarDecl(decl(x,a)) = x


TypeDecl : (dl:Decl)Type                                      [] I

  TypeDecl(decl(x,a)) = a
```

A.8 Definition of the Set List of Declarations

```
ListDecl = List(Decl) : Set                                   []


Concat : (L:ListDecl; L1:ListDecl)ListDecl                    [] I

  Concat(L,nil(_))       = L
  Concat(L,cons(_,dl,l)) = cons(Decl,dl,Concat(L,l))
```

```
Fresh : (x:VAR; L:ListDecl)Set                                          [] C

  FreshEmpty : (x:VAR)Fresh(x,nil(Decl))                                [] C
  FreshCons  : (x:VAR;
                dl:Decl;
                L:ListDecl;
                fs:Fresh(x,L);
                ni:Not(Id(VAR,x,VarDecl(dl)))
               )Fresh(x,cons(Decl,dl,L))                                [] C


FreshMon : (dl:Decl;
            G:ListDecl;
            x:VAR;
            fs:Fresh(x,cons(Decl,dl,G))
           )Fresh(x,G)                                                  [] I

  FreshMon(dl,G,x,FreshCons(_,_,_,fs1,ni)) = fs1


ConcatFreshMon : (dl:Decl;
                  G:ListDecl; D:ListDecl;
                  x:VAR;
                  fs:Fresh(x,Concat(cons(Decl,dl,G),D))
                 )Fresh(x,Concat(G,D))                                  [] I

  ConcatFreshMon(dl,G,nil(_),x,FreshCons(_,_,_,fs1,ni))      = fs1
  ConcatFreshMon(dl,G,cons(_,a,l),x,FreshCons(_,_,_,fs1,ni)) =
    FreshCons(x,a,Concat(G,l),ConcatFreshMon(dl,G,l,x,fs1),ni)
```

A.9 Definition of the Predicate Context

```
Context : (L:ListDecl)Set                                               [] C

  ContextEmpty : Context(nil(Decl))                                     [] C
  ContextCons  : (dl:Decl;
                  L:ListDecl;
                  c:Context(L);
                  fs:Fresh(VarDecl(dl),L)
                 )Context(cons(Decl,dl,L))                              [] C
```

```
ContextMon : (dl:Decl; G:ListDecl; c:Context(cons(Decl,dl,G)))Context(G) [] I

  ContextMon(dl,G,ContextCons(_,_,c1,fs)) = c1



ConcatCtxtMon : (dl:Decl;
                 G:ListDecl; D:ListDecl;
                 c:Context(Concat(cons(Decl,dl,G),D))
                )Context(Concat(G,D))                                    [] I

  ConcatCtxtMon(dl,G,nil(_),c)                            =
    ContextMon(dl,Concat(G,nil(Decl)),c)
  ConcatCtxtMon(dl,G,cons(_,a,l),ContextCons(_,_,c1,fs)) =
    ContextCons(a,
                Concat(G,l),
                ConcatCtxtMon(dl,G,l,c1),
                ConcatFreshMon(dl,G,l,VarDecl(a),fs))



FreshCtxtToAbsurd : (dl:Decl;
                     G:ListDecl; D:ListDecl;
                     c:Context(Concat(cons(Decl,dl,G),D));
                     fs:Fresh(VarDecl(dl),Concat(cons(Decl,dl,G),D))
                    )Absurdity                                           [] I

  FreshCtxtToAbsurd(dl,G,nil(_),c,FreshCons(_,_,_,fs1,ni)) =
    case ni : Not(Id(VAR,VarDecl(dl),VarDecl(dl))) of
      Imply_intro(_,_,f) => case f(id(VAR,VarDecl(dl))) : Absurdity of
                            end
    end
  FreshCtxtToAbsurd(dl,G,cons(_,dl1,l),c,fs)              =
    FreshCtxtToAbsurd(dl,
                      G,
                      l,
                      ContextMon(dl1,Concat(cons(Decl,dl,G),l),c),
                      FreshMon(dl1,Concat(cons(Decl,dl,G),l),VarDecl(dl),fs))
```

```
CtxtAndIdToAbsurdity : (x:VAR; x1:VAR;
                        a:Type; b:Type;
                        G:ListDecl; D:ListDecl;
                        i:Id(VAR,x,x1);
                        c:Context(cons(Decl,
                                       decl(x1,a),
                                       Concat(cons(Decl,decl(x,b),G),D)))
                       )Absurdity                                          [] I

  CtxtAndIdToAbsurdity(x,x1,a,b,G,D,i,ContextCons(_,_,c1,fs)) =
    FreshCtxtToAbsurd(decl(x,b),
                      G,
                      D,
                      c1,
                      idsubst'(VAR,
                               [y]Fresh(y,Concat(cons(Decl,decl(x,b),G),D)),
                               x,
                               x1,
                               i,
                               fs))
```

A.10 Definition of the Type System

```
TypeSystem : (G:ListDecl; e:Exp; a:Type)Set                                    [] C

   TSVar   : (dl:Decl;
              G:ListDecl;
              c:Context(G);
              fs:Fresh(VarDecl(dl),G)
             )TypeSystem(cons(Decl,dl,G),Var(VarDecl(dl)),TypeDecl(dl)) [] C
   TSThinn : (G:ListDecl;
              e:Exp;
              a:Type;
              dl:Decl;
              p:TypeSystem(G,e,a);
              fs:Fresh(VarDecl(dl),G)
             )TypeSystem(cons(Decl,dl,G),e,a)                                  [] C
   TSAbs   : (G:ListDecl;
              x:VAR;
              e:Exp;
              a:Type;b:Type;
              p:TypeSystem(cons(Decl,decl(x,a),G),e,b)
             )TypeSystem(G,Abs(x,e),TFunc(a,b))                                [] C
   TSApp   : (G:ListDecl;
              d:Exp; e:Exp;
              a:Type; b:Type;
              p:TypeSystem(G,d,TFunc(a,b));
              p1:TypeSystem(G,e,a)
             )TypeSystem(G,App(d,e),b)                                         [] C
   TSFix   : (G:ListDecl;
              x:VAR;
              e:Exp;
              a:Type;
              p:TypeSystem(cons(Decl,decl(x,a),G),e,a)
             )TypeSystem(G,Fix(x,e),a)                                         [] C
   TSTrue  : (G:ListDecl; c:Context(G))TypeSystem(G,True,TBool)               [] C
   TSFalse : (G:ListDecl; c:Context(G))TypeSystem(G,False,TBool)              [] C
   TSIf    : (G:ListDecl;
              d:Exp; e:Exp; f:Exp; a:Type;
              p:TypeSystem(G,d,TBool);
              p1:TypeSystem(G,e,a);
              p2:TypeSystem(G,f,a)
             )TypeSystem(G,If(d,e,f),a)                                        [] C
```

A.11 Formalization of the Proofs Related to the Type System

```
ValidCtxt : (G:ListDecl; e:Exp; a:Type; p:TypeSystem(G,e,a))Context(G)    [] I

  ValidCtxt(_,_,_,TSVar(dl,G1,c,fs))         = ContextCons(dl,G1,c,fs)
  ValidCtxt(_,e,a,TSThinn(G1,_,_,dl,p1,fs))  =
    ContextCons(dl,G1,ValidCtxt(G1,e,a,p1),fs)
  ValidCtxt(G,_,_,TSAbs(_,x,e1,a1,b,p1))     =
    ContextMon(decl(x,a1),G,ValidCtxt(cons(Decl,decl(x,a1),G),e1,b,p1))
  ValidCtxt(G,_,a,TSApp(_,d,e1,a1,_,p1,p2))  = ValidCtxt(G,e1,a1,p2)
  ValidCtxt(G,_,a,TSFix(_,x,e1,_,p1))        =
    ContextMon(decl(x,a),G,ValidCtxt(cons(Decl,decl(x,a),G),e1,a,p1))
  ValidCtxt(G,_,_,TSTrue(_,c))               = c
  ValidCtxt(G,_,_,TSFalse(_,c))              = c
  ValidCtxt(G,_,a,TSIf(_,d,e1,f,_,p1,p2,p3)) = ValidCtxt(G,d,TBool,p1)



EqAndNotEqToId : (x:VAR; y:VAR;
                  i:Id(VAR,x,y);
                  ni:Not(Id(VAR,x,y));
                  d:Exp
                 )Id(Exp,Var(y),d)                                        [] I

  EqAndNotEqToId(x,y,i,Imply_intro(_,_,b),d) = case b(i) : Absurdity of
                                               end



VarNotFreeInExp : (G:ListDecl;
                   x:VAR;
                   d:Exp; e:Exp;
                   a:Type;
                   fs:Fresh(x,G);
                   p:TypeSystem(G,e,a)
                  )Id(Exp,e,Subst(x,d,e))                                 [] I
```

```
VarNotFreeInExp(_,x,d,_,_,FreshCons(_,_,_,fs2,ni),TSVar(dl,G1,c,fs1)) =
  Or_elim(Id(VAR,x,VarDecl(dl)),
          Not(Id(VAR,x,VarDecl(dl))),
          [o]Id(Exp,
                Var(VarDecl(dl)),
                Or_elim(Id(VAR,x,VarDecl(dl)),
                        Not(Id(VAR,x,VarDecl(dl))),
                        [o']Exp,
                        [i]d,
                        [ni']Var(VarDecl(dl)),
                        o)),
          [i]EqAndNotEqToId(x,VarDecl(dl),i,ni,d),
          [ni']id(Exp,Var(VarDecl(dl))),
          IdVarDec(x,VarDecl(dl)))
VarNotFreeInExp(_,x,d,e,a,fs,TSThinn(G1,_,_,dl,p1,fs1))                =
  VarNotFreeInExp(G1,x,d,e,a,FreshMon(dl,G1,x,fs),p1)
VarNotFreeInExp(G,x,d,_,_,fs,TSAbs(_,x1,e1,a1,b,p1))                   =
  Or_elim(Id(VAR,x,x1),
          Not(Id(VAR,x,VarDecl(decl(x1,a1)))),
          [o]Id(Exp,
                Abs(x1,e1),
                Or_elim(Id(VAR,x,x1),
                        Not(Id(VAR,x,VarDecl(decl(x1,a1)))),
                        [o']Exp,
                        [i]Abs(x1,e1),
                        [ni]Abs(x1,Subst(x,d,e1)),
                        o)),
          [i]id(Exp,Abs(x1,e1)),
          [ni]idsubst(Exp,
                      [e']Id(Exp,Abs(x1,e1),Abs(x1,e')),
                      e1,
                      Subst(x,d,e1),
                      VarNotFreeInExp(cons(Decl,decl(x1,a1),G),
                                      x,
                                      d,
                                      e1,
                                      b,
                                      FreshCons(x,decl(x1,a1),G,fs,ni),
                                      p1),
                      id(Exp,Abs(x1,e1))),
          IdVarDec(x,x1))
```

```
VarNotFreeInExp(G,x,d,_,a,fs,TSApp(_,d1,e1,a1,_,p1,p2)) =
  idsubst(Exp,
          [d']Id(Exp,App(d1,e1),App(d',Subst(x,d,e1))),
          d1,
          Subst(x,d,d1),
          VarNotFreeInExp(G,x,d,d1,TFunc(a1,a),fs,p1),
          idsubst(Exp,
                  [e']Id(Exp,App(d1,e1),App(d1,e')),
                  e1,
                  Subst(x,d,e1),
                  VarNotFreeInExp(G,x,d,e1,a1,fs,p2),
                  id(Exp,App(d1,e1))))
VarNotFreeInExp(G,x,d,_,a,fs,TSFix(_,x1,e1,_,p1))       =
  Or_elim(Id(VAR,x,x1),
          Not(Id(VAR,x,VarDecl(decl(x1,a)))),
          [o]Id(Exp,
                Fix(x1,e1),
                Or_elim(Id(VAR,x,x1),
                        Not(Id(VAR,x,VarDecl(decl(x1,a)))),
                        [o']Exp,
                        [i]Fix(x1,e1),
                        [ni]Fix(x1,Subst(x,d,e1)),
                        o)),
          [i]id(Exp,Fix(x1,e1)),
          [ni]idsubst(Exp,
                      [e']Id(Exp,Fix(x1,e1),Fix(x1,e')),
                      e1,
                      Subst(x,d,e1),
                      VarNotFreeInExp(cons(Decl,decl(x1,a),G),
                                      x,
                                      d,
                                      e1,
                                      a,
                                      FreshCons(x,decl(x1,a),G,fs,ni),
                                      p1),
                      id(Exp,Fix(x1,e1))),
          IdVarDec(x,x1))
VarNotFreeInExp(G,x,d,_,_,fs,TSTrue(_,c))               = id(Exp,True)
VarNotFreeInExp(G,x,d,_,_,fs,TSFalse(_,c))              = id(Exp,False)
```

```
  VarNotFreeInExp(G,x,d,_,a,fs,TSIf(_,d1,e1,f1,_,p1,p2,p3)) =
    idsubst(Exp,
            [d']Id(Exp,If(d1,e1,f1),If(d',Subst(x,d,e1),Subst(x,d,f1))),
            d1,
            Subst(x,d,d1),
            VarNotFreeInExp(G,x,d,d1,TBool,fs,p1),
            idsubst(Exp,
                    [e']Id(Exp,If(d1,e1,f1),If(d1,e',Subst(x,d,f1))),
                    e1,
                    Subst(x,d,e1),
                    VarNotFreeInExp(G,x,d,e1,a,fs,p2),
                    idsubst(Exp,
                            [f']Id(Exp,If(d1,e1,f1),If(d1,e1,f')),
                            f1,
                            Subst(x,d,f1),
                            VarNotFreeInExp(G,x,d,f1,a,fs,p3),
                            id(Exp,If(d1,e1,f1)))))



TypeSysofAbs : (x:VAR;
                e:Exp;
                a:Type; b:Type;
                p:TypeSystem(nil(Decl),Abs(x,e),TFunc(a,b))
               )TypeSystem(cons(Decl,decl(x,a),nil(Decl)),e,b)      [] I

  TypeSysofAbs(x,e,a,b,TSAbs(_,_,_,_,_,p1)) = p1



EqAndNotEqToTS : (x:VAR;
                  i:Id(VAR,x,x);
                  ni:Not(Id(VAR,x,x));
                  G:ListDecl;
                  a:Type
                 )TypeSystem(G,Var(x),a)                           [] I

  EqAndNotEqToTS(x,i,Imply_intro(_,_,f),G,a) = case f(i) : Absurdity of
                                               end
```

```
AbsurdityToTS : (bo:Absurdity; G:ListDecl; e:Exp; a:Type)TypeSystem(G,e,a) [] I

  AbsurdityToTS(bo,G,e,a) = case bo : Absurdity of
                            end
```

```
SubstLemma : (S:ListDecl; G:ListDecl; D:ListDecl;
              x:VAR;
              d:Exp; e:Exp;
              a:Type; b:Type;
              i:Id(ListDecl,S,Concat(cons(Decl,decl(x,a),G),D));
              p:TypeSystem(S,e,b); p1:TypeSystem(G,d,a)
             )TypeSystem(Concat(G,D),Subst(x,d,e),b)                    [] I

  SubstLemma(_,G,nil(_),x,d,_,a,_,id(_,_),TSVar(_,_,c,fs),p1) =
    Or_elim(Id(VAR,x,x),
            Not(Id(VAR,x,x)),
            [o]TypeSystem(G,
                          Or_elim(Id(VAR,x,x),
                                  Not(Id(VAR,x,x)),
                                  [o']Exp,
                                  [i]d,
                                  [ni]Var(x),
                                  o),
                          a),
            [i]p1,
            [ni]EqAndNotEqToTS(x,id(VAR,x),ni,G,a),
            IdVarDec(x,x))
```

```
SubstLemma(_,G,cons(_,dl,l),x,d,_,a,_,id(_,_),TSVar(_,_,c,fs),p1)  =
  Or_elim(Id(VAR,x,VarDecl(dl)),
          Not(Id(VAR,x,VarDecl(dl))),
          [o]TypeSystem(cons(Decl,dl,Concat(G,l)),
                        Or_elim(Id(VAR,x,VarDecl(dl)),
                                Not(Id(VAR,x,VarDecl(dl))),
                                [o']Exp,
                                [i]d,
                                [ni]Var(VarDecl(dl)),
                                o),
                        TypeDecl(dl)),
          [i]AbsurdityToTS(FreshCtxtToAbsurd(decl(x,a),
                                             G,
                                             l,
                                             c,
                                             idsubst'(VAR,
                                                  (*) [y]Fresh1,
                                                      x,
                                                      VarDecl(dl),
                                                      i,
                                                      fs)),
                           cons(Decl,dl,Concat(G,l)),
                           d,
                           TypeDecl(dl)),
          [ni]TSVar(dl,
                    Concat(G,l),
                    ConcatCtxtMon(decl(x,a),G,l,c),
                    ConcatFreshMon(decl(x,a),G,l,VarDecl(dl),fs)),
          IdVarDec(x,VarDecl(dl)))
SubstLemma(_,G,nil(_),x,d,e,a,b,id(_,_),TSThinn(_,_,_,_,p2,fs),p1) =
  idsubst(Exp,
          [e']TypeSystem(G,e',b),
          e,
          Subst(VarDecl(decl(x,a)),d,e),
          VarNotFreeInExp(G,VarDecl(decl(x,a)),d,e,b,fs,p2),
          p2)
```

```
SubstLemma(_,G,cons(_,dl,l),x,d,e,a,b,id(_,_),TSThinn(_,_,_,_,p2,fs),p1) =
  TSThinn(Concat(G,l),
          Subst(x,d,e),
          b,
          dl,
          SubstLemma(Concat(cons(Decl,decl(x,a),G),l),
                     G,
                     l,
                     x,
                     d,
                     e,
                     a,
                     b,
                     id(ListDecl,Concat(cons(Decl,decl(x,a),G),l)),
                     p2,
                     p1),
          ConcatFreshMon(decl(x,a),G,l,VarDecl(dl),fs))
```

```
SubstLemma(_,G,D,x,d,_,a,_,id(_,_),TSAbs(_,x1,e1,a1,b1,p2),p1) =
  Or_elim(Id(VAR,x,x1),
          Not(Id(VAR,x,x1)),
          [o]TypeSystem(Concat(G,D),
                        Or_elim(Id(VAR,x,x1),
                                Not(Id(VAR,x,x1)),
                                [o']Exp,
                                [i]Abs(x1,e1),
                                [ni]Abs(x1,Subst(x,d,e1)),
                                o),
                        TFunc(a1,b1)),
          [i]AbsurdityToTS(CtxtAndIdToAbsurdity(x,
                                                x1,
                                                a1,
                                                a,
                                                G,
                                                D,
                                                i,
                                             (*) Valid_Ctxt1),
                           Concat(G,D),
                           Abs(x1,e1),
                           TFunc(a1,b1)),
          [ni]TSAbs(Concat(G,D),
                    x1,
                    Subst(x,d,e1),
                    a1,
                    b1,
                    SubstLemma(Concat(cons(Decl,decl(x,a),G),
                                      cons(Decl,decl(x1,a1),D)),
                               G,
                               cons(Decl,decl(x1,a1),D),
                               x,
                               d,
                               e1,
                               a,
                               b1,
                               id(ListDecl,
                                  Concat(cons(Decl,decl(x,a),G),
                                         cons(Decl,decl(x1,a1),D))),
                               p2,
                               p1)),
          IdVarDec(x,x1))
```

```
SubstLemma(S,G,D,x,d,_,a,b,i,TSApp(_,d1,e1,a1,_,p2,p3),p1) =
  TSApp(Concat(G,D),
        Subst(x,d,d1),
        Subst(x,d,e1),
        a1,
        b,
        SubstLemma(S,G,D,x,d,d1,a,TFunc(a1,b),i,p2,p1),
        SubstLemma(S,G,D,x,d,e1,a,a1,i,p3,p1))
```

```
SubstLemma(_,G,D,x,d,_,a,b,id(_,_),TSFix(_,x1,e1,_,p2),p1) =
  Or_elim(Id(VAR,x,x1),
          Not(Id(VAR,x,x1)),
          [o]TypeSystem(Concat(G,D),
                        Or_elim(Id(VAR,x,x1),
                                Not(Id(VAR,x,x1)),
                                [o']Exp,
                                [i]Fix(x1,e1),
                                [ni]Fix(x1,Subst(x,d,e1)),
                                o),
                        b),
          [i]AbsurdityToTS(CtxtAndIdToAbsurdity(x,
                                                x1,
                                                b,
                                                a,
                                                G,
                                                D,
                                                i,
                                             (*) Valid_Ctxt2),
                           Concat(G,D),
                           Fix(x1,e1),
                           b),
          [ni]TSFix(Concat(G,D),
                    x1,
                    Subst(x,d,e1),
                    b,
                    SubstLemma(Concat(cons(Decl,decl(x,a),G),
                                      cons(Decl,decl(x1,b),D)),
                               G,
                               cons(Decl,decl(x1,b),D),
                               x,
                               d,
                               e1,
                               a,
                               b,
                               id(ListDecl,
                                  Concat(cons(Decl,decl(x,a),G),
                                         cons(Decl,decl(x1,b),D))),
                               p2,
                               p1)),
          IdVarDec(x,x1))
SubstLemma(_,G,D,x,d,_,a,_,id(_,_),TSTrue(_,c),p1)         =
  TSTrue(Concat(G,D),ConcatCtxtMon(decl(x,a),G,D,c))
SubstLemma(_,G,D,x,d,_,a,_,id(_,_),TSFalse(_,c),p1)        =
  TSFalse(Concat(G,D),ConcatCtxtMon(decl(x,a),G,D,c))
```

```
  SubstLemma(S,G,D,x,d,_,a,b,i,TSIf(_,d1,e1,f,_,p2,p3,p4),p1) =
    TSIf(Concat(G,D),
         Subst(x,d,d1),
         Subst(x,d,e1),
         Subst(x,d,f),
         b,
         SubstLemma(S,G,D,x,d,d1,a,TBool,i,p2,p1),
         SubstLemma(S,G,D,x,d,e1,a,b,i,p3,p1),
         SubstLemma(S,G,D,x,d,f,a,b,i,p4,p1))
```

Where the codes of the abbreviations are :

```
Fresh1 = Fresh(y,Concat(cons(Decl,decl(x,a),G),l))
```

```
Valid_Ctxt1 =
  ValidCtxt(cons(Decl,decl(x1,a1),Concat(cons(Decl,decl(x,a),G),D)),e1,b1,p2)
```

```
Valid_Ctxt2 =
  ValidCtxt(cons(Decl,decl(x1,b),Concat(cons(Decl,decl(x,a),G),D)),e1,b,p2)
```

A.12 Formalization of the Subject Reduction Property

```
SubjectReduction : (d:Exp; e:Exp;
                    a:Type;
                    p:DynSem(d,e);
                    p1:TypeSystem(nil(Decl),d,a)
                   )TypeSystem(nil(Decl),e,a)                    [] I

  SubjectReduction(_,_,a,DSAbs(x,e1),p1) = p1
```

```
SubjectReduction(_,e,a,DSApp(x,d1,f,e1,_,p2,p3),TSApp(_,_,_,a1,_,p,p4)) =
  SubjectReduction(Subst(x,e1,d1),
                   e,
                   a,
                   p3,
                   SubstLemma(Concat(cons(Decl,decl(x,a1),nil(Decl)),
                                     nil(Decl)),
                              nil(Decl),
                              nil(Decl),
                              x,
                              e1,
                              d1,
                              a1,
                              a,
                              id(ListDecl,
                                 Concat(cons(Decl,decl(x,a1),nil(Decl)),
                                        nil(Decl))),
                              TypeSysofAbs(x,
                                           d1,
                                           a1,
                                           a,
                                           SubjectReduction(f,
                                                            Abs(x,d1),
                                                            TFunc(a1,a),
                                                            p2,
                                                            p)),
                              p4))
SubjectReduction(_,e,a,DSFix(x,e1,_,p2),TSFix(_,_,_,_,p))                =
  SubjectReduction(Subst(x,Fix(x,e1),e1),
                   e,
                   a,
                   p2,
                   SubstLemma(Concat(cons(Decl,decl(x,a),nil(Decl)),
                                     nil(Decl)),
                              nil(Decl),
                              nil(Decl),
                              x,
                              Fix(x,e1),
                              e1,
                              a,
                              a,
                              id(ListDecl,
                                 Concat(cons(Decl,decl(x,a),nil(Decl)),
                                        nil(Decl))),
                              p,
                              TSFix(nil(Decl),x,e1,a,p)))
```

```
SubjectReduction(_,_,a,DSTrue,p1)                                    = p1
SubjectReduction(_,_,a,DSFalse,p1)                                   = p1
SubjectReduction(_,e,a,DSIfT(d1,e1,f,_,p2,p3),TSIf(_,_,_,_,_,_,p,p4,p5)) =
  SubjectReduction(e1,e,a,p3,p4)
SubjectReduction(_,e,a,DSIfF(d1,e1,f,_,p2,p3),TSIf(_,_,_,_,_,_,p,p4,p5)) =
  SubjectReduction(f,e,a,p3,p5)
```

References

[Alt 94] *A User's guide to ALF.* T. Altenkirch, V. Gaspes, B. Nordström, B. von Sydow. Department of Computer Science, University of Göteborg / Chalmers, Sweden. June, 1994.

[Aug 90] *A Short Description of Another Logical Framework.* L. Augustsson, T. Coquand, B. Nordström. In Proceedings of the First Workshop on Logical Frameworks, Antibes, pages 39 - 42. 1990.

[Bar 92] *Lambda Calculi with Types.* H. P. Barendregt. In Handbook of Logic in Computer Science, Vol. II. Gabbai, Abramsky and Maibaum editors. Oxford University Press. 1992.

[Cle 86] *A Simple Applicative Language : Mini - ML.* D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn. Technical Report 529, INRIA, France. May, 1986.

[Coq 94] *Type Theory and Programming.* T. Coquand, B. Nordström, J. M. Smith, B. von Sydow. EATCS Bulletin. February, 1994.

[Dam 82] *Principal Type-Schemes for Functional Programs.* L. Damas, R. Milner. In Proceedings 9th. ACM Symposium on Principles of Programming Languages, Albuquerque NM. January, 1982.

[Geu 90] *Type Systems for Higher Order Logic.* J. H. Geuvers. Technical Report of the Department of Computer Science, Catholic University, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands. 1990.

[Har 93] *A Framework for Defining Logics.* R. Harper, F. Honsell, G. Plotking. In Journal of the Association for Computing Machinery, Vol. 40, no 1. January, 1993.

[Hen 90] *The Semantics of Programming Languages : An Elementary Introduction using Structural Operational Semantics.* M. Hennessy. University of Sussex, U.K. John Wiley & Sons Eds. 1990.

[Hol 83] *Polymorphic Type Systems : A Proof - Theoretic Approach.* S. Holmström. Technical Report 6, University of Goteborg and Chalmers University of Technology, Sweden. September, 1983.

[Kah 87] *Natural Semantics.* G. Kahn. Technical Report 601, INRIA, France. February, 1987.

[Mag 92] *The New Implementation of ALF.* L. Magnusson. In Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden. June, 1992.

[Mag 94] *The ALF Proof Editor and its Proof Engine.* L. Magnusson, B. Nordström. In Types for Proofs and Programs : International Workshop TYPES'93, Selected Papers. Nijmegen, The Netherlands. Springer-Verlag Lecture Notes in Computer Science 806. H. Barendregt and T. Nipkow Eds. 1994.

[Mic 91] *Natural Semantics and Some of its Meta-Theory in Elf.* S. Michaylov, F. Pfenning. In Proceedings of the Second International Workshop on Extensions of Logic Programming, Stockholm, Sweden. Springer-Verlag Lecture Notes in Artificial Intelligence 596. L.-H. Eriksson, L. Hallnäs and P. Schroeder-Heister Eds. 1991.

[Mil 78] *A Theory of Type Polymorphism in Programming.* R. Milner. In Journal of Computer and System Sciences, Vol. 17, no. 3. 1978.

[Nie 92] *Semantics with Applications : A Formal Introduction.* H. Nielson, F. Nielson. Aarhus University, Denmark. John Wiley & Sons Eds. 1992.

[Nor 90] *Programming in Martin - Löf's Type Theory. An Introduction.* B. Nordström, K. Petersson, J. M. Smith. Oxford University Press. 1990.

[Pfe 91] *Logic Programming in the LF Logical Framework.* F. Pfenning. In *Logical Frameworks.* Cambridge University Press. G. Huet and G. Plotkin Eds. 1991.

[Plo 81] *A Structural Approach to Operational Semantics.* G. Plotkin. Report DAIMI FN-19. University of Aarhus, Denmark. 1981.

[Pol 93] *Closure Under Alpha-Conversion.* R. Pollack. In Types for Proofs and Programs : International Workshop TYPES'93, Selected Papers. Nijmegen, The Netherlands. Springer-Verlag Lecture Notes in Computer Science 806. H. Barendregt and T. Nipkow Eds. 1994.

[Pol 94] *The Theory of LEGO : A Poof Checker for the Extended Calculus of Constructions.* R. Pollack. Doctor of Philosophy Thesis. University of Edinburgh. Available by anonymous ftp from `ftp.cs.chalmers.se` in directory `pub/users/pollack`. 1994.