A Machine-assisted Proof that Well Typed Expressions Cannot Go Wrong

Ana Bove
bove@cs.chalmers.se

Department of Computing Science, Chalmers University of Technology
Göteborg, Sweden

Abstract. This paper deals with the application of constructive type theory to the theory of programming languages. The main aim of this work is to investigate constructive formalisations of the mathematics of programs. Here, we consider a small typed functional language and prove some properties about it, arriving at the property that establishes that *well typed expressions cannot go wrong*. First, we give the definitions and proofs in an informal style, and then we present and explain the formalisation of these definitions and proofs. For the formalisation, we use the proof editor ALF and its pattern matching facility.

1 Introduction

This paper deals with the application of constructive type theory to the theory of programming languages. By constructive type theory we understand first and foremost Martin-Löf's theory of logical types (see [NPS90]), which is conceived as a formal language in which to carry out constructive mathematics.

However, constructive type theory can also be viewed as a programming language. In type theory, we represent theorems as types and proofs of the theorems as objects of the corresponding types. When a 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 hypotheses of the theorem. Thus, many important algorithms used in the process of interpretation or compilation of programming languages arise naturally as proofs of properties of the language.

The main aim of this work is to investigate constructive formalisations of the mathematics of programs. It also aims to explore the production of verified implementations of programming languages.

This paper is intended for readers who have some basic knowledge of operational semantics, type systems, Martin-Löf's type theory and its proof editor ALF. An extended version of this paper can be found in [Bov98].

This paper is organised as follows. In section 2 we present the language we use. Furthermore, we present several properties of the language, concluding with the property that shows that *well typed (closed) expressions cannot go wrong*. In section 3 we first give a brief introduction to Martin-Löf's type theory and to its interactive proof editor ALF, and then we present a formalisation of the results

$$
\begin{array}{ll}
y\,[d/x] & \stackrel{def}{=} \begin{cases} y & \text{if } y \neq x \\ d & \text{if } y = x \end{cases} \\
(\lambda y.e)\,[d/x] & \stackrel{def}{=} \begin{cases} \lambda y.(e\,[d/x]) & \text{if } y \neq x \\ \lambda y.e & \text{if } y = x \end{cases} \\
(e \;\; f)\,[d/x] & \stackrel{def}{=} (e\,[d/x] \;\; f\,[d/x]) \\
(\mathsf{fix}\ y\,.\,e)\,[d/x] & \stackrel{def}{=} \begin{cases} \mathsf{fix}\ y\,.\,(e\,[d/x]) & \text{if } y \neq x \\ \mathsf{fix}\ y\,.\,e & \text{if } y = x \end{cases} \\
\mathsf{true}\ [d/x] & \stackrel{def}{=} \mathsf{true} \\
\mathsf{false}\ [d/x] & \stackrel{def}{=} \mathsf{false} \\
(\mathsf{if}\ e\ \mathsf{then}\ f\ \mathsf{else}\ g)\,[d/x] & \stackrel{def}{=} \mathsf{if}\ e\,[d/x]\ \mathsf{then}\ f\,[d/x]\ \mathsf{else}\ g\,[d/x]
\end{array}
$$

Fig. 1 Definition of the function $\_\,[\_\,/\_\,]$ over *Exp*

presented in section 2, using Martin-Löf's type theory. Finally, in section 4 we present some conclusions, related work and further work.

2 Informal Presentation

2.1 The Syntax of the Language

In order to define the set of expressions *Exp*, we assume a (possibly infinite) set of variables *Var* such that for each pair of variables, it is decidable whether or not they are equal. We use x, y, z to range over variables and d, e, f, g (possibly primed or subscripted), to range over expressions.

The expressions in *Exp* are those of a (small) functional language. We define the expressions in *Exp* by means of its abstract syntax as follows :

$$e ::= x \;\; | \, \lambda x.e \;\; | \, (d \;\; e) \;\; | \, \mathsf{fix}\ x\,.\,e \;\; | \, \mathsf{true} \;\; | \, \mathsf{false} \;\; | \, \mathsf{if}\ d\ \mathsf{then}\ e\ \mathsf{else}\ f$$

When writing concrete expressions, we sometimes use parentheses to avoid ambiguity. We define the set FV of free variables in an expression in the usual way (see [Bar92]). A *closed expression* is an expression with no free variables.

It is easy to see that because the equality of variables is decidable, so is the equality of expressions.

In order to define the dynamic semantics for the language, we need to introduce the *substitution* of an expression d for a variable x in an expression e, which we write $e\,[d/x]$. In the definition of the substitution function given in figure 1, we do not care about capturing variables. This is because the evaluation of expressions is intended only for closed expressions, and no evaluation is performed inside a binding operator as is normal practice in functional programming. Thus, no capture can occur.

2.2 The Dynamic Semantics of the Language

We give the dynamic semantics of the language in an operational style (see [Plo81]). Here we use *Computational semantics* [Hen90], also known as *Structural*

$$\mathsf{App_{cs}} \quad \frac{d \longrightarrow d'}{(d\ \ e) \longrightarrow (d'\ \ e)}$$

$$\mathsf{App\_Abs_{cs}} \quad (\lambda x.d\ \ e) \longrightarrow d[e/x]$$

$$\mathsf{Fix_{cs}} \quad \mathsf{fix}\ x\,.\,e \longrightarrow e[\mathsf{fix}\ x\,.\,e/x]$$

$$\mathsf{If_{cs}} \quad \frac{d \longrightarrow d'}{\mathsf{if}\ d\ \mathsf{then}\ e\ \mathsf{else}\ f \longrightarrow \mathsf{if}\ d'\ \mathsf{then}\ e\ \mathsf{else}\ f}$$

$$\mathsf{If\_True_{cs}} \quad \mathsf{if\ true\ then}\ e\ \mathsf{else}\ f \longrightarrow e$$

$$\mathsf{If\_False_{cs}} \quad \mathsf{if\ false\ then}\ e\ \mathsf{else}\ f \longrightarrow f$$

$$\mathsf{Refl_{cl}} \quad e \longrightarrow^* e$$

$$\mathsf{AddStep_{cl}} \quad \frac{e \longrightarrow d \qquad d \longrightarrow^* f}{e \longrightarrow^* f}$$

Fig. 2 Inductive Definition of the Computational Semantics of the Language and its Reflexive and Transitive Closure

Operational semantics [Plo81], because it allows us to define those expressions that *go wrong*. Computational semantics defines a "one step relation". Thus, a computation consists of a sequence of one step reductions. If we denote the Computational semantics between two expressions with $\longrightarrow$, then we write $e \longrightarrow^* e'$ if for some $k \geqslant 0$, there exist expressions e_0, e_1, ..., e_k such that $e = e_0 \longrightarrow e_1 \longrightarrow \ldots \longrightarrow e_k = e'$. The relation $\longrightarrow^*$ defined as above is called the *reflexive and transitive closure* of $\longrightarrow$. We define both the relation $\longrightarrow$ and $\longrightarrow^*$ in figure 2. Note that for any expression e, $e \longrightarrow^* e$ holds even if $e \longrightarrow e$ is not true.

If $\longrightarrow^*$ represents an arbitrary number (possibly zero) of computation steps of $\longrightarrow$, the normal procedure would be to run the machine until no further computation steps are possible. Then, we obtain an *irreducible* expression as a result. For our language, we call the expressions of the form true, false or $\lambda x.e$ *canonical*. Note that no reduction rules are given for the canonical expressions; thus, canonical expressions are irreducible. On the other hand, there are irreducible expressions that are not canonical, such as the expression if $\lambda x.x$ then true else false. We call *error expressions* those irreducible expressions that are not canonical. If for expressions d and e, $d \longrightarrow^* e$ and e is irreducible, we say that e is the *result* (of evaluating d) or the *value* of d. Observe that canonical expressions have themselves as values. When there exists a result of d that is an error expression, we say that d *goes wrong* (see [Hol83,Mil78]). There are also expressions with no value, such as the expression fix $x\,.\,x$, which gives rise to an infinite computation.

2.3 The Type System

The Set of Types. We call *Type* the set of types we use in this paper and it contains both the type of boolean expression and the function types. Below, we define it by means of its abstract syntax. We use α, β, γ, δ to denote elements in the set *Type*. We sometimes use parentheses to avoid ambiguity.

$$\mathsf{Empty}_{\mathsf{ctxt}} \quad [\,] : \mathit{Ctxt}$$

$$\mathsf{Cons}_{\mathsf{ctxt}} \quad [\Gamma,\ x\ :\ \alpha] : \mathit{Ctxt}$$

$$\mathsf{Empty}_{\mathsf{fresh}} \quad x\ \mathsf{fresh}\ [\,]$$

$$\mathsf{Cons}_{\mathsf{fresh}} \quad \frac{x\ \mathsf{fresh}\ \Gamma \qquad x \neq y}{x\ \mathsf{fresh}\ [\Gamma,\ y\ :\ \alpha]}$$

$$\mathsf{Empty}_{\mathsf{val}} \quad [\,]\ \mathsf{Valid}$$

$$\mathsf{Cons}_{\mathsf{val}} \quad \frac{\Gamma\ \mathsf{Valid} \qquad x\ \mathsf{fresh}\ \Gamma}{[\Gamma,\ x\ :\ \alpha]\ \mathsf{Valid}}$$

Fig. 3 Inductive Definition of *Ctxt* and predicates fresh and Valid

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

The Contexts for the Type System. The contexts we use here are (in principle) lists of *declarations*. Each declaration associates a type to a variable and has the form $x\ :\ \alpha$. We use Γ, Δ, Σ (possibly primed), to range over contexts.

In this paper, we are interested only in those contexts where each variable is declared at most once. To ensure this, we define two predicates over contexts : the predicate fresh that says whether or not a variable x is fresh (not declared) in a context Γ, and the predicate Valid that says whether or not a context is valid in the sense that no variable in the context is declared more than once. We define the set of contexts *Ctxt* and the predicates fresh and Valid in figure 3. When writing concrete contexts, we sometimes use parentheses to avoid ambiguity.

To formulate the *Substitution Lemma*, we need to define the concatenation of two contexts. We denote the function that concatenates two contexts by $\_ +\!+ \_$ and we define it by induction on the second argument.

Presentation of the Type System. The type system we present in this section is essentially the same as that presented in [Bov95] and [Tas97]. The type system tells us when an expression e has type α under a context Γ, which is denoted by $\Gamma \vdash e : \alpha$, and it is defined in figure 4. Both [Bov95] and [Tas97] describe the importance of the thinning rule ($\mathsf{Th}_{\mathsf{ts}}$) when working with this particular notion of contexts, and the possible use of other similar type systems.

For simplicity, we write $\vdash e : \alpha$ instead of $[\,] \vdash e : \alpha$.

2.4 Properties of the Language

In this section, we present some properties of the language that relate its dynamic semantics with its type system. We first present the Subject Reduction property and then we finish the section with the proof that *well typed (closed) expressions cannot go wrong*. For a more detailed presentation, see [Bov95,Bov98].

$$\mathsf{Var_{ts}} \quad \frac{\Gamma \text{ Valid} \qquad x \text{ fresh } \Gamma}{[\Gamma,\ x\ :\ \alpha] \vdash x : \alpha}$$

$$\mathsf{Th_{ts}} \quad \frac{\Gamma \vdash e : \alpha \qquad x \text{ fresh } \Gamma}{[\Gamma,\ x\ :\ \beta] \vdash e : \alpha}$$

$$\mathsf{Abs_{ts}} \quad \frac{[\Gamma,\ x\ :\ \alpha] \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha\ \rightarrow\ \beta}$$

$$\mathsf{App_{ts}} \quad \frac{\Gamma \vdash e : \alpha\ \rightarrow\ \beta \qquad \Gamma \vdash f : \alpha}{\Gamma \vdash (e\ \ f) : \beta}$$

$$\mathsf{Fix_{ts}} \quad \frac{[\Gamma,\ x\ :\ \alpha] \vdash e : \alpha}{\Gamma \vdash \mathsf{fix}\ x\,.\,e : \alpha}$$

$$\mathsf{True_{ts}} \quad \frac{\Gamma \text{ Valid}}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}$$

$$\mathsf{False_{ts}} \quad \frac{\Gamma \text{ Valid}}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

$$\mathsf{If_{ts}} \quad \frac{\Gamma \vdash d : \mathsf{Bool} \qquad \Gamma \vdash e : \alpha \qquad \Gamma \vdash f : \alpha}{\Gamma \vdash \mathsf{if}\ d\ \mathsf{then}\ e\ \mathsf{else}\ f : \alpha}$$

Fig. 4 Inductive Definition of the Type System for *Exp*

Theorem 1 (Substitution Lemma). *Let* $[\Gamma,\ x\ :\ \alpha]$ *and* Δ *be 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. By induction on the derivation of $[\Gamma,\ x\ :\ \alpha] +\!\!+ \Delta \vdash e : \beta$. ■

Theorem 2 (Subject Reduction). *Let* d *and* e *be expressions and* α *a type. If* $d \longrightarrow e$ *and* $\vdash d : \alpha$*, then* $\vdash e : \alpha$*.*

Proof. The proof is by induction on the derivation of $d \longrightarrow e$. For each case in this derivation, we consider the possible cases in the derivation of $\vdash d : \alpha$. ■

Lemma 1. *Let* d *be a closed expression and* α *a type. If* $\vdash d : \alpha$*, then either* d *is a canonical expression or there exists an expression* e *such that* $d \longrightarrow e$*.*

Proof. The proof is by induction on the derivation of $\vdash d : \alpha$. We present the case where the rule $\mathsf{App_{ts}}$ is applied. The other cases are rather straightforward.

If the rule applied was the rule $\mathsf{App_{ts}}$, then d is of the form $(f\ \ g)$ for two suitable expressions f and g. From the premises of this rule, we have that f has type $\beta\ \rightarrow\ \alpha$ (under the empty context), for a type β. This means that f is a well typed closed expression. Thus, by the induction hypothesis, either f is canonical or it reduces. If f is canonical, as it has type $\beta\ \rightarrow\ \alpha$, it can only be of the form $\lambda x.f'$ for a variable x and an expression f'. Then, we can apply the rule $\mathsf{App\_Abs_{cs}}$ to reduce the whole expression $(f\ \ g)$ and obtain $f'[g/x]$ as a result. On the other hand, if f is reducible, then, there exists f'' such that $f \longrightarrow f''$. Now, we can apply the rule $\mathsf{App_{cs}}$ to reduce the whole expression

and obtain $(f''\ \ g)$. In both cases, we show that the original expression can be reduced. ■

Lemma 2. *Let d be a closed expression and α be a type. From assuming that $\vdash d : \alpha$ holds and also that d goes wrong, we obtain a contradiction.*

Proof. Let us assume that $\vdash d : \alpha$ holds and also that d goes wrong. Thus, we know that d has an error expression e as value. Then, we perform the proof of the proposition by induction on the derivation of $d \longrightarrow^* e$.

If the rule applied was the rule $\mathsf{Refl}_{\mathsf{CI}}$, we have that d $\longrightarrow^*$ d and so d = e. Then, by assumption, d is an error expression. As $\vdash d : \alpha$, we can apply lemma 1 to obtain that either d is a canonical expression or there exists an expression f to which expression d reduces. However, d is an error expression, so it is non-canonical and irreducible, which contradicts the result of lemma 1.

If the rule applied was the rule $\mathsf{AddStep}_{\mathsf{CI}}$, it means that there exists an expression f such that $d \longrightarrow f \longrightarrow^* e$. As e is a value of d, e is irreducible. Hence, it is also a value of f. Since e is an error expression, then, by definition, f goes wrong. As $\vdash d : \alpha$ holds, by Subject Reduction, we also have that $\vdash f : \alpha$. We have now that $\vdash f : \alpha$ and f goes wrong. Thus, by the induction hypothesis, we obtain a contradiction. ■

Theorem 3 (Well Typed Expressions Cannot Go Wrong). *Let d be a well typed closed expression. Then, d cannot go wrong.*

Proof. By hypothesis, the expression d is a well typed closed expression. Then, if we assume that d goes wrong, we can apply the previous lemma and obtain a contradiction. Thus, d cannot go wrong. ■

For another but similar method to prove this property, see [Hol83,Mil78].

3 Formalisation in ALF

In this section, we first present a brief introduction to Martin-Löf's type theory and to its interactive proof editor ALF, and then we present a formalisation of the results presented in section 2, using Martin-Löf's type theory.

3.1 Brief Introduction to Martin-Löf's Type Theory

Below, we explain the basic concepts of Martin-Löf's type theory to make the following sections more readable. We write $a \in \alpha$ for "a is an object of type α". For a more complete introduction to the subject, refer to [NPS90].

Sets : The only basic type is the type of sets. Sets are inductively defined. In other words, 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 form a type called **El**(S). However, for simplicity, if a is an element in the set S, it is said that a has type S and thus, we can simply write $a \in S$ instead of $a \in$ **El**(S).

Dependent (and Non-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. 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 $(x \in \alpha)\,\beta(x)$ for the type of dependent functions from α to β. If f is a function of type $(x \in \alpha)\,\beta(x)$, then when applying f to an object a of type α we obtain an object of type $\beta(a)$. We write $f(a)$ for such an application.

A (non-dependent) function is considered a special case of a dependent function, where the type β does not depend on a value of type α. When this is the case, we may write $(\alpha)\,\beta$ for the function type from α to β.

Let us now consider predicates and relations on sets, and arbitrary complex propositions. Predicates and relations are seen in type theory as functions yielding propositions as output. As well as sets, propositions are inductively defined. So, a proposition is determined by the rules that construct its proofs. To prove a proposition P, we have to construct an object of type P. The way propositions are introduced allows us to identify propositions and sets, which is actually done in type theory, and then, we also write **Set** to refer to the type of propositions.

3.2 Brief Introduction to ALF

ALF is an interactive proof assistant for Martin-Löf's type theory and it ensures that the constructed objects are well-formed and well-typed. Since proofs are objects, checking well-typing of objects amounts to checking correctness of proofs. For more information about ALF see [AGNvS94,MN94].

A set former, or in general, any inductive definition is introduced as a constant S of type $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\,\mathbf{Set}$, for $\alpha_1, \ldots, \alpha_n$ types. For each set former, we have to introduce the constructors associated to the set. They construct the elements of $S(a_1, \ldots, a_n)$, for $a_1 \in \alpha_1; \ldots; a_n \in \alpha_n$. A theorem is introduced as a dependent type $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\,\beta(x_1, \ldots, x_n)$. Abstractions are written as $[x_1, \ldots, x_n]\,e$.

If the name of a variable is not important, one can simply write (α) instead of $(x \in \alpha)$, both in the introduction of inductive definitions and in the declaration of (dependent) functions. Whenever $(x_1 \in \alpha;\ x_2 \in \alpha; \ldots;\ x_n \in \alpha)$ occurs, ALF displays $(x_1, x_2, \ldots, x_n \in \alpha)$ instead.

A proof for a theorem can be defined by *pattern matching* over one of the arguments of the theorem. The various cases in the pattern matching are exhaustive and mutually disjoint. Moreover, they are computed by ALF according to the definition of the set to which the selected argument belongs. In general, theorems are proven by induction. Unfortunately, ALF does not check well-foundedness when working with inductive proofs. However, for the proofs we present in this paper, these checks are easy – even if rather tedious – to perform manually.

3.3 Working with ALF

All the definitions and proofs we present here have been pretty printed by ALF itself. Then, all of them have been checked in ALF. In addition, we have made use of the layout facility of ALF that allows us to hide some parameters, both in the definitions of sets and theorems. However, this has only been done when the hidden parameters do not contribute to the understanding of the definition.

The proofs are made by pattern matching. In some of the proofs we also apply recursion over some of the arguments. However, termination is guaranteed because we always apply the recursion over a structurally smaller argument.

Due to lack of space, we present only the type of some of the properties introduced in this section. Once again, this is done when the complete formalisation of the property is rather straightforward.

General Sets. In the proofs that we present in the next part, we make use of the following set formers and constructors, and theorems :

Absurdity : The set former is $\bot \in \mathbf{Set}$, and has no set constructors.

And : This set represents the conjunction of two propositions. The set former is $\wedge \in (A, B \in \mathbf{Set})\,\mathbf{Set}$ and the only set constructor is $\wedge_{\mathrm{I}} \in (A; B)\wedge(A, B)$.

Exists : Represents the existential quantifier over a set. The set former is $\exists \in (A \in \mathbf{Set};\ (A)\,\mathbf{Set})\,\mathbf{Set}$ and its set constructor is $\exists_{\mathrm{I}} \in (a \in A;\ B(a))\,\exists(A, B)$.

Imply : Represents the implication between two propositions. The set former is $\supset \in (A, B \in \mathbf{Set})\,\mathbf{Set}$ and the only set constructor is $\supset_{\mathrm{I}} \in (f \in (A)\,B)\supset(A, B)$.

Id : Represents the propositional equality. The set former is $= \in (x, y \in A)\,\mathbf{Set}$ and the only set constructor is $\mathrm{refl} \in\ =(x, x)$.

Not : Represents the negation operator. We have that $\neg \in (A \in \mathbf{Set})\,\mathbf{Set}$ and it is defined as the abbreviation $\neg \equiv [A]\supset(A,\bot)$.

N : Represents the set of natural numbers. Together with the definition of the set, we present a property over the set N.

$$
\begin{array}{l}
\mathrm{N} \in \mathbf{Set} \\
\quad 0 \in \mathrm{N} \\
\quad \mathrm{s} \in (n \in \mathrm{N})\,\mathrm{N} \\
\mathrm{N}_{\mathrm{dec}} \in (n, m \in \mathrm{N})\,\mathrm{Dec}(=(n, m))
\end{array}
$$

Or : This set represents the disjunction of two propositions. The set former is $\vee \in (A, B \in \mathbf{Set})\,\mathbf{Set}$ and its two set constructors are $\vee_{\mathrm{Il}} \in (A)\vee(A, B)$ and $\vee_{\mathrm{Ir}} \in (B)\vee(A, B)$.

The set former Dec used above is not as general as the ones just introduced. We can think of it as the set of decidable of propositions.

$$
\begin{array}{l}
\mathrm{Dec} \in (\mathbf{Set})\,\mathbf{Set} \\
\quad \mathrm{yes} \in (P)\,\mathrm{Dec}(P) \\
\quad \mathrm{no} \in (\neg(P))\,\mathrm{Dec}(P)
\end{array}
$$

Specific Sets and Implicit Constants. We can now focus on the formalisation of our functional language and its properties.

Variables and Expressions. We represent the set of variables with the set of natural numbers and then, the decidability of the equality of natural numbers becomes the decidability of the equality of variables.

$\mathrm{Var} \in \mathbf{Set}$
$\quad \mathrm{Var} \equiv \mathrm{N}$
$\mathrm{var}_{\mathrm{dec}} \in (n, m \in \mathrm{Var})\ \mathrm{Dec}(=(n, m))$
$\quad \mathrm{var}_{\mathrm{dec}} \equiv \mathrm{N}_{\mathrm{dec}}$

The formalisation of expressions and canonical expressions follows naturally :

$\mathrm{Exp} \in \mathbf{Set}$
$\quad \mathrm{v} \in (x \in \mathrm{Var})\ \mathrm{Exp}$
$\quad \lambda \in (x \in \mathrm{Var};\ e \in \mathrm{Exp})\ \mathrm{Exp}$
$\quad . \in (d, e \in \mathrm{Exp})\ \mathrm{Exp}$
$\quad \mathrm{fix} \in (x \in \mathrm{Var};\ e \in \mathrm{Exp})\ \mathrm{Exp}$
$\quad \mathrm{true} \in \mathrm{Exp}$
$\quad \mathrm{false} \in \mathrm{Exp}$
$\quad \mathrm{if} \in (d, e, f \in \mathrm{Exp})\ \mathrm{Exp}$

$\mathrm{Can} \in (\mathrm{Exp})\ \mathbf{Set}$
$\quad \mathrm{can}_{\mathrm{abs}} \in \mathrm{Can}(\lambda(x, e))$
$\quad \mathrm{can}_{\mathrm{false}} \in \mathrm{Can}(\mathrm{false})$
$\quad \mathrm{can}_{\mathrm{true}} \in \mathrm{Can}(\mathrm{true})$

For the definition of the substitution function we need to choose between two expressions, depending on the equality of two (given) variables. We present now the definition of a function that helps us in this process :

$\mathrm{var_{to}exp}_{\mathrm{dec}} \in (d, e \in \mathrm{Exp};\ \mathrm{Dec}(=(x, y)))\ \mathrm{Exp}$
$\quad \mathrm{var_{to}exp}_{\mathrm{dec}}(d, e, \mathrm{yes}(h_1)) \equiv d$
$\quad \mathrm{var_{to}exp}_{\mathrm{dec}}(d, e, \mathrm{no}(h_1)) \equiv e$

We present now the definition of the substitution function :

$:= \in (x \in \mathrm{Var};\ d, e \in \mathrm{Exp})\ \mathrm{Exp}$
$\quad :=(x, d, \mathrm{v}(x_1)) \equiv \mathrm{var_{to}exp}_{\mathrm{dec}}(d, \mathrm{v}(x_1), \mathrm{var}_{\mathrm{dec}}(x, x_1))$
$\quad :=(x, d, \lambda(x_1, e_1)) \equiv \mathrm{var_{to}exp}_{\mathrm{dec}}(\lambda(x_1, e_1), \lambda(x_1, :=(x, d, e_1)), \mathrm{var}_{\mathrm{dec}}(x, x_1))$
$\quad :=(x, d, .(d_1, e_1)) \equiv .(:=(x, d, d_1), :=(x, d, e_1))$
$\quad :=(x, d, \mathrm{fix}(x_1, e_1)) \equiv \mathrm{var_{to}exp}_{\mathrm{dec}}(\mathrm{fix}(x_1, e_1), \mathrm{fix}(x_1, :=(x, d, e_1)), \mathrm{var}_{\mathrm{dec}}(x, x_1))$
$\quad :=(x, d, \mathrm{true}) \equiv \mathrm{true}$
$\quad :=(x, d, \mathrm{false}) \equiv \mathrm{false}$
$\quad :=(x, d, \mathrm{if}(d_1, e_1, f)) \equiv \mathrm{if}(:=(x, d, d_1), :=(x, d, e_1), :=(x, d, f))$

The following two lemmas help us to prove the equality of two expressions.

$:=_{\mathrm{diff\text{–}var}} \in (\neg(=(x, y));\ p \in \mathrm{Dec}(=(x, y))) = (\mathrm{var_{to}exp}_{\mathrm{dec}}(d, e, p), e)$
$:=_{\mathrm{same\text{–}var}} \in (p \in \mathrm{Dec}(=(x, x))) = (\mathrm{var_{to}exp}_{\mathrm{dec}}(d, e, p), d)$

We use these two lemmas to prove several lemmas about equality of expressions, where one of the expressions is the result of performing a substitution.

$=_{\mathrm{same\text{–}var}_{:=}} \in = (:=(x, e, \mathrm{v}(x)), e)$
$=_{\mathrm{same\text{–}\lambda}_{:=}} \in (=(x, y)) = (:=(x, d, \lambda(y, e)), \lambda(y, e))$
$=_{\mathrm{same\text{–}fix}_{:=}} \in (=(x, y)) = (:=(x, d, \mathrm{fix}(y, e)), \mathrm{fix}(y, e))$
$=_{\mathrm{diff\text{–}var}_{:=}} \in (\neg(=(x, y))) = (:=(x, e, \mathrm{v}(y)), \mathrm{v}(y))$
$=_{\mathrm{diff\text{–}\lambda}_{:=}} \in (\neg(=(x, y))) = (:=(x, d, \lambda(y, e)), \lambda(y, :=(x, d, e)))$
$=_{\mathrm{diff\text{–}fix}_{:=}} \in (\neg(=(x, y))) = (:=(x, d, \mathrm{fix}(y, e)), \mathrm{fix}(y, :=(x, d, e)))$

Computational Semantics and its Reflexive and Transitive Closure. We introduce now the formalisation of the Computational semantics for the expressions and its reflexive and transitive closure.

$\rightarrow \in$ (Exp; Exp) **Set**
$.\rightarrow \in (\rightarrow(d,f)) \rightarrow(.(d,e),.(f,e))$
$.\lambda\rightarrow \in \rightarrow(.(\lambda(x,d),e),:=(x,e,d))$
$\text{fix}\rightarrow \in \rightarrow(\text{fix}(x,e),:=(x,\text{fix}(x,e),e))$
$\text{if}\rightarrow \in (\rightarrow(d,d')) \rightarrow(\text{if}(d,e,f),\text{if}(d',e,f))$
$\text{if}_{\text{true}}\rightarrow \in \rightarrow(\text{if}(\text{true},e,f),e)$
$\text{if}_{\text{false}}\rightarrow \in \rightarrow(\text{if}(\text{false},e,f),f)$

$\rightarrow^* \in$ (Exp; Exp) **Set**
$\text{refl}_{\rightarrow^*} \in \rightarrow^*(e,e)$
$\text{addstep}_{\rightarrow^*} \in (\rightarrow(d,e);\ \rightarrow^*(e,f)) \rightarrow^*(d,f)$

Types, Contexts and Properties. We first present the formalisation of the set of types.

Type $\in$ **Set**
bool $\in$ Type
$\rightarrow_T \in$ (Type; Type) Type

We introduce now the formalisation of declarations, contexts and two functions over contexts.

Decl $\in$ **Set**
$. \in (x \in \text{Var}; A \in \text{Type})$ Decl
Ctxt $\in$ **Set**
$[] \in$ Ctxt
$: \in (D \in \text{Ctxt}; d \in \text{Decl})$ Ctxt

$\text{mk}_{\text{nectxt}} \in (G \in \text{Ctxt}; x \in \text{Var}; A \in \text{Type})$ Ctxt
$\text{mk}_{\text{nectxt}}(G,x,A) \equiv :(G,.(x,A))$
$++ \in (G, D \in \text{Ctxt})$ Ctxt
$++(G,[]) \equiv G$
$++(G,:(D_1,d)) \equiv :(++(G,D_1),d)$

Below are the formalisation of the predicates fresh and Valid.

$\text{Fresh} \in (x \in \text{Var}; G \in \text{Ctxt})$ **Set**
$[]_{\text{fresh}} \in \text{Fresh}(x,[])$
$:_{\text{fresh}} \in (\neg(=(x,y)); \text{Fresh}(x,G))\ \text{Fresh}(x,\text{mk}_{\text{nectxt}}(G,y,A))$
$\text{Valid} \in (G \in \text{Ctxt})$ **Set**
$[]_{\text{valid}} \in \text{Valid}([])$
$:_{\text{valid}} \in (\text{Valid}(G); \text{Fresh}(x,G))\ \text{Valid}(\text{mk}_{\text{nectxt}}(G,x,A))$

We need two auxiliary properties about the freshness of variables.

$\text{dep}_{++\text{fresh}} \in (D \in \text{Ctxt}; \text{Fresh}(x, ++(:(G,d),D)))\ \text{Fresh}(x, ++(G,D))$
$\text{diff}_{\text{fresh}} \in (D \in \text{Ctxt}; \text{Fresh}(y, ++(\text{mk}_{\text{nectxt}}(G,x,A),D)))\ \neg(=(x,y))$

We also need some properties related to the validity of contexts.

$\text{diff}_{\text{valid}} \in (\text{Valid}(\text{mk}_{\text{nectxt}}(++(\text{mk}_{\text{nectxt}}(G,x,A),D),y,B)))\ \neg(=(x,y))$
$\text{dep}_{++\text{valid}} \in (D \in \text{Ctxt}; \text{Valid}(++(\text{mk}_{\text{nectxt}}(G,x,A),D)))\ \text{Valid}(++(G,D))$
$\text{dep}_{\text{valid}} \in (\text{Valid}(\text{mk}_{\text{nectxt}}(G,x,A)))\ \text{Valid}(G)$

The Type System. We present now the formalisation of the type system.

$\vdash$ $\in$ (Ctxt; Exp; Type) **Set**
  $v_{\vdash}$ $\in$ (Valid(G); Fresh(x, G)) $\vdash$(mk$_{ne_{ctxt}}$(G, x, A), v(x), A)
  th$_{\vdash}$ $\in$ (Fresh(x, G); $\vdash$(G, e, B)) $\vdash$(mk$_{ne_{ctxt}}$(G, x, A), e, B)
  $\lambda_{\downarrow\vdash}$ $\in$ ($\vdash$(mk$_{ne_{ctxt}}$(G, x, A), e, B)) $\vdash$($G, \lambda(x, e), \rightarrow_T(A, B)$)
  $._{\downarrow\vdash}$ $\in$ ($\vdash$($G, d, \rightarrow_T(A, B)$); $\vdash$(G, e, A)) $\vdash$($G, .(d, e), B$)
  fix$_{\vdash}$ $\in$ ($\vdash$(mk$_{ne_{ctxt}}$(G, x, A), e, A)) $\vdash$(G, fix(x, e), A)
  true$_{\vdash}$ $\in$ (Valid(G)) $\vdash$(G, true, bool)
  false$_{\vdash}$ $\in$ (Valid(G)) $\vdash$(G, false, bool)
  if$_{\vdash}$ $\in$ ($\vdash$(G, d, bool); $\vdash$(G, e, A); $\vdash$(G, f, A)) $\vdash$(G, if(d, e, f), A)

The next property follows immediately.

subst$_{=\vdash}$ $\in$ (=(d, e); $\vdash$(G, e, A)) $\vdash$(G, d, A)

Substitution Lemma. First, we present two auxiliary properties.

valid$_{ctxt}$ $\in$ ($\vdash$(G, e, A)) Valid(G)
=$_{:=_{free\_var}}$ $\in$ (Fresh(x, G); $\vdash$(G, e, A)) =(:=(x, d, e), e)

We introduce now the formalisation of the Substitution Lemma. We perform the proof of the lemma by induction on the derivation of $[\Gamma,\ x\ :\ \alpha] +\!\!+ \Delta \vdash e : \beta$. For this, ALF tries to unify $[\Gamma,\ x\ :\ \alpha] +\!\!+ \Delta \vdash e : \beta$ with the conclusion of each rule in the type system. Then, ALF needs to know whether Δ is empty or inhabited. As Δ can be any context, ALF has no information about whether Δ is empty or not and hence, it cannot unify. To solve this problem we use an auxiliary proposition called sl. In this proposition, we replace the assumption $[\Gamma,\ x\ :\ \alpha] +\!\!+ \Delta \vdash e : \beta$ by $\Sigma \vdash e : \beta$ and add the assumption $\Sigma = [\Gamma,\ x\ :\ \alpha] +\!\!+ \Delta$. In this way we can perform the induction on the derivation of $\Sigma \vdash e : \beta$ and we then analyse the form of Σ when it is needed.

sl $\in$ (D $\in$ Ctxt; =(S, $_{++}$(mk$_{ne_{ctxt}}$(G, x, A), D)); $\vdash$(S, e, B); $\vdash$(G, d, A)) $\vdash$($_{++}$(G, D), :=(x, d, e), B)
  sl([], refl, v$_{\vdash}$(h_3, h_4), h_2) $\equiv$ subst$_{=\vdash}$(=$_{same\text{-}var_{:=}}$, h_2)
  sl(:(D_1, _), refl, v$_{\vdash}$(h_3, h_4), h_2) $\equiv$
    subst$_{=\vdash}$(=$_{diff\text{-}var_{:=}}$(diff$_{fresh}$(D_1, h_4)), v$_{\vdash}$(dep$_{++_{valid}}$(D_1, h_3), dep$_{++_{fresh}}$(D_1, h_4)))
  sl([], refl, th$_{\vdash}$(h_3, h_4), h_2) $\equiv$ subst$_{=\vdash}$(=$_{:=_{free\_var}}$(h_3, h_4), h_4)
  sl(:(D_1, _), refl, th$_{\vdash}$(h_3, h_4), h_2) $\equiv$ th$_{\vdash}$(dep$_{++_{fresh}}$(D_1, h_3), sl(D_1, refl, h_4, h_2))
  sl(D, refl, $\lambda_{\downarrow\vdash}$($h_3$), h_2) $\equiv$
    subst$_{=\vdash}$(=$_{diff\text{-}\lambda_{:=}}$(diff$_{valid}$(valid$_{ctxt}$($h_3$))), $\lambda_{\downarrow\vdash}$(sl(mk$_{ne_{ctxt}}$($D, x_1, A_1$), refl, h_3, h_2)))
  sl(D, h, $._{\downarrow\vdash}$(h_3, h_4), h_2) $\equiv$ $._{\downarrow\vdash}$(sl(D, h, h_3, h_2), sl(D, h, h_4, h_2))
  sl(D, refl, fix$_{\vdash}$(h_3), h_2) $\equiv$
    subst$_{=\vdash}$(=$_{diff\text{-}fix_{:=}}$(diff$_{valid}$(valid$_{ctxt}$(h_3))), fix$_{\vdash}$(sl(mk$_{ne_{ctxt}}$(D, x_1, B), refl, h_3, h_2)))
  sl(D, refl, true$_{\vdash}$(h_3), h_2) $\equiv$ true$_{\vdash}$(dep$_{++_{valid}}$(D, h_3))
  sl(D, refl, false$_{\vdash}$(h_3), h_2) $\equiv$ false$_{\vdash}$(dep$_{++_{valid}}$(D, h_3))
  sl(D, h, if$_{\vdash}$(h_3, h_4, h_5), h_2) $\equiv$ if$_{\vdash}$(sl(D, h, h_3, h_2), sl(D, h, h_4, h_2), sl(D, h, h_5, h_2))
subst_lemma $\in$ ($\vdash$($_{++}$(mk$_{ne_{ctxt}}$(G, x, A), D), e, B); $\vdash$(G, d, A)) $\vdash$($_{++}$(G, D), :=(x, d, e), B)
  subst_lemma(h, h_1) $\equiv$ sl(D, refl, h, h_1)

A similar discussion to the one we presented here can be found both in [Bov95] and [Tas97]. For a more detailed explanation about the formalisation of this lemma, see [Tas97].

Subject Reduction Property. We present now the formalisation of the Subject Reduction property. The proof is made by pattern matching on the first

argument. For each case in the first argument, we perform pattern matching on the second argument.

$$
\begin{array}{l}
\mathrm{sub\_reduction} \in (\rightarrow(d, e);\ \vdash([], d, A))\ \vdash([], e, A) \\
\quad \mathrm{sub\_reduction}(.\rightarrow(h_2), .\vdash(h, h_3)) \equiv .\vdash(\mathrm{sub\_reduction}(h_2, h), h_3) \\
\quad \mathrm{sub\_reduction}(.\lambda\rightarrow, .\vdash(\lambda\vdash(h_1), h_2)) \equiv \mathrm{subst\_lemma}(h_1, h_2) \\
\quad \mathrm{sub\_reduction}(\mathrm{fix}\rightarrow, \mathrm{fix}\vdash(h)) \equiv \mathrm{subst\_lemma}(h, \mathrm{fix}\vdash(h)) \\
\quad \mathrm{sub\_reduction}(\mathrm{if}\rightarrow(h_2), \mathrm{if}\vdash(h, h_3, h_4)) \equiv \mathrm{if}\vdash(\mathrm{sub\_reduction}(h_2, h), h_3, h_4) \\
\quad \mathrm{sub\_reduction}(\mathrm{if}_{\mathrm{true}}\rightarrow, \mathrm{if}\vdash(h, h_2, h_3)) \equiv h_2 \\
\quad \mathrm{sub\_reduction}(\mathrm{if}_{\mathrm{false}}\rightarrow, \mathrm{if}\vdash(h, h_2, h_3)) \equiv h_3
\end{array}
$$

Well Typed Expressions Cannot Go Wrong. We conclude this section with the formalisation of the property that well typed expressions cannot go wrong. For that, we first need to formalise the definition of error expressions and the definition of expressions that go wrong.

$$
\begin{array}{l}
\mathrm{error}_{\mathrm{exp}} \in (\mathrm{Exp})\ \mathbf{Set} \\
\quad \mathrm{error}_{\mathrm{exp}} \equiv [h]\wedge(\neg(\exists(\mathrm{Exp}, [h_1]\rightarrow(h, h_1))), \neg(\mathrm{Can}(h)))
\end{array}
$$

$$
\begin{array}{l}
\mathrm{go}_{\mathrm{wrong}} \in (\mathrm{Exp})\ \mathbf{Set} \\
\quad \mathrm{go}_{\mathrm{wrong}} \equiv [h]\exists(\mathrm{Exp}, [h_1]\wedge(\rightarrow^*(h, h_1), \mathrm{error}_{\mathrm{exp}}(h_1)))
\end{array}
$$

Below, we prove that a well typed expression is either canonical or it reduces.

$$
\begin{array}{l}
\mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red} \in (\vdash([], e, A))\vee(\mathrm{Can}(e), \exists(\mathrm{Exp}, [h]\rightarrow(e, h))) \\
\quad \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(\lambda\vdash(h_1)) \equiv \vee_{\mathrm{Il}}(\mathrm{can}_{\mathrm{abs}}) \\
\quad \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(.\vdash(h_1, h_2)) \equiv \\
\qquad \mathbf{case}\ \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(h_1) \in \vee(\mathrm{Can}(d), \exists(\mathrm{Exp}, [h]\rightarrow(d, h)))\ \mathbf{of} \\
\qquad\quad \vee_{\mathrm{Il}}(c) \Rightarrow \mathbf{case}\ c \in \mathrm{Can}(d)\ \mathbf{of} \\
\qquad\qquad\qquad \mathrm{can}_{\mathrm{abs}} \Rightarrow \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(:=(x, e_1, e), .\lambda\rightarrow)) \\
\qquad\qquad\qquad \mathrm{can}_{\mathrm{true}} \Rightarrow \mathbf{case}\ h_1 \in \vdash([], \mathrm{true}, \rightarrow_{\mathrm{T}}(A_1, A))\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{end} \\
\qquad\qquad\qquad \mathrm{can}_{\mathrm{false}} \Rightarrow \mathbf{case}\ h_1 \in \vdash([], \mathrm{false}, \rightarrow_{\mathrm{T}}(A_1, A))\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{end} \\
\qquad\qquad \mathbf{end} \\
\qquad\quad \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(a, h)) \Rightarrow \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(.(a, e_1), .\rightarrow(h))) \\
\qquad \mathbf{end} \\
\quad \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(\mathrm{fix}\vdash(h_1)) \equiv \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(:=(x, \mathrm{fix}(x, e_1), e_1), \mathrm{fix}\rightarrow)) \\
\quad \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(\mathrm{true}\vdash(h_1)) \equiv \vee_{\mathrm{Il}}(\mathrm{can}_{\mathrm{true}}) \\
\quad \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(\mathrm{false}\vdash(h_1)) \equiv \vee_{\mathrm{Il}}(\mathrm{can}_{\mathrm{false}}) \\
\quad \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(\mathrm{if}\vdash(h_1, h_2, h_3)) \equiv \\
\qquad \mathbf{case}\ \mathrm{typed}_{\mathrm{prog}}\Rightarrow\mathrm{can}\vee\mathrm{red}(h_1) \in \vee(\mathrm{Can}(d), \exists(\mathrm{Exp}, [h]\rightarrow(d, h)))\ \mathbf{of} \\
\qquad\quad \vee_{\mathrm{Il}}(c) \Rightarrow \mathbf{case}\ c \in \mathrm{Can}(d)\ \mathbf{of} \\
\qquad\qquad\qquad \mathrm{can}_{\mathrm{abs}} \Rightarrow \mathbf{case}\ h_1 \in \vdash([], \lambda(x, e), \mathrm{bool})\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{end} \\
\qquad\qquad\qquad \mathrm{can}_{\mathrm{true}} \Rightarrow \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(e_1, \mathrm{if}_{\mathrm{true}}\rightarrow)) \\
\qquad\qquad\qquad \mathrm{can}_{\mathrm{false}} \Rightarrow \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(f, \mathrm{if}_{\mathrm{false}}\rightarrow)) \\
\qquad\qquad \mathbf{end} \\
\qquad\quad \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(a, h)) \Rightarrow \vee_{\mathrm{Ir}}(\exists_{\mathrm{I}}(\mathrm{if}(a, e_1, f), \mathrm{if}\rightarrow(h))) \\
\qquad \mathbf{end}
\end{array}
$$

The proof is made by pattern matching on the derivation of $\vdash e : A$. We only consider the first two equations here. The other equations are either straightforward or similar to the first two.

The first equation corresponds to the case where $\vdash e : A$ is proven by using the rule $\mathsf{Abs_{ts}}$. Then, e is the expression $\lambda x.e_1$ and h_1 is a proof of $[x : A] \vdash e_1 : B$. Thus, e is a canonical expression.

The second equation corresponds to the case where the rule $\mathsf{App_{ts}}$ is used. Here, e is of the form $(d\ \ e_1)$, h_1 is a proof of $\vdash d : A_1 \to A$, and h_2 is a proof of $\vdash e_1 : A_1$. Then, by the inductive hypothesis, we have a proof that either d is canonical or it reduces. By doing a case analysis on this result, we obtain two possibilities. The first possibility says that d is canonical being c a proof of that. Then, a case analysis on c shows that d can only be an abstraction due to its functional type. Thus, we can reduce the original expression e by using the rule $\mathsf{App\_Abs_{cs}}$. The second possibility establishes that there exists an expression a such that $d \longrightarrow a$, with h being a proof of that. Hence, by using the rule $\mathsf{App_{cs}}$ we can reduce the original expression $(d\ \ e_1)$ and obtain the expression $(a\ \ e_1)$.

Here, we present another auxiliary proposition.

$$\mathsf{absurd} \in (\vdash([], e, A);\ \neg(\mathsf{Can}(e));\ \neg(\exists(\mathsf{Exp}, [h]\to(e, h)))) \perp$$

For the formalisation of the lemma 2, we need another auxiliary proposition.

$$\mathsf{well}_{\mathsf{tp\_aux}\Rightarrow\perp} \in (\vdash([], d, A);\ \to^*(d, e);\ \mathsf{error}_{\mathsf{exp}}(e)) \perp$$
$$\mathsf{well}_{\mathsf{tp\_aux}\Rightarrow\perp}(h, \mathsf{refl}_{\to^*}, \wedge_{\mathrm{I}}(h_1, h_3)) \equiv \mathsf{absurd}(h, h_3, h_1)$$
$$\mathsf{well}_{\mathsf{tp\_aux}\Rightarrow\perp}(h, \mathsf{addstep}_{\to^*}(h_3, h_4), h_2) \equiv \mathsf{well}_{\mathsf{tp\_aux}\Rightarrow\perp}(\mathsf{sub\_reduction}(h_3, h), h_4, h_2)$$

The proof is made by pattern matching on the derivation of $d \longrightarrow^* e$.

The first equation correspond to the case where $d \longrightarrow^* e$ by using the rule $\mathsf{Refl_{cl}}$. Thus, $d = e$. Here, h is a proof of $\vdash d : A$, h_1 is a proof that there does not exist an expression f such that $e \longrightarrow f$ and h_3 is a proof that e is not canonical. Then, we apply the previous proposition and obtain the absurdity.

The second equation corresponds to the case where $d \longrightarrow e_1 \longrightarrow^* e$ by using the rule $\mathsf{AddStep_{cl}}$, for an expression e_1. Here, h is a proof of $\vdash d : A$, h_3 is a proof of $d \longrightarrow e_1$, h_4 is a proof of $e_1 \longrightarrow^* e$ and h_2 is a proof that e is an error expression. By Subject Reduction, we have that $\vdash e_1 : A$. Hence, by the inductive hypothesis we obtain the absurdity.

Now we introduce the formalisation of the lemma 2. In its conclusion we obtain a contradiction, which in type theory is formalised by the absurdity set.

$$\mathsf{well}_{\mathsf{tp}\wedge\mathsf{gw}\Rightarrow\perp} \in (\vdash([], e, A);\ \mathsf{go}_{\mathsf{wrong}}(e)) \perp$$
$$\mathsf{well}_{\mathsf{tp}\wedge\mathsf{gw}\Rightarrow\perp}(h, \exists_{\mathrm{I}}(a, \wedge_{\mathrm{I}}(h_1, h_3))) \equiv \mathsf{well}_{\mathsf{tp\_aux}\Rightarrow\perp}(h, h_1, h_3)$$

In this proof, we do pattern matching on the proof that e goes wrong and obtain that there exists an expression a such that $e \longrightarrow^* a$ and a is an error expression, with proofs h_1 and h_3 respectively. Using the auxiliary proposition that we introduced above, we obtain the absurdity.

Using this last proposition, we now formalise the main property of the paper, that is, that well typed expressions cannot go wrong.

$$\mathsf{well}_{\mathsf{tp}\Rightarrow\mathsf{cannot}_{\mathsf{gw}}} \in (\vdash([], e, A))\ \neg(\mathsf{go}_{\mathsf{wrong}}(e))$$
$$\mathsf{well}_{\mathsf{tp}\Rightarrow\mathsf{cannot}_{\mathsf{gw}}}(h) \equiv \supset_{\mathrm{I}}(\mathsf{well}_{\mathsf{tp}\wedge\mathsf{gw}\Rightarrow\perp}(h))$$

4 Conclusions

We think that the formalisation we have presented here is clear and can be understood without too much effort. This is mainly because the formalisation remains close to the informal presentation we gave of it. The pattern matching facility of ALF is a great help in this respect. In addition, once we had understood the results, their formalisation went smoothly and it did not take too much time.

We find that ALF is a nice tool to perform this kind of formalisations. It has a friendly interface and it is easy to use for those who have some knowledge of type theory. Since the possibility of performing pattern matching was introduced, the proofs have become simpler to do and easier to read than their equivalents using elimination rules. In our opinion, the major inconvenience with ALF is that ALF does not check well-foundedness of the recursive proofs.

Related Work. In [Bar92], Barendregt studies the Subject Reduction for the λ_calculus. In the proofs, he relies on what he calls *variable convention*, which says that bound and free variables are chosen such that they differ from each other. This convention 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. The way Barendregt uses the variable convention is, in our view, not formal.

Holmström also proves the Subject Reduction property for a language similar to ours (see [Hol83]). In some of his proofs, he shows that the conclusion of a theorem holds by informally manipulating the derivations in the type system.

In [Pol94], Pollack studies and formalises the Subject Reduction property for Pure Type Systems in the proof checker LEGO. In his work, he distinguishes between bound variables that he calls *variables* and free variables that he calls *parameters*. Parameters and variables are disjoint sets. These two sets lead him to have two substitutions and to change the usual typing rule for the abstraction for a rule where the bound variable is replaced by a completely fresh parameter.

In [MP91], there is a formalisation of the dynamic semantics and the type system of Mini-ML in the logic programming language Elf ([Pfe91]), which is founded upon the logical framework LF ([HHP87]). Although the informal presentation of the type system presented in the paper is similar to ours, the Elf formalisation is quite different to the ALF formalisation we present here. The formal counterpart of a set of Martin-Löf's type theory is called a type in LF. However, 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. Because types are not inductively defined in Elf, there is no way of formalising properties such as, for example, the Subject Reduction.

Tasistro (see [Tas97]) also performed a formalisation of the Subject Reduction for the same language as the one we work with here, but with a slightly different type system. Although the approach Tasistro follows is similar to the one presented in [Bov95], there are some improvements in his formalisation, which we followed to perform the formalisation we introduced here.

In [Bov95], when choosing between two expressions or when establishing the equality of two expression regarding the equality or inequality of the variables x and y, we used or_elimination on the proposition $x = y \vee x \neq y$. In Tasistro's formalisation, he uses functions similar to the ones presented in the subsection **Variables and Expressions** of section 3.3. The use of these functions allows us to perform the proofs using only pattern matching and hence, avoid the mixture of the pattern matching facility and elimination rules in our proofs. This fact makes the proofs simpler and more readable.

Another improvement was the introduction of an auxiliary proposition in order to prove the Substitution Lemma. This auxiliary proposition allows us to prove the lemma as desired but without changing the formulation of the lemma itself, as it is done in [Bov95].

We believe that the notion that well typed expression cannot go wrong was first introduced by Milner in [Mil78]. However, the approach presented there differs from the one we use here. In [Mil78], there exists a special semantic value called wrong which is shown to have no type. Then, it follows directly that well typed expressions cannot have the semantic value wrong. For our work, we followed the approach presented by Holmström in [Hol83].

To our knowledge, there have been no previous attempts to formalise the main property we presented here.

Further Work. The main extension to this work we are interested in is the addition of *type schemas* to our language of types. This addition will allow us to work with type inference instead of with type checking. Then, provided that an expression e has type, we can infer the *most general type scheme* for e. In [Mil78,Dam85], the algorithm $\mathcal{W}$ is presented. This algorithm performs a type inference for a language similar to ours, but which also has let expressions. What we want then is to formalise the algorithm $\mathcal{W}$ together with some of its properties, like the soundness and completeness properties.

Acknowledgements

I want to thank Johan Jeuring, Bengt Nordström and Björn von Sydow for their helpful comments on earlier versions of this paper. I would also like to thank Paula Severi for pointing out an error in one of the proofs.

References

[AGNvS94] T. Altenkirch, V. Gaspes, B. Nordström, and B. von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994. Available on the WWW `ftp://ftp.cs.chalmers.se/pub/users/alti/alf.ps.Z`.

[Bar92] H. P. Barendregt. Lambda calculi with types. In Abramsky Gabbai and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[Bov95] A. Bove. A machine-assisted proof of the subject reduction property for a small typed functional language. Master's thesis, Department of Computer Science, University of the Republic, Uruguay, November 1995. Technical Report INCO-95-06. Available on the WWW http://md.chalmers.se/~bove/Papers/master_thesis.ps.gz.

[Bov98] A. Bove. A machine-assisted proof that well typed expressions cannot go wrong. Technical report, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, 1998. Available on the WWW http://md.chalmers.se/~bove/Papers/wtecngw.ps.gz.

[Dam85] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Cimputer Science, University of Edinburgh, April 1985.

[Hen90] M. Hennessy. *The Semantics of Programming Languages : An Elementary Introduction using Structural Operational Semantics.* John Wiley & Sons, 1990.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Proceedings of the Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, June 1987.

[Hol83] S. Holmström. Polymorphic type systems : A proof - theoretic approach. Technical Report 6, University of Goteborg and Chalmers University of Technology, Sweden, September 1983.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[MP91] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in elf. In L. Hallnäs L.-H. Eriksson and P. Schroeder-Heister, editors, *Second International Workshop on Extensions of Logic Programming*, number 596 in Springer-Verlag Lecture Notes in Artificial Intelligence, Stockholm, Sweden, 1991.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Pfe91] F. Pfenning. Logic programming in the lf logical framework. In *Logical Frameworks.* Cambridge University Press. G. Huet and G. Plotkin Eds., 1991.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, September 1981.

[Pol94] R. Pollack. *The Theory of Lego A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[Tas97] A. Tasistro. Machine-assisted application of constructive type theory to the theory of functional programming languages. a first experiment using alf. in PhD thesis, 1997. Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden.