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

Ana Bove *

bove@cs.chalmers.se

May, 1998

Abstract

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. 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 (closed) expressions cannot go wrong*. First, we give all 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.

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

Contents

List of Figures

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. In general then, a proof of a theorem is a function that, given proofs of the hypothesis of the theorem, computes the proof of the thesis. In particular, 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. In other words, the formalisation of the relevant parts of the theory of programming languages gives implementations of these languages that are correct by construction.

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.

Here, we consider a small typed functional language and prove some properties about it, arriving at the property that establishes that *well typed (closed) expressions cannot go wrong.* This work can be seen as an extension of the work presented in [Bov95], where we presented a formalisation of the Subject Reduction property for the same language as the one considered here. However, in order to express the notion of going wrong, we work here with one step semantics and its reflexive and transitive closure instead of with long step semantics as in [Bov95]. This, however, is not a problem because, as we prove in appendix A, the value of an expression is the same regardless of which of the two semantics we choose for the evaluation.

In this paper, we first give all 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 (see [AGNvS94, Mag92, MN94]) for Martin-Löf's monomorphic type theory. We also use the pattern matching facility provided by ALF in our proofs, which in fact is not part of Martin-Löf's framework but which makes the proofs easier to carry out.

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.

1.1 About this Paper

This paper is organised as follows :

In section 2 we present the language we use. For this, we give its syntax, its dynamic semantics and a suitable type system for the language. Furthermore, we present several properties of the language, concluding with the main property we present in this paper : 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 presented in section 2, using Martin-Löf's type theory.

In section 4 we present some conclusions, related work and further work.

Finally, in appendix A we introduce the formalisation of the Evaluation semantics for the language we use, some auxiliary lemmas and a proof that the Evaluation semantics and the semantic relation $\rightarrow^*$ presented in section 2.2 are equivalent for our language.

2 Informal Presentation

In this section, we present the language we use. For this, we give its syntax, its dynamic semantics and a suitable type system for the language. Furthermore, we present several properties of the language, concluding with the main property we present in this paper : the property that shows that *well typed (closed) expressions cannot go wrong.*

For elements a and b in a given set $\mathcal{S}$, we write $a =_{\mathcal{S}} b$ and $a \neq_{\mathcal{S}} b$ for the equality and inequality, respectively, of the elements a and b. However, whenever the set $\mathcal{S}$ can be deduced from the context and no ambiguity can be generated from not explicitly subscripting it, we will not subscript it.

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 and are the same expressions as those considered in [Bov95, Tas97] : we have variables, abstractions, applications, fix points operators, boolean values and conditionals. In contrast to what is usual in functional languages, we do not allow local definitions here. This kind of expression is interesting if polymorphic type inference using type schemes is present, but such types are not considered in this paper. We define the expressions in *Exp* by means of its abstract syntax as follows :

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

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* or *program* 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 programs, and no evaluation is performed inside a binding operator (see figure 2 for the definition of the dynamic semantics) as is normal practice in functional programming. Thus, no capture can occur.

2.2 The Dynamic Semantics of the Language

In this section, we present the dynamic semantics of the language by describing how to evaluate an expression in *Exp*.

We give the dynamic semantics of the language in an operational style (for an introduction to operational semantics see [Plo81]). In contrast with the approach used in [Bov95, Tas97], where *Evaluation semantics* ([Hen90], also known as *Natural semantics* [NN92]) is used, here we use *Computational semantics* [Hen90], also known as *Structural Operational semantics*

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

Figure 1: Definition of the function $\_\,[\_\,/\_\,]$ over *Exp*

[Plo81, NN92]. The use of this dynamic semantics allows us to define those expressions that *go wrong*, whereas this is not possible using Evaluation semantics.

Evaluation semantics describes the relationship between the initial and the final state of an execution without giving any information about how this final state was obtained. For this reason, this semantics is sometimes referred to as a "long step relation". On the other hand, Computational semantics defines a "one step relation". While the final state in Evaluation semantics is an expression that cannot be reduced any further, this is not necessarily the case with Computational semantics. Thus, here, a computation consists of a sequence (possibly empty) 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. An irreducible expression is an expression that cannot be reduced any further. 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]).

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

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

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

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

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

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

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

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

Figure 2: Inductive Definition of the Computational Semantics of the Language and its Reflexive and Transitive Closure

There are also expressions with no value, such as the expression $\mathsf{fix}\ x\,.\,x$, which gives rise to an infinite computation.

Observe that for expressions of the form $(d\ \ e)$, $\mathsf{fix}\ x\,.\,e$ and $\mathsf{if}\ d\ \mathsf{then}\ e\ \mathsf{else}\ f$ there is at most one rule that can be applied at every moment. If the expression is an application, either we can reduce the leftmost subexpression or the leftmost subexpression is canonical (and thus, it cannot be reduced any further) and thus, we can reduce the whole expression at once. In each of these cases, there is only one possible rule to apply. If the expression is a fix point expression, there is only one rule that reduces the whole expression at once. Finally, if the expression is a conditional, as for the application case, we can either reduce the leftmost subexpression (and there is only one rule that reduces the leftmost subexpression) or the leftmost subexpression is canonical. In this last case, there are two rules where the leftmost subexpression of a conditional is canonical but in one of these two rules, the leftmost subexpression is the canonical expression true while in the other rule it is the canonical expression false. All this allows us to prove that the result of the relation $\longrightarrow$ is unique.

Proposition 1 (Functionality of $\longrightarrow$) *For d, e and f expressions, if $d \longrightarrow e$ and $d \longrightarrow f$ then $e = f$.*

We can relate the result of evaluating an expression both with the relation $\Rightarrow$ used in [Bov95, Tas97] and with the relation $\longrightarrow^*$ used here. As we have already mentioned, the semantics relation used both in [Bov95] and [Tas97] is the Evaluation semantics. Thus, $\Rightarrow$ is the "long step semantics relation" between two expressions : if we have that $d \Rightarrow e$, then e is the value of d.

The relation between $\Rightarrow$ and $\longrightarrow$ is as follows. For d and e expressions, if $d \Rightarrow e$ holds, then $d \longrightarrow^* e$ also holds. This statement can easily be proven by induction on the derivation of $d \Rightarrow e$. However, the converse is not necessarily true. As an example that it is not true, we have that $(\mathsf{if\ true\ then\ true\ else\ true}\ \ \lambda x.x) \longrightarrow^* (\mathsf{true}\ \ \lambda x.x)$ holds, but $(\mathsf{if\ true\ then\ true\ else\ true}\ \ \lambda x.x) \Rightarrow (\mathsf{true}\ \ \lambda x.x)$ does not hold. Thus, we cannot just prove that whenever $d \longrightarrow^* e$, then $d \Rightarrow e$ also holds, but instead, we can prove that if $d \longrightarrow^* e$ and e is a canonical expression, then $d \Rightarrow e$ also follows. To prove this, we first prove that if for expressions d, e and f we have $d \longrightarrow e$ and $e \Rightarrow f$, then we also have $d \Rightarrow f$. This can be proven by induction on the derivation of $d \longrightarrow e$. Then, to prove that if $d \longrightarrow^* e$ and e is a canonical expression, $d \Rightarrow e$ also holds, we proceed by induction on the derivation of $d \longrightarrow^* e$ and apply the previous result in the inductive step. In appendix A, we formally present the definition of $\Rightarrow$ and the lemmas showing the relation between the results obtained both with the relation $\Rightarrow$ and the relation $\longrightarrow^*$.

2.3 The Type System

In this section, we present the type system of the language. For this, we first introduce the types that the system uses and the notion of context.

2.3.1 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. We do not consider either type variables or type schemes here. We use α, β, γ, δ to denote elements in the set *Type*. We define it by means of its abstract syntax as follows :

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

In concrete types, we sometimes use parentheses to avoid ambiguity.

2.3.2 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 or subscripted), 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 need to 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 as usual, by induction on the second argument. Observe that $\Gamma +\!+ \Delta\ \mathsf{Valid}$ is satisfied when there is no variable x which is declared in both contexts.

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

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

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

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

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

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

Figure 3: Inductive Definition of *Ctxt* and predicates fresh and Valid

2.3.3 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 main proof of this paper, that is, the proof that *well typed (closed) expressions cannot go wrong.*

2.4.1 Subject Reduction Property

Before presenting the Subject Reduction property, we introduce some lemmas. The first lemma states that every time we use a context Γ for typing an expression e using the rules of the type system, then the context is a valid context.

Lemma 2 *Let Γ be a context, e an expression and α a type. If we can derive $\Gamma \vdash e : \alpha$, then Γ is a valid context, that is, we can derive Γ Valid.*

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

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

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

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

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

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

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

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

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

Figure 4: Inductive Definition of the Type System for *Exp*

Observe that if a context $[\Gamma,\ x\ :\ \alpha]$ is valid, then we can prove that x fresh Γ. Although this will not be used in the remainder of this section, it will be of great importance in the next section, where we formalise the proofs presented here.

An auxiliary lemma we use says that if a variable x is not free in a typed expression e (that is, an expression that has type), 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 a typed 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 $\Gamma \vdash e : \alpha$. For a detailed proof see [Bov95]. ■

The next lemma we need is known as the *Substitution Lemma* and states (informally) that if an expression e has a type under a context where the variable x (which possibly 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 as follows :

Theorem 4 (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. The proof is by induction on the derivation of $[\Gamma,\ x : \alpha] +\!+ \Delta \vdash e : \beta$. A detailed proof is presented both in [Bov95] and [Tas97]. ■

At first sight, and considering the definition of the substitution function presented in figure 1, it could seem strange that the Substitution Lemma works for *any* expression d that has type α under Γ, and not only for programs. In his paper [Tas97], Tasistro discusses why the Substitution Lemma works in this case. We briefly summarise his discussion here. Consider that the expression e in which we perform the substitution contains a binding. We might have for example that $[\Gamma,\ x : \alpha] +\!+ \Delta \vdash \lambda y.e' : \beta \to \gamma$. Now, as $\Gamma \vdash d : \alpha$, it might happen that y occurs free in d. Then, the free occurrences of y in d would be captured in $e\,[d/x]$, and the type of the expression after we perform the substitution would not be valid in general. As a concrete example, assume that $[y : \alpha,\ x : \alpha] \vdash \lambda y.x : \beta \to \alpha$ and $[y : \alpha] \vdash y : \alpha$. Then, as $(\lambda y.x)\,[y/x] = \lambda y.y$, by the Substitution Lemma we have that $[y : \alpha] \vdash \lambda y.y : \beta \to \alpha$. However, $\lambda y.y$ will have type $\beta \to \alpha$ only if α and β are the same type. Tasistro shows that if expression e is of the form $\lambda x.e'$ or $\mathsf{fix}\ x\,.\,e'$ and $[\Gamma,\ x : \beta] +\!+ \Delta \vdash e : \alpha$, then $\Gamma \vdash e : \alpha$ also holds. This, in turn, implies that $[y : \alpha,\ x : \alpha] \vdash \lambda y.x : \beta \to \alpha$ cannot be derivable because, if it would be derivable so would be $\vdash \lambda y.x : \beta \to \alpha$. However, it is easy to see that the latter cannot be derivable because x occurs free in the expression but it is not declared in the context.

Now, we can present the *Subject Reduction* property that says, in our case, the following :

Theorem 5 (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 $\vdash d : \alpha$. 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.

We consider here only the case where we use the $\mathsf{App\_Abs_{CS}}$ rule to obtain d $\longrightarrow$ e. The other cases are rather straightforward. For the case considered here, d has the form $(\lambda x.f\ \ g)$ and e has the form $f\,[g/x]$, for some variable x and some suitable expressions f and g. The expression d has type α if for some β, $\lambda x.f$ has type $\beta \to \alpha$ and g has type β (in both cases under the empty context). Now, $\lambda x.f$ can only have type $\beta \to \alpha$ if $[x : \beta] \vdash f : \alpha$ holds. Thus, if we apply the Substitution Lemma, we obtain $\vdash f\,[g/x] : \alpha$ as wanted. ■

2.4.2 The *Well Typed Expressions Cannot Go Wrong* Property

Before presenting the main property of this paper, we need to introduce two extra lemmas. The first one is formulated as follows :

Lemma 6 *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 $\vdash d : \alpha$. Again, notice that neither the variable rule nor the thinning rule can be applied in the derivation of $\vdash d : \alpha$.

If the rule that was applied in order to get $\vdash d : \alpha$ was the rule $\mathsf{Abs_{ts}}$, it means that the expression d has the form $\lambda x.f$ for a variable x and a suitable expression f. In this case, the expression is a canonical expression by definition.

If the rule applied was the rule $\mathsf{App_{ts}}$, then d is of the form $(f \quad g)$ 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 $(f \quad g)$ 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 $(f'' \quad g)$. In both cases, we show that the original expression can be reduced.

If the rule applied was the rule $\mathsf{Fix_{ts}}$, then d is of the form $\mathsf{fix}\ x\,.\,f$ for a variable x and a suitable expression f. Then, we can always apply the rule $\mathsf{Fix_{cs}}$ to obtain $f\,[\mathsf{fix}\ x\,.\,f/x]$ and thus show that the original expression can be reduced.

If the rule applied was the rule $\mathsf{True_{ts}}$ or the rule $\mathsf{False_{ts}}$, then d was either the canonical expression true or the canonical expression false respectively.

Finally, if the rule applied was the rule $\mathsf{If_{ts}}$, using a similar method to that used in the $\mathsf{App_{ts}}$ case, we show how the original expression can be reduced. ■

The second auxiliary lemma is as follows :

Lemma 7 *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.

Remember that, by definition, that d goes wrong means that there exists a value e of d that is an error expression. Remember also, that an error expression is an irreducible but non-canonical expression.

As we have assumed that d goes wrong, 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 $\mathsf{Refl_{cl}}$, 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 6 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 6.

If the rule applied was the rule $\mathsf{AddStep_{cl}}$, 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 8 (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

Although, as was stated in the introduction, this paper is intended mainly for those who already have some knowledge of type theory, and in particular of Martin-Löf's type theory, we present in this section a brief introduction to this theory to make the following sections more readable. For a more complete introduction to the subject, the reader can refer to [CNSvS94, NPS90].

Martin-Löf's type theory has a basic type and two type formers. The basic type is the type of sets, which we write **Set**. For each set S, the elements of S form a type. Given a type α and a family β of types over α, we can construct the function type from α to β. We write $a \in \alpha$ for "a is an object of type α".

Sets, elements of sets and functions are explained as follows :

- **Sets :** Sets are inductively defined. In other words, a set is determined by the rules that construct its elements, that is, the set's constructors. As mentioned above, 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 $\mathbf{El}(S)$. 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 $a \in \mathbf{El}(S)$.

- **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 $\beta(a)$ (actually, this is shorthand for $\beta(x := a)$). 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. In other words, a proposition is true if we can build an object of type P and it is false if the type P is not inhabitated. The way propositions are introduced allows us to identify propositions and sets, which is actually done in type theory. We write **Prop** to refer

to the type of propositions. As propositions are identified with sets, usually we write **Set** instead of **Prop**.

Then, if S and S' are sets, a predicate P over S is a function of type $(S)\,\mathbf{Prop}$ and a relation R on these sets is a function of type $(S;\ S')\,\mathbf{Prop}$. As propositions are inductively defined, as mentioned above, for each element a of type S and element b of type S', we have to give the rules that construct both the proofs of $P(a)$ and $R(a, b)$.

3.2 Brief Introduction to ALF

ALF (Another Logical Framework) is an interactive proof assistant for Martin-Löf's type theory. In this theory, theorems are identified with types and a proof is an object of the type, generally a function mapping proofs of the hypotheses into proofs of its thesis. ALF 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, Mag92, 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 $[x_1, \ldots, x_n]\,e$.

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. 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.

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.

Here we present two subsections. In the first one, we present general set formers and constructors. In the second subsection, we introduce the definition of the sets and implicit constants we need for our particular problem.

3.3.1 General Sets

In the proofs that we present in the next subsection, 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 :** 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. The set former is $\exists \in (A \in \mathbf{Set}; (A)\,\mathbf{Set})\,\mathbf{Set}$ and the only 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 propositional equality. Its only constructor states that an object is equal to itself. Together with the definition of the set, we prove the symmetry and transitivity properties, the congruence property with respect to functions of one, two and three arguments, and the substitutivity property. Below is the code for the mentioned definitions and properties.

$$
\begin{array}{l}
= \in (x, y \in A)\ \mathbf{Set} \\
\quad \mathrm{refl} \in =(x, x) \\
\mathrm{symm}_{=} \in (=(x, y)) =(y, x) \\
\quad \mathrm{symm}_{=}(\mathrm{refl}) \equiv \mathrm{refl} \\
\mathrm{trans}_{=} \in (=(x, y);\ =(y, z)) =(x, z) \\
\quad \mathrm{trans}_{=}(\mathrm{refl}, \mathrm{refl}) \equiv \mathrm{refl} \\
\mathrm{cong}_1 \in (f \in (A)\,B;\ =(a_1, a_2)) =(f(a_1), f(a_2)) \\
\quad \mathrm{cong}_1(f, \mathrm{refl}) \equiv \mathrm{refl} \\
\mathrm{cong}_2 \in (f \in (A; B)\,C;\ =(a_1, a_2);\ =(b_1, b_2)) =(f(a_1, b_1), f(a_2, b_2)) \\
\quad \mathrm{cong}_2(f, \mathrm{refl}, \mathrm{refl}) \equiv \mathrm{refl} \\
\mathrm{cong}_3 \in (f \in (A; B; C)\,D;\ =(a_1, a_2);\ =(b_1, b_2);\ =(c_1, c_2)) =(f(a_1, b_1, c_1), f(a_2, b_2, c_2)) \\
\quad \mathrm{cong}_3(f, \mathrm{refl}, \mathrm{refl}, \mathrm{refl}) \equiv \mathrm{refl}
\end{array}
$$

- **Not :** Represents the negation operator. This operator is actually defined as an abbreviation. We have that $\neg \in (A \in \mathbf{Set})\,\mathbf{Set}$ and it is defined as $\neg \equiv [A]\supset(A, \bot)$.
- **N :** Represents the set of natural numbers. Together with the definition of the set, we give below several theorems over the set N.

$$
\begin{array}{l}
\mathrm{N} \in \mathbf{Set} \\
\quad 0 \in \mathrm{N} \\
\quad \mathrm{s} \in (n \in \mathrm{N})\ \mathrm{N} \\
\mathrm{cong}_{\mathrm{s}} \in (=(m, n)) =(\mathrm{s}(m), \mathrm{s}(n)) \\
\quad \mathrm{cong}_{\mathrm{s}}(h) \equiv \mathrm{cong}_1(\mathrm{s}, h) \\
\mathrm{inj}_{\mathrm{s}} \in (h \in =(\mathrm{s}(m), \mathrm{s}(n))) =(m, n) \\
\quad \mathrm{inj}_{\mathrm{s}}(\mathrm{refl}) \equiv \mathrm{refl} \\
0{\neq}\mathrm{s} \in (=(0, \mathrm{s}(n))) \bot \\
\quad 0{\neq}\mathrm{s}(h) \equiv \mathbf{case}\ h \in =(0, \mathrm{s}(n))\ \mathbf{of} \\
\qquad\qquad \mathbf{end}
\end{array}
$$

$s{\neq}0 \in (=(s(n), 0)) \bot$
$\quad s{\neq}0(h) \equiv$ **case** $h \in =(s(n), 0)$ **of**
end
$s_n{\neq}s_m \in (\neg(=(n, m)); =(s(n), s(m))) \bot$
$\quad s_n{\neq}s_m(\supset_I(f), h_1) \equiv$ **case** $f(\text{inj}_s(h_1)) \in \bot$ **of**
end
$\neg 0{=}s \in \neg(=(0, s(n)))$
$\quad \neg 0{=}s \equiv \supset_I(0{\neq}s)$
$\neg s{=}0 \in \neg(=(s(n), 0))$
$\quad \neg s{=}0 \equiv \supset_I(s{\neq}0)$
$\neg s_n{=}s_m \in (\neg(=(n, m))) \neg(=(s(n), s(m)))$
$\quad \neg s_n{=}s_m(h) \equiv \supset_I(s_n{\neq}s_m(h))$
$N_{dec} \in (n, m \in N)\ \text{Dec}(=(n, m))$
$\quad N_{dec}(0, 0) \equiv \text{yes}(\text{refl})$
$\quad N_{dec}(0, s(n)) \equiv \text{no}(\neg 0{=}s)$
$\quad N_{dec}(s(n_1), 0) \equiv \text{no}(\neg s{=}0)$
$\quad N_{dec}(s(n_1), s(n)) \equiv$ **case** $N_{dec}(n_1, n) \in \text{Dec}(=(n_1, n))$ **of**
$\text{yes}(h) \Rightarrow \text{yes}(\text{cong}_s(h))$
$\text{no}(h) \Rightarrow \text{no}(\neg s_n{=}s_m(h))$
end

- **Or :** Represents the disjunction of two propositions. The set former is $\vee \in (A, B \in \textbf{Set})\,\textbf{Set}$ and its two set constructors are $\vee_{\text{Il}} \in (A)\vee(A, B)$ and $\vee_{\text{Ir}} \in (B)\vee(A, B)$.

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. It has two constructors, depending on whether a proposition or its negation can be proven. We give here the definition of this set.

$\text{Dec} \in (\textbf{Set})\,\textbf{Set}$
$\quad \text{yes} \in (P)\,\text{Dec}(P)$
$\quad \text{no} \in (\neg(P))\,\text{Dec}(P)$

3.3.2 Specific Sets and Implicit Constants

We can now focus on the formalisation of our functional language and its properties. For the formalisation, when choosing between two expressions or when establishing the equality of two expressions regarding the equality or inequality of two given variables, we follow Tasistro's approach ([Tas97]) instead of the one used in [Bov95].

Variables and Expressions

The first decision we have to make is how to formalise the set of variables of the language. Remember the importance in the informal presentation we have given to the decidability of the equality of variables. 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. Here we present the code for the set of variables :

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

Then, the formalisation of expressions and canonical expressions follows naturally :

```
Exp ∈ Set
    v ∈ (x ∈ Var) Exp
    λ ∈ (x ∈ Var; e ∈ Exp) Exp
    . ∈ (d, e ∈ Exp) Exp
    fix ∈ (x ∈ Var; e ∈ Exp) Exp
    true ∈ Exp
    false ∈ Exp
    if ∈ (d, e, f ∈ Exp) Exp
Can ∈ (Exp) Set
    can_abs ∈ Can(λ(x, e))
    can_false ∈ Can(false)
    can_true ∈ Can(true)
```

As it can already be seen in the informal presentation, 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 :

```
var_to_exp_dec ∈ (d, e ∈ Exp; Dec(=(x, y))) Exp
    var_to_exp_dec(d, e, yes(h1)) ≡ d
    var_to_exp_dec(d, e, no(h1)) ≡ e
```

We present now the definition of the substitution function :

```
:= ∈ (x ∈ Var; d, e ∈ Exp) Exp
    :=(x, d, v(x1)) ≡ var_to_exp_dec(d, v(x1), var_dec(x, x1))
    :=(x, d, λ(x1, e1)) ≡ var_to_exp_dec(λ(x1, e1), λ(x1, :=(x, d, e1)), var_dec(x, x1))
    :=(x, d, .(d1, e1)) ≡ .(:=(x, d, d1), :=(x, d, e1))
    :=(x, d, fix(x1, e1)) ≡ var_to_exp_dec(fix(x1, e1), fix(x1, :=(x, d, e1)), var_dec(x, x1))
    :=(x, d, true) ≡ true
    :=(x, d, false) ≡ false
    :=(x, d, if(d1, e1, f)) ≡ if(:=(x, d, d1), :=(x, d, e1), :=(x, d, f))
```

The following two lemmas help us to prove the equality of two expressions. In both lemmas, we make use of the decidability of the equality of variables.

```
:=_same_var ∈ (p ∈ Dec(=(x, x))) =(var_to_exp_dec(d, e, p), d)
    :=_same_var(yes(h)) ≡ refl
    :=_same_var(no(⊃I(f))) ≡ case f(refl) ∈ ⊥ of
                             end
:=_diff_var ∈ (¬(=(x, y)); p ∈ Dec(=(x, y))) =(var_to_exp_dec(d, e, p), e)
    :=_diff_var(⊃I(f), yes(h1)) ≡ case f(h1) ∈ ⊥ of
                                  end
    :=_diff_var(h, no(h1)) ≡ refl
```

The first lemma states that if we have two variables that are equal, then choosing between the expression d and e depending on the equality of the variables, is equal to the expression d. In the second equation on the proof of this lemma, $\supset_{\mathrm{I}}(f)$ is a proof of $\neg(=(x, x))$. Then, $f(\mathrm{refl})$ is a proof of $\bot$. By case analysis on the possible proofs of $\bot$, we obtain the desired result. Remember that, actually, $\bot$ has no proof.

The second lemma is similar, but it deals with the case where the variables are not equals.

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

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

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

We proceed with the proof of the functionality of the reduction of expressions.

$\text{unique–result}_{\text{red}} \in (\rightarrow(d, e); \rightarrow(d, f)) =(e, f)$ []
$\text{unique–result}_{\text{red}}(.\rightarrow(h_2), .\rightarrow(h)) \equiv \text{cong}_1([e].(e, e_1), \text{unique–result}_{\text{red}}(h_2, h))$
$\text{unique–result}_{\text{red}}(.\rightarrow(h_2), .\lambda\rightarrow) \equiv$ **case** $h_2 \in \rightarrow(\lambda(x, d_1), f_1)$ **of**
end
$\text{unique–result}_{\text{red}}(.\lambda\rightarrow, .\rightarrow(h)) \equiv$ **case** $h \in \rightarrow(\lambda(x, d_1), f_1)$ **of**
end
$\text{unique–result}_{\text{red}}(.\lambda\rightarrow, .\lambda\rightarrow) \equiv \text{refl}$
$\text{unique–result}_{\text{red}}(\text{fix}\rightarrow, \text{fix}\rightarrow) \equiv \text{refl}$
$\text{unique–result}_{\text{red}}(\text{if}\rightarrow(h_2), \text{if}\rightarrow(h)) \equiv \text{cong}_1([e]\text{if}(e, e_1, f_1), \text{unique–result}_{\text{red}}(h_2, h))$
$\text{unique–result}_{\text{red}}(\text{if}\rightarrow(h_2), \text{if}_{\text{true}}\rightarrow) \equiv$ **case** $h_2 \in \rightarrow(\text{true}, d')$ **of**
end
$\text{unique–result}_{\text{red}}(\text{if}\rightarrow(h_2), \text{if}_{\text{false}}\rightarrow) \equiv$ **case** $h_2 \in \rightarrow(\text{false}, d')$ **of**
end
$\text{unique–result}_{\text{red}}(\text{if}_{\text{true}}\rightarrow, \text{if}\rightarrow(h)) \equiv$ **case** $h \in \rightarrow(\text{true}, d')$ **of**
end
$\text{unique–result}_{\text{red}}(\text{if}_{\text{true}}\rightarrow, \text{if}_{\text{true}}\rightarrow) \equiv \text{refl}$
$\text{unique–result}_{\text{red}}(\text{if}_{\text{false}}\rightarrow, \text{if}\rightarrow(h)) \equiv$ **case** $h \in \rightarrow(\text{false}, d')$ **of**
end
$\text{unique–result}_{\text{red}}(\text{if}_{\text{false}}\rightarrow, \text{if}_{\text{false}}\rightarrow) \equiv \text{refl}$

The proof is made by pattern matching on the argument $\rightarrow(d, e)$. For each equation that we obtain after performing the pattern matching, we consider cases on the argument $\rightarrow(d, f)$. As a result of this, we obtain several equations where proofs like $h \in \rightarrow(\text{true},d')$ are established. We obtain the desired result by doing case analysis on these proofs. As there are no rules for reducing canonical expressions, there are no cases in each of the analyses. Thus,

we can conclude the desired result.

In appendix A we present the definition of the Evaluation semantics for the language and its equivalence with the relation $\rightarrow^*$ we presented above.

Types, Contexts and Properties

We first present the formalisation of the set of types. Remember that in this paper we only consider the type of the boolean expressions and the type of functions.

$$\begin{array}{l}\mathrm{Type} \in \mathbf{Set}\\ \quad \mathrm{bool} \in \mathrm{Type}\\ \quad \rightarrow_{\mathrm{T}} \in (\mathrm{Type};\ \mathrm{Type})\ \mathrm{Type}\end{array}$$

We introduce now the formalisation of declarations and contexts. As we have said in section 2.3.2, contexts are lists of declarations, where a declaration associates a type to a variable.

$$\begin{array}{l}\mathrm{Decl} \in \mathbf{Set}\\ \quad . \in (x \in \mathrm{Var};\ A \in \mathrm{Type})\ \mathrm{Decl}\\ \mathrm{Ctxt} \in \mathbf{Set}\\ \quad [] \in \mathrm{Ctxt}\\ \quad : \in (D \in \mathrm{Ctxt};\ d \in \mathrm{Decl})\ \mathrm{Ctxt}\end{array}$$

We need two functions over contexts. The first one takes a context, a variable and a type, and extends the given context with the declaration built from the variable and the type. The second function is the concatenation function over contexts, which is defined by recursion on its second argument.

$$\begin{array}{l}\mathrm{mk}_{\mathrm{nec}_{\mathrm{txt}}} \in (G \in \mathrm{Ctxt};\ x \in \mathrm{Var};\ A \in \mathrm{Type})\ \mathrm{Ctxt}\\ \quad \mathrm{mk}_{\mathrm{nec}_{\mathrm{txt}}}(G, x, A) \equiv :(G, .(x, A))\\ {}^{++} \in (G, D \in \mathrm{Ctxt})\ \mathrm{Ctxt}\\ \quad {}^{++}(G, []) \equiv G\\ \quad {}^{++}(G, :(D_1, d)) \equiv :({}^{++}(G, D_1), d)\end{array}$$

As we are interested only in those contexts where each variable is declared at most once, we define two predicates over contexts. The next two predicates are the formalisation of the predicates fresh and Valid presented in section 2.3.2.

$$\begin{array}{l}\mathrm{Fresh} \in (x \in \mathrm{Var};\ G \in \mathrm{Ctxt})\ \mathbf{Set}\\ \quad []_{\mathrm{fresh}} \in \mathrm{Fresh}(x, [])\\ \quad :_{\mathrm{fresh}} \in (\neg(=(x, y));\ \mathrm{Fresh}(x, G))\ \mathrm{Fresh}(x, \mathrm{mk}_{\mathrm{nec}_{\mathrm{txt}}}(G, y, A))\\ \mathrm{Valid} \in (G \in \mathrm{Ctxt})\ \mathbf{Set}\\ \quad []_{\mathrm{valid}} \in \mathrm{Valid}([])\\ \quad :_{\mathrm{valid}} \in (\mathrm{Valid}(G);\ \mathrm{Fresh}(x, G))\ \mathrm{Valid}(\mathrm{mk}_{\mathrm{nec}_{\mathrm{txt}}}(G, x, A))\end{array}$$

We need two auxiliary properties regarding the freshness of variables.

$$\begin{array}{l}\mathrm{dep}^{{}^{++}}{}_{\mathrm{fresh}} \in (D \in \mathrm{Ctxt};\ \mathrm{Fresh}(x, {}^{++}(:(G, d), D)))\ \mathrm{Fresh}(x, {}^{++}(G, D))\\ \quad \mathrm{dep}^{{}^{++}}{}_{\mathrm{fresh}}([], :_{\mathrm{fresh}}(h_1, h_2)) \equiv h_2\\ \quad \mathrm{dep}^{{}^{++}}{}_{\mathrm{fresh}}(:(D_1, \_), :_{\mathrm{fresh}}(h_1, h_2)) \equiv :_{\mathrm{fresh}}(h_1, \mathrm{dep}^{{}^{++}}{}_{\mathrm{fresh}}(D_1, h_2))\\ \mathrm{diff}_{\mathrm{fresh}} \in (D \in \mathrm{Ctxt};\ \mathrm{Fresh}(y, {}^{++}(\mathrm{mk}_{\mathrm{nec}_{\mathrm{txt}}}(G, x, A), D)))\ \neg(=(x, y))\\ \quad \mathrm{diff}_{\mathrm{fresh}}([], :_{\mathrm{fresh}}(\supset_{\mathrm{I}}(f), h_2)) \equiv \supset_{\mathrm{I}}([h]f(\mathrm{symm}_{=}(h)))\\ \quad \mathrm{diff}_{\mathrm{fresh}}(:(D_1, \_), :_{\mathrm{fresh}}(h_1, h_2)) \equiv \mathrm{diff}_{\mathrm{fresh}}(D_1, h_2)\end{array}$$

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

$\mathrm{diff}_{\mathrm{valid}} \in (\mathrm{Valid}(\mathrm{mk}_{\mathrm{nectxt}}(+\!+(\mathrm{mk}_{\mathrm{nectxt}}(G,x,A),D),y,B))) \neg(=(x,y))$
$\quad \mathrm{diff}_{\mathrm{valid}}(:_{\mathrm{valid}}(h_1,h_2)) \equiv \mathrm{diff}_{\mathrm{fresh}}(D,h_2)$
$\mathrm{dep}^{++}{}_{\mathrm{valid}} \in (D \in \mathrm{Ctxt};\ \mathrm{Valid}(+\!+(\mathrm{mk}_{\mathrm{nectxt}}(G,x,A),D))) \mathrm{Valid}(+\!+(G,D))$
$\quad \mathrm{dep}^{++}{}_{\mathrm{valid}}([],:_{\mathrm{valid}}(h_1,h_2)) \equiv h_1$
$\quad \mathrm{dep}^{++}{}_{\mathrm{valid}}(:(D_1,\_),:_{\mathrm{valid}}(h_1,h_2)) \equiv :_{\mathrm{valid}}(\mathrm{dep}^{++}{}_{\mathrm{valid}}(D_1,h_1),\mathrm{dep}^{++}{}_{\mathrm{fresh}}(D_1,h_2))$
$\mathrm{dep}_{\mathrm{valid}} \in (\mathrm{Valid}(\mathrm{mk}_{\mathrm{nectxt}}(G,x,A))) \mathrm{Valid}(G)$
$\quad \mathrm{dep}_{\mathrm{valid}}(:_{\mathrm{valid}}(h_1,h_2)) \equiv h_1$

This last property could have been defined as a derived property from the previous one instead of from the definition of validity.

The Type System

We present now the formalisation of the type system.

$\vdash \in (\mathrm{Ctxt};\ \mathrm{Exp};\ \mathrm{Type})\ \mathbf{Set}$
$\quad \mathrm{v}\vdash \in (\mathrm{Valid}(G);\ \mathrm{Fresh}(x,G)) \vdash(\mathrm{mk}_{\mathrm{nectxt}}(G,x,A),\mathrm{v}(x),A)$
$\quad \mathrm{th}\vdash \in (\mathrm{Fresh}(x,G);\ \vdash(G,e,B)) \vdash(\mathrm{mk}_{\mathrm{nectxt}}(G,x,A),e,B)$
$\quad \lambda\vdash \in (\vdash(\mathrm{mk}_{\mathrm{nectxt}}(G,x,A),e,B)) \vdash(G,\lambda(x,e),\rightarrow_{\mathrm{T}}(A,B))$
$\quad .\vdash \in (\vdash(G,d,\rightarrow_{\mathrm{T}}(A,B));\ \vdash(G,e,A)) \vdash(G,.(d,e),B)$
$\quad \mathrm{fix}\vdash \in (\vdash(\mathrm{mk}_{\mathrm{nectxt}}(G,x,A),e,A)) \vdash(G,\mathrm{fix}(x,e),A)$
$\quad \mathrm{true}\vdash \in (\mathrm{Valid}(G)) \vdash(G,\mathrm{true},\mathrm{bool})$
$\quad \mathrm{false}\vdash \in (\mathrm{Valid}(G)) \vdash(G,\mathrm{false},\mathrm{bool})$
$\quad \mathrm{if}\vdash \in (\vdash(G,d,\mathrm{bool});\ \vdash(G,e,A);\ \vdash(G,f,A)) \vdash(G,\mathrm{if}(d,e,f),A)$

The next property follows immediately.

$\mathrm{subst}{=}\vdash \in (=(d,e);\ \vdash(G,e,A)) \vdash(G,d,A)$
$\quad \mathrm{subst}{=}\vdash(\mathrm{refl},h_1) \equiv h_1$

Substitution Lemma

As in the informal presentation of the Substitution Lemma in section 2.4.1, we need some auxiliary properties in order to prove the lemma. From now on, to make the explanations more understandable, we allow ourselves to be more informal when explaining the formalisations.

The first property we need establishes that if a context is used to derive that an expression has a certain type, then the context is valid.

$\mathrm{valid}_{\mathrm{ctxt}} \in (\vdash(G,e,A))\ \mathrm{Valid}(G)$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(\mathrm{v}\vdash(h_1,h_2)) \equiv :_{\mathrm{valid}}(h_1,h_2)$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(\mathrm{th}\vdash(h_1,h_2)) \equiv :_{\mathrm{valid}}(\mathrm{valid}_{\mathrm{ctxt}}(h_2),h_1)$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(\lambda\vdash(h_1)) \equiv \mathrm{dep}_{\mathrm{valid}}(\mathrm{valid}_{\mathrm{ctxt}}(h_1))$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(.\vdash(h_1,h_2)) \equiv \mathrm{valid}_{\mathrm{ctxt}}(h_2)$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(\mathrm{fix}\vdash(h_1)) \equiv \mathrm{dep}_{\mathrm{valid}}(\mathrm{valid}_{\mathrm{ctxt}}(h_1))$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(\mathrm{true}\vdash(h_1)) \equiv h_1$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(\mathrm{false}\vdash(h_1)) \equiv h_1$
$\quad \mathrm{valid}_{\mathrm{ctxt}}(\mathrm{if}\vdash(h_1,h_2,h_3)) \equiv \mathrm{valid}_{\mathrm{ctxt}}(h_1)$

The proof is performed by pattern matching on the derivation of $G \vdash e : A$.

The first equation corresponds to the case where $G \vdash e : A$ is proven by using the rule $\mathsf{Var_{ts}}$. Then, we have that $[G_1,\ x : A] \vdash x : A$. Here, h_1 is a proof that G_1 is valid and h_2 is a proof that x is fresh in G_1. Using the proof constructor for validity of non-empty context, we can easily have a proof that $[G_1,\ x : A]$ is valid.

The second equation corresponds to the case where the rule $\mathsf{Th_{ts}}$ is used. Then , we have that $[G_1,\ x : A_1] \vdash e : A$. Here, h_1 is a proof that x is fresh in G_1 and h_2 is a proof of

$G_1 \vdash e : A$. The inductive hypothesis is given by the recursive call $\mathsf{valid}_{\mathsf{ctxt}}(h_2)$, and it gives us a proof that G_1 is valid. As in the previous equation, we can easily now have a proof that $[G_1,\ x : A_1]$ is valid.

The next equation corresponds to the use of the rule $\mathsf{Abs}_{\mathsf{ts}}$. We have that $G \vdash \lambda x.e_1 : A_1 \to B$. Here, h_1 is a proof of $[G,\ x : A_1] \vdash e_1 : B$. By the inductive hypothesis, we obtain that the context $[G,\ x : A_1]$ is valid. Now, using the function $\mathsf{dep}_{\mathsf{valid}}$ already explained, we obtain that the context G is valid.

The following equation corresponds to the case where we have $G \vdash (d\ \ e_1) : A$, by using the rule $\mathsf{App}_{\mathsf{ts}}$. Here, h_1 is a proof of $G \vdash d : A_1 \to A$ and h_2 is a proof of $G \vdash e_1 : A_1$. By the inductive hypothesis on either h_1 or h_2, we obtain that G is valid.

The next equation is similar to the third one. The other two following equations are straightforward. In these two last equations, h_1 is a proof that G is valid. The very last equation is similar to the fourth one.

The next property we present here formalises the lemma 3 of section 2.4.1. When formalising this lemma, we followed Tasistro's approach. For a detailed explanation of the formalisation, please refer to [Tas97]. One should not be confused by the difference in names or in the order of the arguments between our proof and that of Tasistro.

$$=_{:=\text{free\_var}} \in (\text{Fresh}(x, G);\ \vdash(G, e, A)) = (:=(x, d, e), e)$$
$$=_{:=\text{free\_var}}(:_{\text{fresh}}(h_1, h_4), \text{v}\vdash(h_2, h_3)) \equiv =_{\text{diff-var}_{:=}}(h_1)$$
$$=_{:=\text{free\_var}}(:_{\text{fresh}}(h_1, h_4), \text{th}\vdash(h_2, h_3)) \equiv =_{:=\text{free\_var}}(h_4, h_3)$$
$$=_{:=\text{free\_var}}(h, \lambda\vdash(h_2)) \equiv$$
$$\textbf{case}\ \text{var}_{\text{dec}}(x, x_1) \in \text{Dec}(=(x, x_1))\ \textbf{of}$$
$$\text{yes}(h_1) \Rightarrow =_{\text{same-}\lambda_{:=}}(h_1)$$
$$\text{no}(h_1) \Rightarrow \text{trans}_{=}(=_{\text{diff-}\lambda_{:=}}(h_1), \text{cong}_1(\lambda(x_1), =_{:=\text{free\_var}}(:_{\text{fresh}}(h_1, h), h_2)))$$
$$\textbf{end}$$
$$=_{:=\text{free\_var}}(h, .\vdash(h_2, h_3)) \equiv \text{cong}_2(., =_{:=\text{free\_var}}(h, h_2), =_{:=\text{free\_var}}(h, h_3))$$
$$=_{:=\text{free\_var}}(h, \text{fix}\vdash(h_2)) \equiv$$
$$\textbf{case}\ \text{var}_{\text{dec}}(x, x_1) \in \text{Dec}(=(x, x_1))\ \textbf{of}$$
$$\text{yes}(h_1) \Rightarrow =_{\text{same-fix}_{:=}}(h_1)$$
$$\text{no}(h_1) \Rightarrow \text{trans}_{=}(=_{\text{diff-fix}_{:=}}(h_1), \text{cong}_1(\text{fix}(x_1), =_{:=\text{free\_var}}(:_{\text{fresh}}(h_1, h), h_2)))$$
$$\textbf{end}$$
$$=_{:=\text{free\_var}}(h, \text{true}\vdash(h_2)) \equiv \text{refl}$$
$$=_{:=\text{free\_var}}(h, \text{false}\vdash(h_2)) \equiv \text{refl}$$
$$=_{:=\text{free\_var}}(h, \text{if}\vdash(h_2, h_3, h_4)) \equiv \text{cong}_3(\text{if}, =_{:=\text{free\_var}}(h, h_2), =_{:=\text{free\_var}}(h, h_3), =_{:=\text{free\_var}}(h, h_4))$$

We introduce now the formalisation of the Substitution Lemma. As mentioned in section 2.4.1, we perform the proof of the lemma by induction on the derivation of $[\Gamma,\ x : \alpha] +\!\!+ \Delta \vdash e : \beta$. The problem now is how to formalise this induction in ALF. A similar discussion to the one we will present here can be found both in [Bov95] and [Tas97].

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 $\Gamma' +\!\!+ \Delta$. For performing the induction in our particular case, ALF tries to unify $[\Gamma,\ x : \alpha] +\!\!+ \Delta \vdash e : \beta$ with the conclusion of each rule in the type system, but this unification involves non trivial unification. In the cases of the rules $\mathsf{Var}_{\mathsf{ts}}$ and $\mathsf{Th}_{\mathsf{ts}}$, ALF has to unify $[\Gamma,\ x : \alpha] +\!\!+ \Delta$ with $[\Gamma',\ y : \beta]$. For unifying these two contexts, ALF needs to know whether Δ is empty or inhabited, because we defined the concatenation function by induction on its second argument. As in these two cases 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 we 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.

A good point of this solution is that it preserves the structure of the informal proof of the Substitution Lemma. For another possible solution to the problem see [Bov95].

For a more detailed explanation about the formalisation of this lemma, see [Tas97]. Note that, because of the difference between our type system and the one used by Tasistro, our formalisation contains different proofs of freshness than those used by Tasistro. Besides, we also need proofs of validity of contexts, while Tasistro does not.

sl ∈ (D ∈ Ctxt; =(S, ++(mk$_{ne\,ctxt}$(G, x, A), D)); |–(S, e, B); |–(G, d, A)) |–(++(G, D), :=(x, d, e), B)
sl([], refl, v|–(h_3, h_4), h_2) ≡ subst$_{=|-(=_{same\text{–}var;=}}$, h_2)
sl(:(D_1, _), refl, v|–(h_3, h_4), h_2) ≡
subst$_{=|-(=_{diff\text{–}var;=}(\mathrm{diff}_{fresh}(D_1, h_4)),\ v|-(\mathrm{dep}^{++}{}_{valid}(D_1, h_3),\ \mathrm{dep}^{++}{}_{fresh}(D_1, h_4)))}$
sl([], refl, th|–(h_3, h_4), h_2) ≡ subst$_{=|-(=_{:=_{free_var}}(h_3, h_4)}$, h_4)
sl(:(D_1, _), refl, th|–(h_3, h_4), h_2) ≡ th|–(dep$^{++}{}_{fresh}$(D_1, h_3), sl(D_1, refl, h_4, h_2))
sl(D, refl, λ|–(h_3), h_2) ≡
subst$_{=|-(=_{diff\text{–}\lambda;=}(\mathrm{diff}_{valid}(\mathrm{valid}_{ctxt}(h_3))),\ \lambda|-(\mathrm{sl}(\mathrm{mk}_{ne\,ctxt}(D, x_1, A_1),\ \mathrm{refl}, h_3, h_2)))}$
sl(D, h, .|–(h_3, h_4), h_2) ≡ .|–(sl(D, h, h_3, h_2), sl(D, h, h_4, h_2))
sl(D, refl, fix|–(h_3), h_2) ≡
subst$_{=|-(=_{diff\text{–}fix;=}(\mathrm{diff}_{valid}(\mathrm{valid}_{ctxt}(h_3))),\ \mathrm{fix}|-(\mathrm{sl}(\mathrm{mk}_{ne\,ctxt}(D, x_1, B),\ \mathrm{refl}, h_3, h_2)))}$
sl(D, refl, true|–(h_3), h_2) ≡ true|–(dep$^{++}{}_{valid}$(D, h_3))
sl(D, refl, false|–(h_3), h_2) ≡ false|–(dep$^{++}{}_{valid}$(D, h_3))
sl(D, h, if|–(h_3, h_4, h_5), h_2) ≡ if|–(sl(D, h, h_3, h_2), sl(D, h, h_4, h_2), sl(D, h, h_5, h_2))
subst_lemma ∈ (|–(++(mk$_{ne\,ctxt}$(G, x, A), D), e, B); |–(G, d, A)) |–(++(G, D), :=(x, d, e), B)
subst_lemma(h, h_1) ≡ sl(D, refl, h, h_1)

Subject Reduction Property

We present now the formalisation of the Subject Reduction property.

sub_reduction ∈ (→(d, e); |–([], d, A)) |–([], e, A)
sub_reduction(.→(h_2), .|–(h, h_3)) ≡ .|–(sub_reduction(h_2, h), h_3)
sub_reduction(.λ→, .|–(λ|–(h_1), h_2)) ≡ subst_lemma(h_1, h_2)
sub_reduction(fix→, fix|–(h)) ≡ subst_lemma(h, fix|–(h))
sub_reduction(if→(h_2), if|–(h, h_3, h_4)) ≡ if|–(sub_reduction(h_2, h), h_3, h_4)
sub_reduction(if$_{true}$→, if|–(h, h_2, h_3)) ≡ h_2
sub_reduction(if$_{false}$→, if|–(h, h_2, h_3)) ≡ h_3

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.

The first equation corresponds to the case where $d \longrightarrow e$ is proven by using the rule $\mathsf{App_{cs}}$. Then, we have that $(d_1\ \ e_1) \longrightarrow (f\ \ e_1)$. Here, h_2 is a proof of $d_1 \longrightarrow f$, h is a proof of $\vdash d_1 : A_1 \ \rightarrow\ A$ and h_3 is a proof of $\vdash e_1 : A_1$. Then, by the inductive hypothesis, we have that $\vdash f : A_1 \ \rightarrow\ A$. Thus, using the rule $\mathsf{App_{ts}}$ we obtain that $\vdash (f\ \ e_1) : A$.

The second equation corresponds to the use of the rule $\mathsf{App\_Abs_{cs}}$. We then have that $(\lambda x.d_1\ \ e_1) \longrightarrow d_1\,[e_1/x]$. Here, h_1 is a proof of $[x\ :\ A_1] \vdash d_1 : A$ and h_2 is a proof of $\vdash e_1 : A_1$. Thus, by the Substitution Lemma, we have that $\vdash d_1\,[e_1/x] : A$.

The third equation corresponds to the case where the rule $\mathsf{Fix_{cs}}$ is used. Then, we have that $\mathsf{fix}\ x\,.\,e_1 \longrightarrow e_1\,[\mathsf{fix}\ x\,.\,e_1/x]$. Here, h is a proof of $[x\ :\ A] \vdash e_1 : A$. Once again, by the

Substitution Lemma, we have that $\vdash e_1\,[\mathsf{fix}\ x\,.\,e_1/x] : A$.

In the fourth equation we have that $\mathsf{if}\ d_1\ \mathsf{then}\ e_1\ \mathsf{else}\ f \longrightarrow \mathsf{if}\ d'\ \mathsf{then}\ e_1\ \mathsf{else}\ f$, by using the rule $\mathsf{If}_{\mathsf{cs}}$. Here, h_2 is a proof of $d_1 \longrightarrow d'$, h is a proof of $\vdash d_1 : \mathsf{Bool}$, h_3 is a proof of $\vdash e_1 : A$ and h_4 is a proof of $\vdash f : A$. Then, by the inductive hypothesis, we have that $\vdash d' : \mathsf{Bool}$. Hence, by the rule $\mathsf{If}_{\mathsf{ts}}$ we obtain that $\vdash \mathsf{if}\ d'\ \mathsf{then}\ e_1\ \mathsf{else}\ f : A$ as desired.

The next equation corresponds to the use of the rule $\mathsf{If\_True}_{\mathsf{cs}}$ when proving $d \longrightarrow e$. Then, we have that $\mathsf{if\ true\ then}\ e\ \mathsf{else}\ f \longrightarrow e$. Here, h is a proof of $\vdash \mathsf{true} : \mathsf{Bool}$, h_2 is a proof of $\vdash e : A$ and h_3 is a proof of $\vdash f : A$. Then, h_2 is the desired proof.

The last equation is similar to the previous one.

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.

```
error_exp ∈ (Exp) Set
    error_exp ≡ [h]∧(¬(∃(Exp, [h1]→(h, h1))), ¬(Can(h)))
go_wrong ∈ (Exp) Set
    go_wrong ≡ [h]∃(Exp, [h1]∧(→*(h, h1), error_exp(h1)))
```

Following the development we presented in section 2.4.2, we introduce now the formalisation of lemma 6, where we prove that a well typed expression is either canonical or it reduces.

```
typed_prog⇒can∨red ∈ (⊢([], e, A)) ∨(Can(e), ∃(Exp, [h]→(e, h)))
    typed_prog⇒can∨red(λ⊢(h1)) ≡ ∨I1(can_abs)
    typed_prog⇒can∨red(.⊢(h1, h2)) ≡
        case typed_prog⇒can∨red(h1) ∈ ∨(Can(d), ∃(Exp, [h]→(d, h))) of
            ∨I1(c) ⇒ case c ∈ Can(d) of
                          can_abs ⇒ ∨Ir(∃I(:=(x, e1, e), .λ→))
                          can_true ⇒ case h1 ∈ ⊢([], true, →T(A1, A)) of
                                     end
                          can_false ⇒ case h1 ∈ ⊢([], false, →T(A1, A)) of
                                      end
                      end
            ∨Ir(∃I(a, h)) ⇒ ∨Ir(∃I(.(a, e1), .→(h)))
        end
    typed_prog⇒can∨red(fix⊢(h1)) ≡ ∨Ir(∃I(:=(x, fix(x, e1), e1), fix→))
    typed_prog⇒can∨red(true⊢(h1)) ≡ ∨I1(can_true)
    typed_prog⇒can∨red(false⊢(h1)) ≡ ∨I1(can_false)
    typed_prog⇒can∨red(if⊢(h1, h2, h3)) ≡
        case typed_prog⇒can∨red(h1) ∈ ∨(Can(d), ∃(Exp, [h]→(d, h))) of
            ∨I1(c) ⇒ case c ∈ Can(d) of
                          can_abs ⇒ case h1 ∈ ⊢([], λ(x, e), bool) of
                                    end
                          can_true ⇒ ∨Ir(∃I(e1, iftrue→))
                          can_false ⇒ ∨Ir(∃I(f, iffalse→))
                      end
            ∨Ir(∃I(a, h)) ⇒ ∨Ir(∃I(if(a, e1, f), if→(h)))
        end
```

The proof is made by pattern matching on the derivation of $\vdash e : A$.

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 $\vdash d : A_1 \rightarrow A$, 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 $(a\ \ e_1)$.

The third equation corresponds to the use of the rule $\mathsf{Fix_{ts}}$. Here, the expression e is of the form $\mathsf{fix}\ x\,.\,e_1$. As the rule $\mathsf{Fix_{cs}}$ says, we can always reduce e and obtain $e_1\,[\mathsf{fix}\ x\,.\,e_1/x]$.

The following two equations are straightforward and similar to the first one. The very last equation is similar to the second one.

Before the formalisation of the main property of this paper, we need another auxiliary proposition whose formalisation should, by now, be easy to understand without further explanations.

```
absurd ∈ (⊢([], e, A); ¬(Can(e)); ¬(∃(Exp, [h]→(e, h)))) ⊥
    absurd(h, ⊃I(f), ⊃I(f1)) ≡
        case typed_prog⇒can∨red(h) ∈ ∨(Can(e), ∃(Exp, [h']→(e, h'))) of
            ∨Il(a) ⇒ case f(a) ∈ ⊥ of
                     end
            ∨Ir(b) ⇒ case f1(b) ∈ ⊥ of
                     end
        end
```

For the formalisation of the lemma 7, we need yet another auxiliary proposition.

```
well_tp_aux⇒⊥ ∈ (⊢([], d, A); →*(d, e); error_exp(e)) ⊥
    well_tp_aux⇒⊥(h, refl→*, ∧I(h1, h3)) ≡ absurd(h, h3, h1)
    well_tp_aux⇒⊥(h, addstep→*(h3, h4), h2) ≡ well_tp_aux⇒⊥(sub_reduction(h3, h), h4, h2)
```

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 $\vdash d : A$ (or what it is the same, h is a proof of $\vdash e : A$), 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 a canonical expression. 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 7 presented in section 2.4.2. In the conclusion of this lemma we obtain a contradiction, which in type theory is formalised by the absurdity set.

```
well_tp∧gw⇒⊥ ∈ (⊢([], e, A); go_wrong(e)) ⊥
    well_tp∧gw⇒⊥(h, ∃I(a, ∧I(h1, h3))) ≡ well_tp_aux⇒⊥(h, h1, h3)
```

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 can now formalise the main property of the paper, that is, we can formalise that well typed expressions cannot go wrong. We do this by applying the constructor of implications to the result of the previous lemma. Remember that $\neg(P)$ is defined in Martin-Löf's type theory as $(P)\,\bot$.

$$\mathrm{well}_{\mathrm{tp}}{\Rightarrow}\mathrm{cannot}_{\mathrm{gw}} \in (|\text{–}([], e, A))\ \neg(\mathrm{go}_{\mathrm{wrong}}(e))$$
$$\mathrm{well}_{\mathrm{tp}}{\Rightarrow}\mathrm{cannot}_{\mathrm{gw}}(h) \equiv \supset_{\mathrm{I}}(\mathrm{well}_{\mathrm{tp}}{\wedge}\mathrm{gw}{\Rightarrow}\bot(h))$$

4 Conclusions

We have presented here some well known results about a small typed functional language. We have first introduced the results in an informal way and then, we have presented and explained the formalisation of these results in type theory.

We think that the formalisation we have performed 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. However, as we have already mentioned in [Bov95], ALF needs some improvements. In our opinion, the major improvement that should be made in ALF is to enforce well-foundedness of the recursive proofs. In the current version of ALF, any verification of well-foundedness of the recursive proofs should be done manually. Although this can be easy in some cases, it is always tedious and occasionally it might lead to errors because proofs that one thinks are well-founded actually are not.

4.1 Related Work

In [Bov95], we already commented on several works where the Subject Reduction property has been studied. We briefly summarise the discussion here.

In [Bar92], Barendregt studies the Subject Reduction property for the λ_calculus, where the dynamic semantics is given by the β_reduction rules. 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 of expressions similar to ours (see [Hol83]). 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 [Pol94], Pollack studies and formalises the Subject Reduction property for Pure Type Systems (PTS) in the proof checker LEGO. 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. 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 this way, he captures the essence of α_conversion where the name of bound variables does not matter. Notice that we can view this method as a formalisation of the variable convention presented in [Bar92].

Another difference between our presentation and Pollack's is the validity of contexts. As an optimisation, 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 [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. Thus, for instance, our set *Exp* would be declared as a type. 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. For instance, the ML abstraction is formalised as a function with type $(exp \rightarrow exp) \rightarrow exp$, 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 formalisation of the type system's contexts in the informal presentation become contexts in the meta-language Elf, so there is no need to formalise the notion of contexts. Because types are not inductively defined in Elf, there is no way of formalising properties such as, for example, the Subject Reduction. Instead, the paper presents 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 internalisation* of the proof of the property.

Tasistro (see [Tas97]) also performed a formalisation of the Subject Reduction property for the same language as the one we work with here, but with a slightly different type system. The type system he defines prevents us for declaring more than once in the context all those variables that occur in the expression. However, for all those variables that do not occur in the expression, there is no restriction on the number of times they can be declared in the context. In our system, because of the condition of validity imposed in some of the rules, no variable can be declared more than once. For example, we can derive $[z : \beta, z : \gamma] \vdash \lambda x.x : \alpha \rightarrow \alpha$ in his type system, but not in ours. However, this is not an important difference because it can be proven in both systems that, if $\Gamma \vdash e : \alpha$ then there exists Γ' included in Γ such that $\Gamma' \vdash e : \alpha$ and Γ' has declarations only for those variables that occur free in the expression e.

Although the approach Tasistro follows is similar to the one presented in [Bov95], there are some important improvements in the formalisation he presents, which we followed to perform the formalisation we introduced here.

One of these improvements was already mentioned at the beginning of section 3.3.2. 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 3.3.2. The use of these functions allows us to perform the proofs using only pattern matching and hence, to avoid the mixture of the pattern matching facility and elimination rules in our proofs. This makes the proofs simpler and more readable.

Another improvement was the introduction of an auxiliary proposition in order to prove the Substitution Lemma. The need for this proposition was already explained when we

presented the formalisation of 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.

4.2 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 extend our work in two directions :

- We can add polymorphic expressions of the form let $x = d$ in e. These expressions allow the definition of local declarations and they have been shown to be very useful when using a functional programming. We can introduce let expressions in a type system without type schemes as it is done in [Hol83], where a let expression is typed using substitutions. However, we believe that it is more appropriate to introduce this kind of expressions in the presence of type schemes as is done in [CDDK86, DM82].

- We can 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. The most general type scheme for an expression e is a type scheme from which all types that can be derived for e are instances.

In fact, these two directions can be joined in one. In [Mil78, DM82], 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. A more detailed work on algorithm $\mathcal{W}$ can be found in [Dam85].

A Formalisation of the Evaluation Semantics and its Equivalence with the Relation $\rightarrow^*$ for our Language

We present here the formalisation of the Evaluation semantics for the language we use, some auxiliary lemmas and a proof that the Evaluation semantics and the relation $\rightarrow^*$ are equivalent for our language.

We present below the formalisation of the Evaluation semantics for the language. For an informal presentation of this definition for the same language, see [Bov95] or [Tas97].

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

Before proving the equivalence of both semantics for our language, we introduce some auxiliary lemmas. The first lemma states the congruence of $\rightarrow^*$ with respect to applications. The second lemma states the congruence of the same relation, but with respect to conditional expressions. The third lemma proves that the transitivity of $\rightarrow^*$ holds. The last lemma shows that if the expression d reduces to expression e in a step of $\rightarrow$ and e reduces to f with $\Rightarrow$, then d also reduces to f with $\Rightarrow$. This lemma would be a kind of transitivity property but mixing both semantics. For an informal presentation of this lemma, see the end of section 2.2.

$\text{cong}_{\text{app}\rightarrow*} \in (\rightarrow*(d,e))\rightarrow*(.(d,f),.(e,f))$
$\text{cong}_{\text{app}\rightarrow*}(\text{refl}_{\rightarrow*}) \equiv \text{refl}_{\rightarrow*}$
$\text{cong}_{\text{app}\rightarrow*}(\text{addstep}_{\rightarrow*}(h_1,h_2)) \equiv \text{addstep}_{\rightarrow*}(.\rightarrow(h_1),\text{cong}_{\text{app}\rightarrow*}(h_2))$
$\text{cong}_{\text{if}\rightarrow*} \in (\rightarrow*(d,d'))\rightarrow*(\text{if}(d,e,f),\text{if}(d',e,f))$
$\text{cong}_{\text{if}\rightarrow*}(\text{refl}_{\rightarrow*}) \equiv \text{refl}_{\rightarrow*}$
$\text{cong}_{\text{if}\rightarrow*}(\text{addstep}_{\rightarrow*}(h_1,h_2)) \equiv \text{addstep}_{\rightarrow*}(\text{if}\rightarrow(h_1),\text{cong}_{\text{if}\rightarrow*}(h_2))$
$\text{trans}_{\rightarrow*} \in (\rightarrow*(d,e);\rightarrow*(e,f))\rightarrow*(d,f)$
$\text{trans}_{\rightarrow*}(\text{refl}_{\rightarrow*},h_1) \equiv h_1$
$\text{trans}_{\rightarrow*}(\text{addstep}_{\rightarrow*}(h_2,h_3),h_1) \equiv \text{addstep}_{\rightarrow*}(h_2,\text{trans}_{\rightarrow*}(h_3,h_1))$
$\text{prop}_{\text{add}\rightarrow\Rightarrow} \in (\rightarrow(d,e);\Rightarrow(e,f))\Rightarrow(d,f)$
$\text{prop}_{\text{add}\rightarrow\Rightarrow}(.\rightarrow(h_2),.\Rightarrow(h,h_3)) \equiv .\Rightarrow(\text{prop}_{\text{add}\rightarrow\Rightarrow}(h_2,h),h_3)$
$\text{prop}_{\text{add}\rightarrow\Rightarrow}(.\lambda\rightarrow,h_1) \equiv .\Rightarrow(\lambda\Rightarrow,h_1)$
$\text{prop}_{\text{add}\rightarrow\Rightarrow}(\text{fix}\rightarrow,h_1) \equiv \text{fix}\Rightarrow(h_1)$
$\text{prop}_{\text{add}\rightarrow\Rightarrow}(\text{if}\rightarrow(h_2),\text{if}_{\text{true}}\Rightarrow(h,h_3)) \equiv \text{if}_{\text{true}}\Rightarrow(\text{prop}_{\text{add}\rightarrow\Rightarrow}(h_2,h),h_3)$
$\text{prop}_{\text{add}\rightarrow\Rightarrow}(\text{if}\rightarrow(h_2),\text{if}_{\text{false}}\Rightarrow(h,h_3)) \equiv \text{if}_{\text{false}}\Rightarrow(\text{prop}_{\text{add}\rightarrow\Rightarrow}(h_2,h),h_3)$
$\text{prop}_{\text{add}\rightarrow\Rightarrow}(\text{if}_{\text{true}}\rightarrow,h_1) \equiv \text{if}_{\text{true}}\Rightarrow(\text{true}\Rightarrow,h_1)$
$\text{prop}_{\text{add}\rightarrow\Rightarrow}(\text{if}_{\text{false}}\rightarrow,h_1) \equiv \text{if}_{\text{false}}\Rightarrow(\text{false}\Rightarrow,h_1)$

The equivalence of both semantics is given by two lemmas. In the first lemma we prove that for every expression, the result of $\Rightarrow$ is the same as the result of $\rightarrow^*$. To prove the other direction, as was already mentioned at the end of section 2.2, we need to introduce in the hypotheses of the theorem that the result of $\rightarrow^*$ is a canonical expression.

$\mathrm{eq}_{\Rightarrow\to^*} \in (\Rightarrow(d, e)) \to^*(d, e)$

$\mathrm{eq}_{\Rightarrow\to^*}(\lambda_\Rightarrow) \equiv \mathrm{refl}_{\to^*}$

$\mathrm{eq}_{\Rightarrow\to^*}(.\!\Rightarrow(h_1, h_2)) \equiv \mathrm{trans}_{\to^*}(\mathrm{cong}_{\mathrm{app}\to^*}(\mathrm{eq}_{\Rightarrow\to^*}(h_1)), \mathrm{addstep}_{\to^*}(.\!\lambda_\to, \mathrm{eq}_{\Rightarrow\to^*}(h_2)))$

$\mathrm{eq}_{\Rightarrow\to^*}(\mathrm{fix}_\Rightarrow(h_1)) \equiv \mathrm{addstep}_{\to^*}(\mathrm{fix}_\to, \mathrm{eq}_{\Rightarrow\to^*}(h_1))$

$\mathrm{eq}_{\Rightarrow\to^*}(\mathrm{true}_\Rightarrow) \equiv \mathrm{refl}_{\to^*}$

$\mathrm{eq}_{\Rightarrow\to^*}(\mathrm{false}_\Rightarrow) \equiv \mathrm{refl}_{\to^*}$

$\mathrm{eq}_{\Rightarrow\to^*}(\mathrm{if}_{\mathrm{true}\Rightarrow}(h_1, h_2)) \equiv \mathrm{trans}_{\to^*}(\mathrm{cong}_{\mathrm{if}\to^*}(\mathrm{eq}_{\Rightarrow\to^*}(h_1)), \mathrm{addstep}_{\to^*}(\mathrm{if}_{\mathrm{true}\to}, \mathrm{eq}_{\Rightarrow\to^*}(h_2)))$

$\mathrm{eq}_{\Rightarrow\to^*}(\mathrm{if}_{\mathrm{false}\Rightarrow}(h_1, h_2)) \equiv \mathrm{trans}_{\to^*}(\mathrm{cong}_{\mathrm{if}\to^*}(\mathrm{eq}_{\Rightarrow\to^*}(h_1)), \mathrm{addstep}_{\to^*}(\mathrm{if}_{\mathrm{false}\to}, \mathrm{eq}_{\Rightarrow\to^*}(h_2)))$

$\mathrm{eq}_{\to^*\Rightarrow} \in (\to^*(d, e);\ \mathrm{Can}(e)) \Rightarrow(d, e)$

$\mathrm{eq}_{\to^*\Rightarrow}(\mathrm{refl}_{\to^*}, \mathrm{can}_{\mathrm{abs}}) \equiv \lambda_\Rightarrow$

$\mathrm{eq}_{\to^*\Rightarrow}(\mathrm{refl}_{\to^*}, \mathrm{can}_{\mathrm{true}}) \equiv \mathrm{true}_\Rightarrow$

$\mathrm{eq}_{\to^*\Rightarrow}(\mathrm{refl}_{\to^*}, \mathrm{can}_{\mathrm{false}}) \equiv \mathrm{false}_\Rightarrow$

$\mathrm{eq}_{\to^*\Rightarrow}(\mathrm{addstep}_{\to^*}(h_2, h_3), h_1) \equiv \mathrm{prop}_{\mathrm{add}\to\Rightarrow}(h_2, \mathrm{eq}_{\to^*\Rightarrow}(h_3, h_1))$

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.

[CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language : Mini - ml. Technical Report 529, INRIA, France, May 1986.

[CNSvS94] T. Coquand, B. Nordström, J. M. Smith, and B. von Sydow. Type theory and programming. *EATCS*, 52, February 1994.

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

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th. ACM Symposium on Principles of Programming Languages*, Albuquerque NM, January 1982.

[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.

[Mag92] L. Magnusson. The new Implementation of ALF. In *The informal proceeding from the logical framework workshop at Båstad, June 1992*, 1992.

[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.

[NN92] H. Nielson and F. Nielson. *Semantics with Applications : A Formal Introduction.* John Wiley & Sons, Aarhus University, Denmark, 1992.

[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.