**Saarland University**

**Faculty of Natural Sciences and Technology I**

**Saarland University**

**Master thesis**

# Formalization of *Types and Programming Languages* [1] in Isabelle/HOL

submitted by

Michaël Noël DIVO

March 2017

| | |
|---|---|
| **Reviewers:** | Dr. Jasmin Christian Blanchette |
| | Prof. Dr. Christoph Weidenbach |

## Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

## Declaration

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

## Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

## Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, March 2017 Michaël Noël DIVO

## Abstract

Today, new languages provide a lot of complex structures. This new expressive power comes with a price, namely it becomes easier to build a compiler that allows bugs. Therefore, it is really important to be sure that the semantic of those programming languages is sound. Since this task is always inspired from previous work, I propose to formalize some chapters of a reference book [1] on type systems *Types and Programming Languages [1]* in Isabelle/HOL. Isabelle/HOL is a proof assistant, providing type checking and thus certifying the consistency of proofs. This environment will allow me to discover some inconsistency and imprecisions.

# Acknowledgment

I would like to thank my supervisor Jasmin Christian Blanchette for his support. He helped me understand the structure of Isabelle/HOL deeper and managed to explain me how to get out of some deadlock situations.

I am highly grateful to:

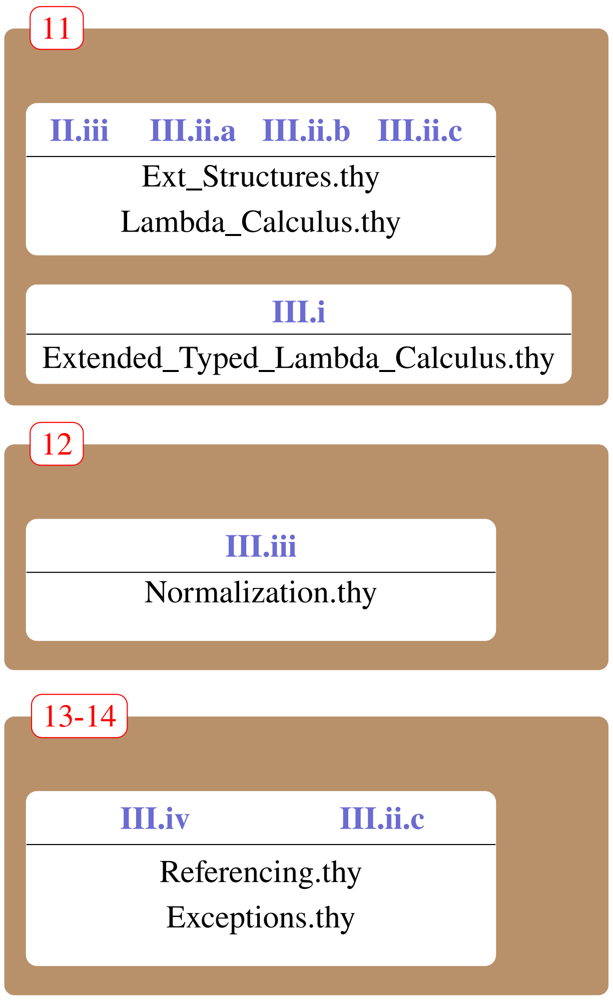
- Jamsin Christian Blanchette, Mathias Fleury and Daniel Wand (**Concrete Semantics with Isabelle/HOL**)
- Professor Gert Smolka and Steven Schäfer (**Semantics - Part 1**)
- Professor Derek Dreyer, Ralf Jung and Jan-Oliver Kaiser (**Semantics - Part 2**)

for their great lectures and tutorials on type systems and semantics.
The knowledges, they transmitted to me, were of great help all along this thesis.

Finally, I want to thank Martin Desharnais, who allowed me to use his repository and previous development.

# Contents

Please refer to the following figure to know the correspondance between files(.thy), chapters of this report and chapters of the book.



| Legend | |
|---|---|
| 11 | Chapter's number in the book |
| III.iv | Chapter's number in this report |
| Referencing.thy | Corresponding Isabelle theory file |

# Introduction

Proving correctness of a program is an hard task, even through a lot of tools already exists. Obviously, for people working in this field, having a lot of tools is generally not such a great thing. Indeed, it implies most of the time that the field contains several marginal cases or, in the worst case, that there exists no generalization. Furthermore, proving correctness of a program is pretty redundant, so it is more interesting to prove first soundness of our compiler. Then since the compiler is sound, considering only sound programs helps a lot. A compiler is based on a language, that can be arbitrarily complex and can then have a lot of marginal cases. But all languages can be simulated with lambda-calculi. Ironically, facts about lambda-calculi are studied and well-known since decades, but are not all formalized. Only intuition and paper works assert that they are correct. Having a formal definition for certain concepts becomes pretty hard and require a lot of work, that finally only look like playing hide-and-seek with supposed errors or imprecisions.

I propose to formalize some of the still unformalized chapters of a famous reference book: Types and Programming Languages [1] about type systems and lambda-calculi in order to enrich the background theory of Isabelle/HOL.
Isabelle/HOL is a proof assistant, that provides a formal environment for proving theories and a verification tool (type-checker) that certifies consistency of the inputed proofs. Martin Desharnais formalized the chapters 3, 5, 6, 8, 9 in his bachelor thesis [2] with Isabelle/HOL (other chapters are either introduction or implementations in ML, which didn't concern his work).
The main goal of this Master thesis is to investigate some remaining chapters of this book [1] in search of *imprecisions* and to develop *formal model for different languages' features* (patterns, referencing, ...).

All the proofs will be available as Isabelle/HOL 2016 files at Martin Desharnais' github repository [3].

My work consists in the formalization of proofs and exercises (when they are proofs) from chapter 11 to 14.

The first section of this thesis will try to explain non-exhaustively what is Isabelle/HOL and what are the features that played a big role in my work. First, I'll go over the notions of types, terms and tactics, which will lead us to the question of definition (how to define, what kind of definition should be favorised in which context, ...). Finally, when working with automation, it's nice to be able to help others by developping and sharing our own tools, which has become possible with the *tactics' language* Eisbach.

However, only knowing about Isabelle/HOL is not enough as background to understand the details of this thesis and in particular the difficulties behind such a work. That's why, the second section will talk about the informal proofs in the book [1] and the related notation. Then, in a second time, I will talk about my predecessor Martin Desharnais and his notations (in Isabelle) and concludes this second part with an introduction to my work: redundant rules and notations...

In the third section, I will present my work following the chapters of the book [1]. The only difference will be that I will fragment the chapter 11 since there is a lot of definitions and subtle works in it. Indeed, in chapter 11, we can found: easy structures (Tuple, Record, pair, ...), but also elaborated construction (pattern matching on disjoint sums, pattern matching on
variants, pattern matching, ...).

Chapters **Sequence and derived forms** (**III.i**) and **Normalization**(**III.iii**)
will present some corrections, since the book [1] is either wrong or imprecise.

Chapter **III.ii.a** will explain in details how to build a **disjoint sum feature** and the **patterns matching for this structure**, but also present a common **operator let**. Since **variants** also behave in a similar way, the particularities of this type, and especially about its pattern matching, will also be explained in this part.

Then, chapter **III.ii.b** will talk about how to **naively** build a **general pattern matching feature**, which is proposed as an exercise in the book [1]. This part will only give intuition about how to formalize this feature and the difficulties, I have encountered. It will be mainly inspired by a publication [4] of **Wolfram Kahl**.

Chapter **III.ii.c** is the last part about structures of the chapter 11. It will show what are the requirements to add a **native list type** to our language. The book [1] proposes the formalization of this part only as an exercise. It will introduce the way to formalize lists correctly, while chapter **III.iv** will present a correct language (type-safe) implementing lists in Professor Pierce's fashion.

Finally, the chapter **III.iv** will present type-safety of a Java-like
exceptions' feature (specifying throwable errors' types in function' s header), which was also given as an exercise in the book [1], and type-safety of a language
containing references(pointers).
This will conclude the third section of this thesis and the presentation of my work. Thus, I will conclude with a small section talking about related works and a brief summary of my work.

# I. Isabelle/HOL and computer-checked proofs

Isabelle/HOL is an application, commonly called proof assistant since it allows the user to generate mathematical proofs and check their correctness. It was originally developed at the University of Cambridge and Technische Universität München. Proof correctness can be checked thanks to the language for mathematical reasoning (proof are terms of this language) and the rules of correctness, that are *similar* to the rules of natural deduction, provided by Isabelle/HOL. With hand of this small consistent kernel, user will state his proof (assumption, fixing variable, ...) using a set of provided instructions (lemma, theorem, inductive, ...).
The set of rules for typing is defined in such a way that everything, that has been inputed (our proof), is well-typed if and only if our proof is consistent, w.r.t the rule of logic.

Furthermore, Isabelle/HOL decomposes logical reasoning in two levels of abstraction:

1. **generic logic**, based on a type bool and providing the common connectors (conjunction, universal quantification, ...)
2. **metalogic**, based on an intuitionistic fragment of Church's simple type theory and dealing with a type for predicates (prop)

The relation between the two levels is illustrated by the term Trueprop, that projects a term of type prop to the corresponding boolean(false, if inconsistent, and true otherwise).
Obviously, a lot of theories requires far more than only logical reasoning, namely a **background theory**. This one already exists in a lot of fields and is included as library with the application, but extra theories can be found on the *Archive of Formal Proofs*, that contains most of the theories proved so far. This thesis and Martin Desharnais' work are using some theory file from this archive (List_index.thy [5]).

Additionally, user is given tools to check if his statement (assumption, lemma) is correct. They are of **3 kinds**:

1. **counter-example** check commands(**quickcheck** and **nitpick**)

2. **tactics (auto, meson, metis, ...)**, they are algorithms that try to derive correctness, given the built-in logical rule and particular strategies

3. **inference tools (solve_direct, sledgehammer)**, that try to build a proof for the current statement given the all lemma and statement already proven. Indeed, sledgehammer will try to prove **the current statement (called Goal)** with the assumption given by the user and will suggest a tactic and the missing assumptions, that were required for this one to be successful.

Obviously, those tools are not all-mighty and all-knowing programms. Therefore, they only try to deduce inconsistency or proof scheme, when our goal belongs to a decidable fragment of logic and the number of assumptions is pretty small.

The previously mentioned tactics can also be written by users in ML or with the Eisbach language (see section I.iii) The next subsection will introduce terms and typing rules briefly and in see section I.ii, I'll give some explanations regarding automatically generated lemmas, function definition and termination proofs.

## I.i. Types, terms and tactics

First, I want to present the different types and related notations proper to Isabelle/HOL.

Types are either:

- **basic types** (nat, bool, prop, ...)
- **arbitrary types** represented by variables (preceded by a single quotation mark, 'a (replaces generic ML notation for polymorphic type $\alpha$)
- **instantiation of polymorphic types** (nat set, nat list, ...) and **function type** (nat⇒nat, ...)

  Notice that Isabelle/HOL doesn't implement dependent types.
- **recursive types**, generated by the user with the instruction **datatype**

  Example: **datatype** mType = **A** | $\mathbf{C}_1$ mType | $\mathbf{C}_2$ nat mType

  Each constructor(bold font) is separated by | and followed by the types of its arguments.

  Here, $\mathbf{C}_1$ and $\mathbf{C}_2$ are taking an element of mType (recursion) and a natural number for $\mathbf{C}_2$.

The type **bool** has the expected connectors: conjunction (∧), disjunction (∨), existential quantifier (∃), universal quantifier(∀).

**bool** is the type of built-in logical objects, so it's not the only way to write down a mathematical statement.

The **metalogic of Isabelle** will use different constructors and the type prop (instead of bool). Its constructors will be explained at the end of this section, when I present tactics.

The type system is important for checking validity of terms and proofs (expressions with prop or Bool type). But users actually deal more with terms than types. Terms in Isabelle HOL are divided into 4 kinds:

- **variables**, fixed or **schematic**.

  A schematic variable (begins by a question mark (?x)) can be instantiate with any term of the same type, while fixed variables are variables considered as instantiated with some value (namely, proving a property for all element of type A is the same as proving that the property is true for a given fixed variable of type A)

- **constant** (can be a function (sin, ...))

- **function application**, that are presented with parenthesis following the currying principle.

  Namely, every function symbol is considered as left-infix, so f x y is the same as ((f x) y) and a n-ary function can always be decomposed as an unary function returning a (n-1)-ary function.

  Isabelle also allows different types of notations (infix left, infix right, mixfix), such that the user can use common operation in the usual way (+, ...).

- **$\lambda$-abstraction**, that represents functions.
  For example, given f of type A⇒nat, we can build a function taking two arguments and returning the sum of their image by f: $\lambda$ x y. (f x) + (f y)

Finally, what interests us is how to prove some theory with Isabelle/HOL. So we need to understand how to build a goal and how to solve it.
**A statement** (goal, assumption, subgoal) can be represented as prop term or bool term. It's easier to see an example:

Mathematical statement: $\forall\, x.\ (\exists k.\ k \leq x) \longrightarrow x \geq 0$

bool term (built-in logic): $\forall\, x.\ (\exists k.\ k \leq x) \longrightarrow x \geq 0$

prop term (metalogic): $\bigwedge x.\ (\exists k. k \leq x) \Longrightarrow (x \geq 0)$

Notice that the bool term is very natural, since it's the formulation, we are used to. The metalogic formulation is convenient in the sense that some automatic checks are quicker with it.

The natural question is why are two levels of abstraction convenient.
The answer is not quiet easy, but intuitively, the metalogic is there to simplifiy automation. Namely, some tools (tatics) are designed to quickly derive inconsistency or correctness of a statement in the metalogic, while they are slow or can't be applied on the built-in version(derivation rules are different).
Tactics are nothing more that derivations abiding the *law of natural deduction* (even through some rules [6] are a bit different). Those derivations are made using an algorithms implemented in ML. For example, we have the tactic simp, that will try to simplify the goal with the assumptions(equalities, ...) and the registered simplification rules. We have then more elaborated tactics like auto(heuristic), metis(more at ease with rule application(symmetry, associativity, ...) than auto, but doesn't automatically simplify), ...

We also need to be able to implement our own constant terms. Indeed, we may need specific functions (example: substitution, ackermann, ...) and inductive definitions. Next section will cover this part of Isabelle/HOL and present a small proof as an example.

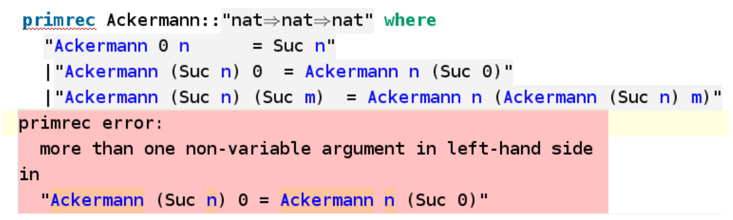
## I.ii. Definitions and termination

In the previous section, I mentioned the possibility of defining any arbitrary recursive type with the instruction **datatype**. Following the same principle, there are 3 common ways to define a function:

- **primrec**, defines a primarly recursive function for which we need to state every computation rules. Since it is pretty strict, we will most of the time get an error message like in the example below (only recursion on the first argument is allowed)
- **fun**, defines a function in a more global sense (allowing to regroup cases, add default behavior, ...) and checks for termination (search of a termination order)
- **function**, defines a function for which we can add any computation rule independently from the termination of the function and its completeness. This allows in particular to define nested-recursive functions or mutually recursive function (when specified with extra instructions). But, we are compelled to prove completeness(every recurive pattern corresponds to a computation rule) and termination(there is a termination order on the function)

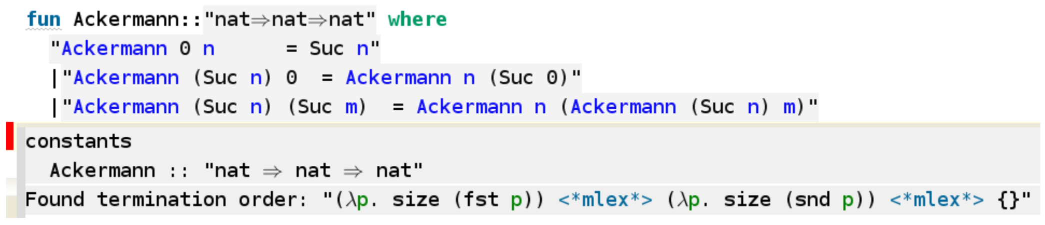
For each case, it's easier to see an example:

**primrec Ackermann**:



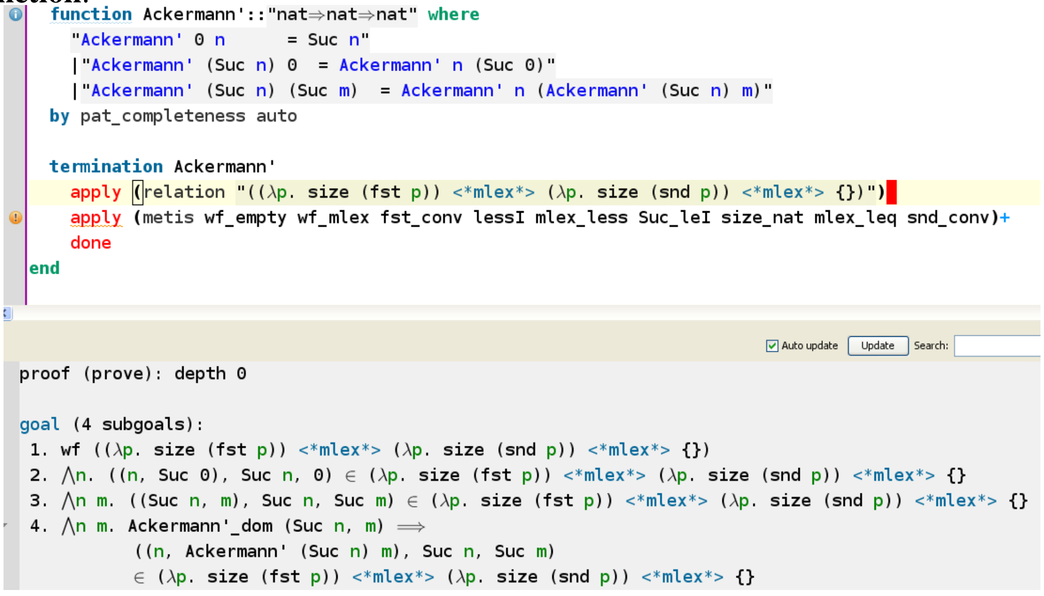
```
primrec Ackermann::"nat⇒nat⇒nat" where
  "Ackermann 0 n      = Suc n"
  |"Ackermann (Suc n) 0  = Ackermann n (Suc 0)"
  |"Ackermann (Suc n) (Suc m)  = Ackermann n (Ackermann (Suc n) m)"
primrec error:
  more than one non-variable argument in left-hand side
in
  "Ackermann (Suc n) 0 = Ackermann n (Suc 0)"
```

**fun**:



```
fun Ackermann::"nat⇒nat⇒nat" where
  "Ackermann 0 n      = Suc n"
  |"Ackermann (Suc n) 0  = Ackermann n (Suc 0)"
  |"Ackermann (Suc n) (Suc m)  = Ackermann n (Ackermann (Suc n) m)"
constants
  Ackermann :: "nat ⇒ nat ⇒ nat"
Found termination order: "(λp. size (fst p)) <*mlex*> (λp. size (snd p)) <*mlex*> {}"
```

**function**:



```
function Ackermann'::"nat⇒nat⇒nat" where
  "Ackermann' 0 n       = Suc n"
 |"Ackermann' (Suc n) 0  = Ackermann' n (Suc 0)"
 |"Ackermann' (Suc n) (Suc m)  = Ackermann' n (Ackermann' (Suc n) m)"
by pat_completeness auto

termination Ackermann'
  apply (relation "((λp. size (fst p)) <*mlex*> (λp. size (snd p)) <*mlex*> {})")
  apply (metis wf_empty wf_mlex fst_conv lessI mlex_less Suc_leI size_nat mlex_leq snd_conv)+
  done
end
```

```
proof (prove): depth 0

goal (4 subgoals):
 1. wf ((λp. size (fst p)) <*mlex*> (λp. size (snd p)) <*mlex*> {})
 2. ⋀n. ((n, Suc 0), Suc n, 0) ∈ (λp. size (fst p)) <*mlex*> (λp. size (snd p)) <*mlex*> {}
 3. ⋀n m. ((Suc n, m), Suc n, Suc m) ∈ (λp. size (fst p)) <*mlex*> (λp. size (snd p)) <*mlex*> {}
 4. ⋀n m. Ackermann'_dom (Suc n, m) ⟹
          ((n, Ackermann' (Suc n) m), Suc n, Suc m)
          ∈ (λp. size (fst p)) <*mlex*> (λp. size (snd p)) <*mlex*> {}
```

The termination order has to be specified by the user if the function is defined with the instruction *function* and has to be well-founded. User can define multiple orders, if required. In the case of the *fun* and *primrec* definition, this order is automatically derived by Isabelle/HOL (multiset lexicographic order(<*mlex*>) based on size of recursive arguments, most of the time).

But that's not the only lemmas that Isabelle generates automatically. Indeed, when a term or a function is defined, the following lemmas are automatically derived and proved:

- **inject**, injectivity of constructors of recursive type
- **distinct**, implied by injectivity of constructors of recursive type (which term is different from wich other)
- **elim**, case analysis lemma on each rule of computation (or constructor in the case of term's definition) given in premise an equality between an application of the function (or a term) and some other expression of the same type
- **simps**, simplification rules (computation rules of function oriented from left to right) for the function automatically registered for tactics
- **induct and case**, respectively, induction principle and case analysis corresponding to the recursive structure
- and a lot of other lemmas : discrimation, nchotomy, ...
  (see Isabelle documentation [7])

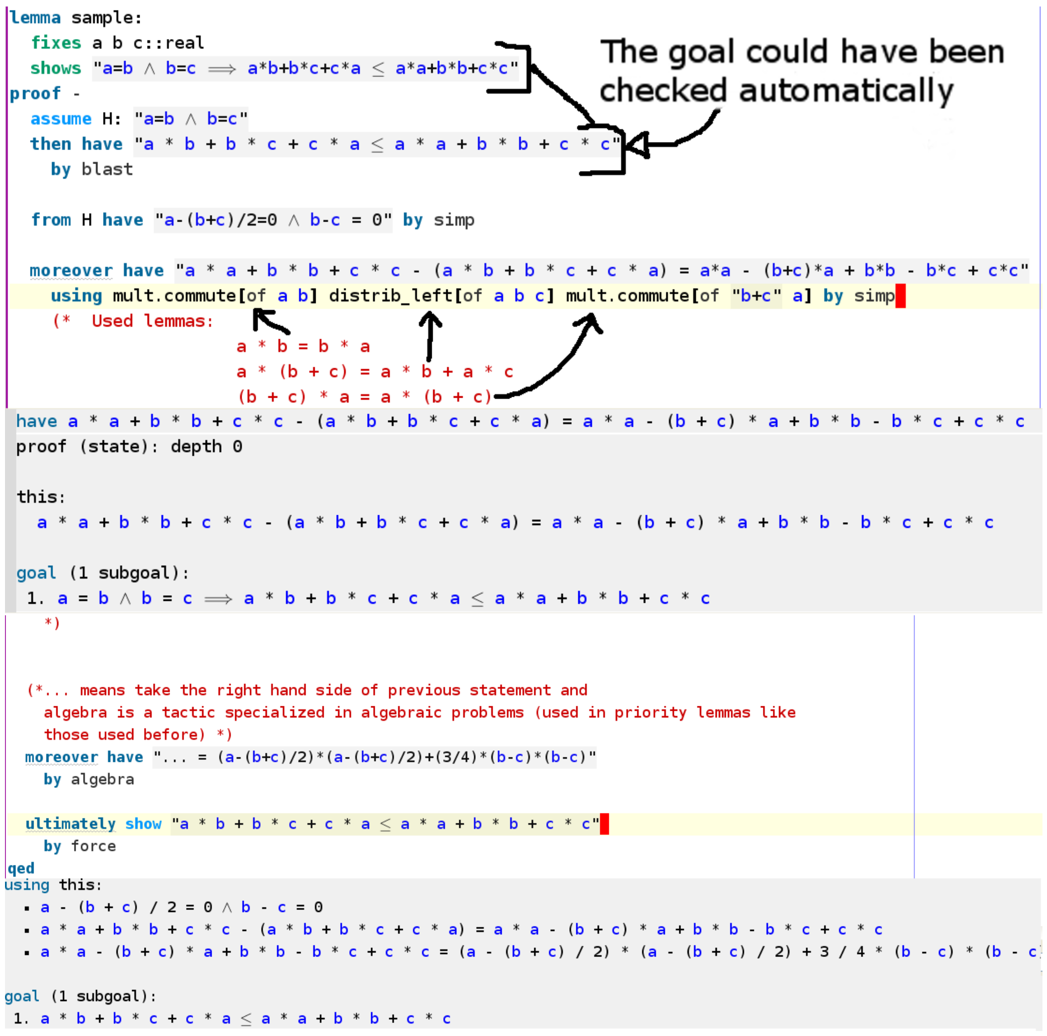
Finally, user wants to be able to define his own recursive predicates. That's possible with the instruction inductive. In the next example, I use the notation #, which stands for the constructor cons of lists.

Example:

```
Inductive OddSubset::"nat list ⇒ nat set ⇒ bool"
  Even_case:  "x mod 2 = 0 ⟹ OddSubset (x#L) O ⟹
              OddSubset L O"
  |Odd_case:  "x mod 2 = 1 ⟹ OddSubset (x#L) O ⟹
              OddSubset L ({x} ∪ O)"
```

For inductive predicates, the generated lemmas are the expected one, namely induction principle, monotonicity, introduction and eliminates rules, simplification rules (*similar* to inversion).

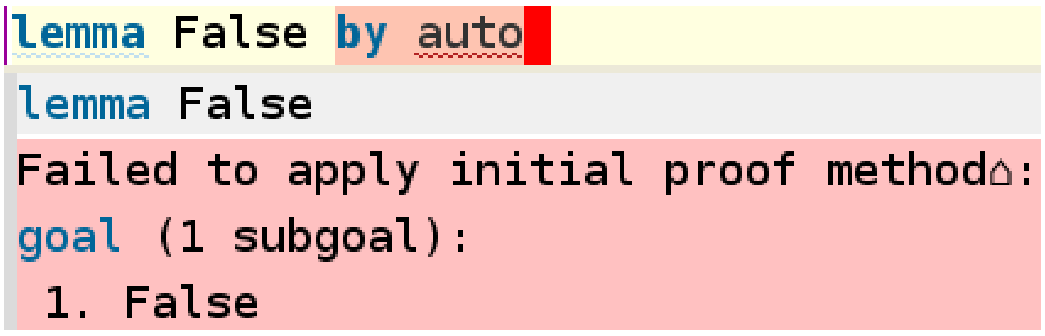
Now, with all those nice features, we can do for example the proof on next page(original proof [8] proposed by **Kunihiko Chikaya**).



```
lemma sample:
  fixes a b c::real
  shows "a=b ∧ b=c ⟹ a*b+b*c+c*a ≤ a*a+b*b+c*c"
proof -
  assume H: "a=b ∧ b=c"
  then have "a * b + b * c + c * a ≤ a * a + b * b + c * c"
    by blast

  from H have "a-(b+c)/2=0 ∧ b-c = 0" by simp

  moreover have "a * a + b * b + c * c - (a * b + b * c + c * a) = a*a - (b+c)*a + b*b - b*c + c*c"
    using mult.commute[of a b] distrib_left[of a b c] mult.commute[of "b+c" a] by simp
    (*  Used lemmas:
                     a * b = b * a
                     a * (b + c) = a * b + a * c
                     (b + c) * a = a * (b + c)
have a * a + b * b + c * c - (a * b + b * c + c * a) = a * a - (b + c) * a + b * b - b * c + c * c
proof (state): depth 0

this:
  a * a + b * b + c * c - (a * b + b * c + c * a) = a * a - (b + c) * a + b * b - b * c + c * c

goal (1 subgoal):
 1. a = b ∧ b = c ⟹ a * b + b * c + c * a ≤ a * a + b * b + c * c
    *)


  (*... means take the right hand side of previous statement and
    algebra is a tactic specialized in algebraic problems (used in priority lemmas like
    those used before) *)
  moreover have "... = (a-(b+c)/2)*(a-(b+c)/2)+(3/4)*(b-c)*(b-c)"
    by algebra

  ultimately show "a * b + b * c + c * a ≤ a * a + b * b + c * c"
    by force
qed
using this:
  ▪ a - (b + c) / 2 = 0 ∧ b - c = 0
  ▪ a * a + b * b + c * c - (a * b + b * c + c * a) = a * a - (b + c) * a + b * b - b * c + c * c
  ▪ a * a - (b + c) * a + b * b - b * c + c * c = (a - (b + c) / 2) * (a - (b + c) / 2) + 3 / 4 * (b - c) * (b - c

goal (1 subgoal):
 1. a * b + b * c + c * a ≤ a * a + b * b + c * c
```

Figure 1: Isar Proof samples in Isabelle/HOL

See also below how Isabelle notifies the user, that it can't apply a tactic (namely here, because the goal is inconsistent).



```
lemma False by auto
lemma False
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. False
```

As remark, I should mention that two styles are allowed, the first one, called *apply style*, is a backward reasoning, going from goal to premises, while the second one, *Isar style*, looks more like the mathematical reasoning, we are used to, with similar notations (then, thus, assume, have, hence ...). My development will be entirely done in Isar style. Since Isabelle provides so much automation, we also would like to do as less copy-paste as possible. Therefore, when we always use the same reasoning (contraposition, contradiction or something more specific) we would like to be able to give this reasoning a name and call it (like a tactic) instead of copy pasting the whole proof.
This can be done and will be describe in the next section.

## I.iii. Eisbach, tactics' language

Given an abitrary statement, in a lot of case, one tactic will not be sufficient to solve the goal. For example, when we have a inductive predicate as premises, we cannot solve the goal directly, but maybe it goes through by using induction and then by a tactic like auto.
In same fashion, we might have 4 subgoals and we need 3 different tactics to solve them, but if a tactic fails to solve the goal, we need to tell Isabelle that it has to try the others that we specified.

Therefore, Isabelle provided tactics combinators for abitrary tactics t1 and t2: "t1 ,t2" which is for sequential application(example: induction, auto in previous case), "t1;t2", which is for sequential application of t2 on each subgoals generated by t1, "t1[n]", which apply t1 to the n first actual subgoals, "t1?", that try to apply t1 and finally "t1|t2", which apply t1 or t2, if t1 fails.

As we can imagine, a very long string composed of tactics and combinators can appear often and for similar reasoning. Thus, Isabelle/HOL proposes *a language of tactics*. Indeed, with the instruction **method**, we can define any combination of basic tactics and give it a name like in the example below.

Example: method test = (simp, meson;((metis?)|force))

We can then call test which will exactly apply the combined tactics specified. Furthermore, tactics like meson or metis, but also auto, can take arguments (lemmas or assumptions) which leads to more complicated tatics. This kind of behavior can be emulated for our own method as well.

Example: method test uses myintros myelims = (auto intro:myintros elim:myelims)

We can then call test like we would have done it for auto (test myintros: (put there our introduction) lemmas myelims: (put there the elimination lemmas)). Finally, we could easily understand that redundancy in reasoning comes from similitude in proof context form.
By context form is meant premises (assumptions) and goal(s). That's why we would like to be able to match on those structures and Eisbach allows we to.
Indeed the second really nice feature of Eisbach is that we can use the instruction **match** on goal and premises to extract elements.

Namely, we get:

**match premises in** (some name): "(some statement with variables like A and B)" **for** A::"(put here type of A)" and B::"(put here the type of B)" ⇒ <(some tactics using lemmas, that can be instantiate with A and B)>

**match conclusion in** (some name): "(some statement with variables like A and B)" **for** A::"(put here type of A)" and B::"(put here the type of B)" ⇒ <(some tactics using lemmas, that can be instantiate with A and B)>

In this example, I mention *instantiation of lemmas*. In fact, Eisbach proposes attributes that allows instantiation and formatting of lemmas and assumptions. Here are some short examples:

- We have *le_Suc* : ?x ≤ Suc ?x, which says that any arbitrary natural number x is less than its successor.
  So we can instantiate it with a fixed variable N, taken from our context, which is le_Suc[of N].
  The result is a lemma stating: N ≤ Suc N.

- We have some lemma *le_imp_leq*: ?x ≤ ?y ⟹ ?x ≤ ?y.
  If we know as assumption or by a lemma H, that A≤B, we can instantiate le_imp_leq: le_imp_leq[OF H].
  The result is a lemma stating: A ≤ Suc B. If some schematic variables not mentioned in the instantiated premise(s) exist, they can because instantiated le_imp_leq[OF H, of (my variables separated by a space)]

- unfolding and simplifying, any assumption of lemma can be followed by the attributes unfolded or/and simplified.
  unfolded will simplify using the lemma(s)/assumption(s) put behind it, while simplified will try to use all registered simplification rules

There are a lot of other attributes, namely for theorems' registering (auto, simp, sym, elim, ...). Example: lemma f_map[simp[, for simplification.

This concludes the introduction to Isabelle/HOL and its feature, every information from this section and further explanations can be found in Isabelle documentation [7] [6] [9]. This report will contain some notations of the Isabelle development. Next part will introduce the background of this thesis (reference book [1], Martin Desharnais' bachelor thesis) and the actual common parts and notations, that will be required for the understanding.

# II. $\lambda$-calculus, background and common notations

## II.i. Pierce's rules and notations

Benjamin C. Pierce describes, in his book [1], paper proofs ruled by common mathematics as we know them. That's why, we need to understand the particular mathematical object(s) we want to study and the axioms, that will be used.

First of all, a calculus is defined by four important parts:

1. **a type for terms**, that are either basic elements (variables, unit, true, false, ...) or higher structures recursive on term (application, abstraction, record, ...).
   For example, a record is built with a list of string (field's label) and a list of terms (field's content): (Record $[x, y]$ [0, 1]), (Record [] [0, 1]), ...

2. **a type for types of terms**, that are arbitrary types (A, B,...) or values' types (Unit, Bool, Nat, ...) or higher types (function type, disjoint sum, pair, record type, ...)
   For example, a function type form abritrary type A (T 1) to arbitrary type B (T 2): (T 1) → (T 2)

3. **evaluation rules**, that state explicitely how the calculus must behave.
   For example, applying a function to a value replace the first argument in the body of the function by the given value: ($\lambda$ a. a+b) 5 evaluates to 5+b

4. **typing rules**, that restrict the considered set of terms to only **well-behaved terms**.
   **Well-behaviour** is some abstract concept, that is commonly used but never named.
   For example, each field of a record should have a label:
   (Record $[x, y]$ [0, 1]) is correct, while (Record [] [0, 1]) is clearly wrong.

This structure needs to be implement for any language and is combined with the following axiom:

**$\alpha$-renaming**: Equality of terms is equivalent to equality up to variable renaming.

Example: $\lambda$ a. a = $\lambda$ b. b, since it's equal up to the renaming that replace b by a or the other way around.
Same goes for ($\lambda$ a. a + b) c and ($\lambda$ y. y + d) f, with renaming that associates a with y, b with d and c with f

Evaluation rules defines a transition system, that simulates the behaviour of a program ($\lambda$-term) during the execution.
Example: If True then t1 else t2 reduces to t1, for t1 and t2, two arbitrary $\lambda$-terms. Pierce use the notation "t1 $\rightarrow$ t2" to express the fact that a term t1 reduces to a term t2.

Now, we know that our programs need to terminate at some point, so $\lambda$-terms should not have an infinite reduction path. Therefore, our calculus need an endpoint for evaluation, which is commonly called a **value** in programming.
Professor Pierce defines the new values inductively, each time he adds a new construction to the set of terms (add a new instruction to the programming language). For example, any function is a value, since its behavior will depend on the arguments parsed to it and we want to be able to call a function that takes a function as argument.

Those values are also **canonical forms** of their type: "*If we have a value and its type, then we know which constructor is used as top of the term, if we consider a term as a syntactic tree*".

Formally, we get lemmas of the form: is_value v $\Rightarrow \Gamma \vdash v : Bool \Rightarrow$ v= True $\vee$ v=False, where is_value is a predicate only true if v is a value. The type Bool is a typical example, but that's not the only one.
Finally, for the typing rules, we need to keep track of the type of each variables. Therefore, we use a **set**, called **context**, containing pairs of variable's name and type.

Example: $\{(x, A)\} \vdash x : A$, means that variable x has type A, knowing that x has type A.

$\vdash \lambda\,(x : A).\,x : A \rightarrow A$, means that a function, taking an argument x of type A and retruning x, has functional type A into A, **without any previous knowledge, empty set will always be omitted**.

After defining all those structures, Pierce starts his real study of the programming language. In fact, all that has been done until now is just a premise to the real work, proving ***Type Safety*** of our language.
If this theorem is correct, then it means informally that:
"*Our language is convergent and a correct program never becomes corrupted during execution (no bugs)*". Then, we can guess that since we speak about two different concepts: convergence and bug-freeness, we will prove this theorem in two parts: **Progress and Preservation**.

The first part, **Progress**, is a lemma, that states informally: "*If a program is correct without prior knowledges about it, then either it has a next execution step or it reached a value and terminates*".

The formal version is the following:

$\forall$ t A. $\vdash\ t\ :\ A \Rightarrow$ is_value t $\vee$ ($\exists$ t1. t $\rightarrow$ t1).

The second part, **Preservation**, states informally that: "*If a program is correct with arbitrary prior knowledges about it and it evaluates in one step to another program, then this new program is correct given the same prior knowledges*".

The formal version is the following:

$\forall\, \Gamma$ A t t1. $\Gamma\ \vdash\ t\ :\ A \Rightarrow$ t $\rightarrow$ t1 $\Rightarrow \Gamma\ \vdash\ t1\ :\ A$.

Furthermore, in order to prove **Preservation**, we will three important facts. First, it's obvious that given a correct program in a context $\Gamma$, we can deduce the following two facts:

- **Weakening**: We can add to the context any new consistent knowledge, i. e. any pair of variable name and type, if there is not already another pair with the same variable name in the context.
- We don't need to care about the order of the context (*set*). Thus, the judgment $\{(x, A), (y, B)\}\ \vdash\ y\ :\ B$ is correct. This reasoning only applies to for the set representation and has to be translated to a different structure, when done on a computer (see next part)

Finally, Professor Pierce describe the last important lemma required for proving **Preservation**. Namely, the question is what happens when we do susbtitution (function application case).
The lemma **Substitution** states that: "*Given a context, any variable n and two terms s and t, such that n and s have the same type in the context and t is well-typed in the same context, substituting the variable n by s in the term t doesn't modify the type*".

Professor Pierce uses the notation $(t)_s^n$ to represent the term t,in which the variable n is substituted by s. The substitution operator abides the non-exhaustive rules' set below (given the fact that x, y and n are always variables and s, t, u are always terms):

$(x)^n_s$ = x if x≠n $(n)^n_s$ =s

$(s\ t)^n_u$ = $(((s)^n_u)\ ((t)^n_u))$

$(\lambda\ x.\ t)^n_s$ = $(\lambda\ y.\ ((t)^x_y)^n_s)$ given y a *fresh* variable (not contained in t) and y≠n

The formal version of the substitution lemma states:

$$\forall \Gamma\ \mathrm{A\ B\ n\ s\ t.}\ \Gamma \vdash t : A \Rightarrow \Gamma \vdash s : B \Rightarrow \Gamma \vdash n : B \Rightarrow \Gamma \vdash (t)^n_s : A.$$

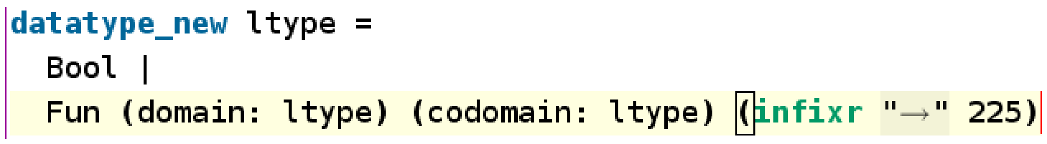
All those concepts(see book [1] for more precision) have to be reformulated, if we want to check the proof correctness with a computer, for the reasons that will be given in next section. I'll also present briefly the work of Martin Desharnais, who started the formalization for his bachelor thesis [2].

## II.ii. Martin's work and notations

In this part, I will present the important results, that Martin formalized. All pictures in this part are screenshots of the original Isabelle files, written by Martin Desharnais and that I updated (copy-paste, get rid of some apply code, replace some tactics by new ones). These files can be found on the repository [3], that belongs to Martin Desharnais.

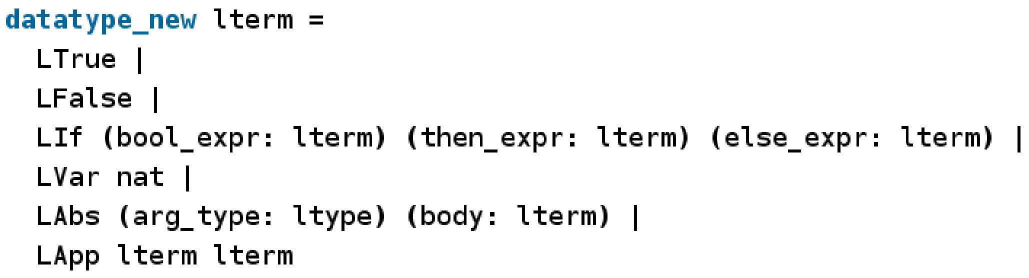
From previous section, we know how to formalize a calculus. Indeed, we first need to define the types of our language. Martin formalized a language, called ***S**implify **T**yped **L**ambda **C**alculus* or *STLC*.

*STLC* is based on a really easy language with a *base* type and a *recursive type constructor*. In the book [1], Professor Pierce chooses to use the type Bool for the base type. The recursive constructor helps to build function types.
Therefore, Martin defined in Isabelle the type as an inductive structure:



```
datatype_new ltype =
  Bool |
  Fun (domain: ltype) (codomain: ltype) (infixr "→" 225)
```

The terms are defined as follows:



```
datatype_new lterm =
  LTrue |
  LFalse |
  LIf (bool_expr: lterm) (then_expr: lterm) (else_expr: lterm) |
  LVar nat |
  LAbs (arg_type: ltype) (body: lterm) |
  LApp lterm lterm
```

Notice that functions are represented in a different way compared to Pierce's definition. The main reason resides in the $\alpha$-renaming axiom.

Indeed, if we give to a computer 2 terms with different *binded variables* and *same* content up to $\alpha$-renaming: $\lambda$ x. x and $\lambda$ y. y, the computer will consider 2 cases:
x=y, then both terms are equal and x≠y then both are different.
If Martin would have assumed $\alpha$-renaming as an axiom, then we could prove 1=0 from $\lambda$ 0. LVar 0 = $\lambda$ 1. LVar 1, which is **inconsistent**.
Therefore, we need some ingenious term representation, that implies $\alpha$-renaming.

This technic is called De Bruijn representation and was proposed by Nicolaas Govert de Bruijn in 1972 [10]. Since we need a infinite type for the naming of variables, choosing natural number is obvious and common. But then, the problem of $\alpha$-renaming arises. De Bruijn proposed to index the $\lambda$ binder in a term. Namely, for an expression like $\lambda$ x. x+1, there is one lambda, so it has index 0 and each occurence of the *binded argument* (x) should be replaced in *the binded term* (x+1). So $\lambda$ x. x+1 becomes $\lambda$. (LVar 0) + 1.

But other problems occur:

- **Indexing order**: $\lambda$ x. $\lambda$ y. x+y becomes $\lambda$. $\lambda$. (LVar 0) + (LVar 1), which lambda bindes the variable 0?
  For convenience, when we need to give a type to such an expression, it's easier to index the lambdas beginning from the deeper one.
  So, in the previous example, $\lambda$.(has index 1) $\lambda$.(has index 0) (LVar 0) + (LVar 1).

- **Substitution problem**: ($\lambda$. (LVar 0) + (LVar 1)) (LVar 1) represents ($\lambda$ x. x + y) z. But if we apply naively the susbtitution, we get that (LVar 1) + (LVar 1) represents z + y, which is wrong generally since z could be different from y.
  In order to get rid of these problems, De Bruijn proposed to modify the substitution operation and to add a new operator, called **shifting**.

Before going more into details with **shifting and substitution** (functions shift-L and subst-L), we need now to think about evaluation rules and typing rules. At this point, once more, problems arise. We could use *set-like structures* to represent the context like Professor Pierce did, but then formally speaking, we will be using something that is like a **dictionnary** (since we have to impose unique key (variable index, here)). This could become really painful and annoying, and that's why Martin implemented the common version of typing with **lists**.

This is convenient since the indexation allows know to easily check the type of each variable (type at the index position in the context). The context will behave like a stack. That's the reason for indexing from the deeper lambda to the shallower one.

Example: from $\Gamma \vdash LAbs\ (B \rightarrow A)(LAbs\ B\ ((LVar1)(LVar0)))\ \ |{:}|\ \ A$,
we can **retrieve the type of the second argument** and push on the stack

$(B \rightarrow A)\#\Gamma \vdash (LAbs\ B\ ((LVar1)(LVar0)))\ \ |{:}|\ \ A$

$B\#((B \rightarrow A)\#\Gamma) \vdash (LVar1)(LVar0)\ \ |{:}|\ \ A$,
now it is **easy to determine** that LVar 0 has type B and
LVar 1 has type $B \rightarrow A$

Martin has defined an inductive judgment for typing, annoted like in the previous example. Martin use a special notation |∈|, which means for (x,A)|∈| Γ that the type A is at the index x in the list Γ. The actual predicate is illustrated in the figure below.

$$\text{LIf\_True}\ \frac{}{\Gamma \vdash LTrue\ \ |{:}|\ \ Bool} \qquad \text{LIf\_False}\ \frac{}{\Gamma \vdash LFalse\ \ |{:}|\ \ Bool} \qquad \text{LVar}\ \frac{\text{(x,A)}|{\in}|\ \Gamma}{\Gamma \vdash LVar\ x\ \ |{:}|\ \ A}$$

$$\text{LAbs}\ \frac{A\#\Gamma \vdash t\ \ |{:}|\ \ B}{\Gamma \vdash LAbs\ A\ t\ \ |{:}|\ \ A \rightarrow B} \qquad \text{LApp}\ \frac{\Gamma \vdash t1\ \ |{:}|\ \ A \rightarrow B \qquad \Gamma \vdash t2\ \ |{:}|\ \ A}{\Gamma \vdash LApp\ t1\ t2\ \ |{:}|\ \ B}$$

$$\text{LIf}\ \frac{\Gamma \vdash c\ \ |{:}|\ \ Bool \qquad \Gamma \vdash t1\ \ |{:}|\ \ A \qquad \Gamma \vdash t2\ \ |{:}|\ \ A}{\Gamma \vdash LIf\ c\ t1\ t2\ \ |{:}|\ \ A}$$

Figure 2: Typing rules for STLC

The operator |∈| check wether the first projection of a pair(left-hand side) is inferior to the length of the context (right-hand side) and if the element at this position in the context is the second projection.
Martin also defined a predicate for evaluation (predicate is named **eval1_L**) and declared a predicate **is_value_L**, that is true for values only, see below.

$$\text{value\_True}\ \frac{}{\text{is\_value\_L LTrue}} \qquad \text{value\_False}\ \frac{}{\text{is\_value\_L False}} \qquad \text{value\_Abs}\ \frac{}{\text{is\_value\_L (LAbs A t)}}$$

$$\text{LIf\_True}\ \frac{}{\text{eval1\_L (LIf LTrue t1 t2) t1}} \qquad \text{LIf\_False}\ \frac{}{\text{eval1\_L (LIf LFalse t1 t2) t1}} \qquad \text{LIf}\ \frac{\text{eval1\_L c c'}}{\text{eval1\_L (LIf c t1 t2) (LIf c' t1 t2)}}$$

$$\text{LApp1}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (LApp t1 t2) (LApp t1' t2)}} \qquad \text{LApp2}\ \frac{\text{is\_value\_L v} \qquad \text{eval1\_L t2 t2'}}{\text{eval1\_L (LApp v t2) (LApp v t2')}}$$

$$\text{LApp\_LAbs}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (LApp (LAbs A t1) v) (shift-L } (-1) \text{ 0 (subst-L 0 (shift-L 1 0 v) t1))}}$$

Figure 3: Evaluation rules and values for STLC

At this point, substitution was a real problem, since we need to shift variables of the term, that we want to substitute, and the variables in the substituted term. Furthermore, we want to shift variables depending of their index. That's why shift-L takes as first argument the span of the shifting and as second argument a number, which indicate the least index that has to be shifted, any variable with an index below it will not be shifted.
**Why?** Answer is in this example:

| | |
|---|---|
| Pierce's notation: | $\{(\mathrm{f}, \mathrm{Nat}\rightarrow A),\ (\mathrm{x}, \mathrm{Nat})\} \vdash (\lambda \mathbf{g}.\ \mathbf{g}\ \mathbf{x})\ f\ :\ A$ |
| After one step of execution: | $\{(\mathrm{f}, \mathrm{Nat}\rightarrow A),\ (x, Nat)\} \vdash \mathbf{f}\ \mathbf{x}\ :\ A$ |
| With De Bruijn representation: | $[\mathrm{Nat}, \mathrm{Nat}\rightarrow A] \vdash (\lambda.\ \textbf{(LVar 0) (LVar 1)})\ \mathbf{1}\ \vert{:}\vert\ A$ |
| Naive substitution: | $[\mathrm{Nat}, \mathrm{Nat}\rightarrow A] \vdash \textbf{(LVar 1) (LVar 1)}\ \vert{:}\vert\ A$ semantically, x x or f f, which is wrong |
| With shifting: | $[\mathrm{Nat}, \mathrm{Nat}\rightarrow A] \vdash \textbf{(LVar 1) (LVar 0)}\ \vert{:}\vert\ A$ |

$(\mathrm{g\ x})^{g}_{f} \triangleq$ shift-L (-1) 0 (subst-L 0 (shift-L 1 0 (LVar 1)) ((LVar 0) (LVar 1)))

In the same fashion, the naive substitution of Professor Pierce had a rule for function, that avoided the binded variable to be *captured*, i. e. to be replaced. The function subst-L has an easy rule for this case:

subst-L j s (LAbs A t) = LAbs A (subst-L (Suc j) (shift-L 1 0 s) t).

Since application of such a term to another yields shift-L (-1) 0, we need to increment the variables in s, so that the result ends up to be the expected one.

The design of such rules for shifting and substitution is really important for the reasoning, since the free (not binded) variables of a term are relevant in a lot of case. The set of such variable is also implement as a function **FV** returning a set of natural number (free variables index). For this function FV, it matters to understand that, for example, **FV (LAbs A t)** is not FV t but the image of FV t without 0 (binded variable) by the function ($\lambda$ x. x-1). This change comes from the fact that we have to deal with shifting and substitution.
Indeed, when we shift a variable, we will apply a function like
($\lambda$x. if x≥x then x+d else x). In order to generalize this notation, namely to show that **FV(shift-L d c t)** is the **image of FV t** by **($\lambda$x. if x≥c then x+d else x)**, we need to do something for **the case LAbs**, since the shifting modify the condition c. That's why FV(LAbs A t) is an image and not FV t without 0 only. This fact is critical, when we want to prove Preservation (evaluation of application). Furthermore, we end up with a lot of cases with beta reductions(rules leading to substitutions), which obviously requires knowledges on free variables. Since we want to type some term of the kind: "shift-L (-1) c (subst-L j ...)", we have no choice but to apply **Shift_down**, which has a precondition on free variables of "subst-L j ..." in this case. Martin created two lemmas that will be useful in those cases:

- FV_shift: ∀ t d c. FV(shift-L d c t) = (if x≥c then x + d else x)@FV t, @ means taking the image of the right-hand side by the left-hand side (see next part).
- FV_subst: ∀ n t u. FV(subst-L n t u) = (if n∈ FV u then FV u -$\{n\}$ ∪ FV t else FV u)

The definitions of subst-L and shift-L will not be described in details for STLC. Please check Appendix B for the formal definition (LVar, LTrue, LFalse, LApp, LAbs and LIf definitions are due to Martin Desharnais and were copy-pasted). The evaluation and typing rules for terms of STLC (variable, lambda, application, boolean and if term) can be found in Martin's file
(Typed_Lambda_Calculus.thy).

Martin proved then **Progress** and **Preservation of STLC**. In order to archieve this, he had to reformulate **Weakening and Substitution** and to create some formalism to deal with context order(the function insert-nth k s L, inserts s at the kth position in L (the head of the list has position 0)):

- **Weakening**:

$$\forall \Gamma\ \text{n S t A}.\ \Gamma \vdash t\ |{:}|\ A \Rightarrow \text{n} \leq \text{length}\ \Gamma$$
$$\Rightarrow \text{insert-nth}\ n\ S\ \Gamma \vdash \text{shift-L 1 n t}\ |{:}|\ A$$

- **shift-down** is a lemma formulated by Martin, that will allows to restructure our context. Indeed, Professor Pierce stated that order is not relevant in the context, but since we use a list now, this is not true anymore. So, the idea is to say: "*we may get rid of all unrelevant types and restructure our term such that it corresponds to the changes in the context*".

  Example: $[A,\ B,\ C]\ \vdash\ LVar1\ \ |{:}|\ \ B$ means also that
  $[A,\ B]\ \vdash\ LVar1\ \ |{:}|\ \ B$
  or $[B]\ \vdash\ LVar0\ \ |{:}|\ \ B$.

  This property becomes really important when we want to prove the substitution lemma.
  (see Martin's development, Typed_Lambda_Calculus.thy [3])

  Formal version:

  ∀ Γ n S t A. insert-nth $n\ S\ \Gamma\ \vdash\ t\ \ |{:}|\ \ A \Rightarrow$ n≤length Γ
  ⇒ (∀ x. x∈FV t ⟶ x≠n)
  ⇒ Γ ⊢ shift-L (-1) n t |:| $A$

- **Substitution**:

  ∀ Γ A B n s t. $\Gamma\ \vdash\ t\ \ |{:}|\ \ A \Rightarrow \Gamma\ \vdash\ s\ \ |{:}|\ \ B$
  $\Rightarrow \Gamma\ \vdash\ LVar\ n\ \ |{:}|\ \ B$
  $\Rightarrow \Gamma\ \vdash\ subst - L\ n\ s\ t\ \ |{:}|\ \ A.$

Finally, the canonical forms' lemmas for STLC are the following:

is_value_L v ⇒ $\Gamma\ \vdash\ v\ \ |{:}|\ \ Bool \Rightarrow$ v = LTrue ∨ v = LFalse

is_value_L v ⇒ $\Gamma\ \vdash\ v\ \ |{:}|\ \ A \rightarrow B \Rightarrow$ ∃A t. v = LAbs A t

## II.iii. New notations and rules

This part will describe all rules about evaluation of terms, typing of terms and the De Bruijn representation for classic and easy term.
For convenience, examples will often use the type Nat and natural number directly. But to be completely rigorous, we actually need to define them as terms using a **wrapper**, which is a constructor that maps a type directly to elements of another one (here nat to terms) and we have to define associated rules for typing and evaluation (addition and other operators needed as terms and simplification as evaluation rules). Such a definition can be found in the book [1] and the formalized version in Martin's files.

### iii.a. My notations and abbreviations

In previous part about Martin's work, a lot of Isabelle notations were already used and this will continue in main part about my work. Furthermore, when it comes to present, in a readable fashion, shifting and substitution rule (with De Bruijn representation), it will be quite impossible or too long. Therefore, this subpart will summarize first all Isabelle/HOL notations and then my abbreviations (namely for substitution and shifting operator).

List and set operations:

| | | |
|---|---|---|
| [] | : | empty list |
| [$x_1$, ..., $x_n$] | : | list containing $x_1$, ... and $x_n$ |
| a#L | : | constructs a list with a as first elements and then the elements of L (in the same order) |
| L@L1 | : | **appends** the elements of the list L1 to the list L ($[1,3]@[2] = [1,3,2]$) |
| length L | : | the number of elements in list L |
| L!i | : | the ith elements of L (undefined if i≥ length L) ($[1,3]!0 = 1$) |
| L[i:=v] replace i v L | : | replaces the ith element of L by v ($[1,3][0 := 5] = [5,3]$) |
| take n L | : | returns the list containing the nth first element of L (in the same order) |
| insert-nth n A L | : | inserts A at the nth position in L |
| set L | : | returns the set, containing all elements of L |
| f@S | : | the image of the set S by the function f |
| map f L | : | applies the function f to all elements of L |
| foldl f r L | : | iterator on L, applies f to r (accumulator) and the elements of L Sum of elements: foldl ($\lambda$ x r. x + r) 0 $[1,3] = 4$ |

Partial functions and pairs:

| | | |
|---|---|---|
| (a,b) | : | pair composed of a and b |
| fst | : | first projection of a pair (fst (a,b) = a) |
| snd | : | second projection of a pair (snd (a,b) = a) |
| empty | : | partial function with empty domain |
| $\sigma$++$\sigma$1 | : | partial function expansion<br>Example: $[1 \mapsto a]{+}{+}[2 \mapsto b] \;=\; [1 \mapsto a,\ 2 \mapsto b]$<br>But $[1 \mapsto a] + +[1 \mapsto b] \;=\; [1 \mapsto b]$. |
| $\sigma$(a:=b) | : | partial function update, adds the mapping a to b, if a doesn't already belong to the domain of $\sigma$, else changes the result for a<br>Example: $[1 \mapsto a](1 := b) \;=\; [1 \mapsto b]$<br>$[1 \mapsto a](2 := b) \;=\; [1 \mapsto a,\ 2 \mapsto b]$ |
| $\bigodot$ F | : | composition of all functions in F, (F!0)∘(F!1)∘... |
| $\bigodot_T$ F | : | partial function expansion of all functions in F, (F!0)++(F!1)++... |

Abbreviations:

| | | |
|---|---|---|
| sL d c t | : | shift-L d c t |
| suL d c t | : | subst-L j s t |
| sV a c | : | (if a>c then a+d else a) |
| sC a c | : | (if a≤c then (Suc c) else c) |
| if a b c | : | (if a then b else c) |

## iii.b. My work and first incrementation of STLC

My work always started by some copy-paste of Martin's proofs. Then, I incremented the different datatypes (lterm, ltype) and functions (shifting, ...) based on his definitions.

In this part, we will **increment** the feature of **SLTC** with different structures. The most easy ones are pairs and the associated projections. In the same direction, we will also add records and tuples with their projections, which will introduce some difficulties in proofs.

Below, I present the types, term constructors and rules for a language extended from STLC with unit, pairs, records and tuples.

Let start quickly with unit and pairs, since the definitions are the same as in the book [1], except for the De Bruijn representation:

- Types: **Unit** and **A**| × | **B**, where **A** and **B** can be any arbitrary types

- Terms: unit, $\{\!|\textbf{t1,t2}|\!\}$, $\pi_1$ **t1**, first projection and $\pi_2$ **t1**, second projection, with **t1** and **t2** arbitrary terms

- De Bruijn representation: shift-L and subst-L are just **recursively called on the subterms** (example: t1 and t2 are subterms of $\{\!|\text{t1,t2}|\!\}$).
We will go back to this notion of subterms later. Free variable (FV function) just gives back the union of the recursively calls on the subterms.

$$\text{value\_Pair}\ \frac{\text{is\_value\_L v1} \qquad \text{is\_value\_L v2}}{\text{is\_value\_L } (\{\!|v1, v2|\!\})} \qquad \text{value\_unit}\ \frac{}{\text{is\_value\_L unit}}$$

$$\text{PairBeta1}\ \frac{\text{is\_value\_L v1} \qquad \text{is\_value\_L v2}}{\text{eval1\_L } (\pi_1\ \{\!|v1, v2|\!\})\ \text{v1}} \qquad \text{PairBeta2}\ \frac{\text{is\_value\_L v1} \qquad \text{is\_value\_L v2}}{\text{eval1\_L } (\pi_2\ \{\!|v1, v2|\!\})\ \text{v2}}$$

$$\text{Proj1}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L } (\pi_1 \text{ t1})\ (\pi_1 \text{ t1'})} \qquad \text{Proj2}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L } (\pi_2 \text{ t1})\ (\pi_2 \text{ t1'})}$$

$$\text{Pair1}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L } (\{\!|t1, t2|\!\})\ (\{\!|t1', t2|\!\})} \qquad \text{Pair2}\ \frac{\text{is\_value\_L v} \qquad \text{eval1\_L t2 t2'}}{\text{eval1\_L } (\{\!|v, t2|\!\})\ (\{\!|v, t2'|\!\})}$$

Figure 4: Evaluation rules and values for pair and unit

$$\text{Pair}\ \frac{\Gamma \vdash t1 \ |{:}|\ A \qquad \Gamma \vdash t2\ |{:}|\ B}{\Gamma \vdash \{\!|t1, t2|\!\}\ |{:}|\ A\,|\!\times\!|\,B} \qquad \text{Proj1}\ \frac{\Gamma \vdash t\ |{:}|\ A\,|\!\times\!|\,B}{\Gamma \vdash \pi_1\, t\ |{:}|\ A}$$

$$\text{Proj2}\ \frac{\Gamma \vdash t\ |{:}|\ A\,|\!\times\!|\,B}{\Gamma \vdash \pi_2\, t\ |{:}|\ B} \qquad \text{Unit}\ \frac{}{\Gamma \vdash unit\ |{:}|\ Unit}$$

Figure 5: Typing rules for pairs and unit

**The canonical forms' lemmas** for unit and pairs are the following:

$$\text{is\_value\_L v} \Rightarrow \Gamma \vdash v\ |{:}|\ Unit \Rightarrow \text{v} = \text{unit}$$

$$\text{is\_value\_L v} \Rightarrow \Gamma \vdash v\ |{:}|\ A\,|\!\times\!|\,B \Rightarrow\ \exists\text{v1 v2. is\_value\_L v1} \land \text{is\_value\_L v2} \land \text{v} = \{\!|\text{v1,v2}|\!\}$$

The different proofs: Weakening, Substitution, shift-down, Progress, Preservation are really easy for these cases. Indeed, Isabelle's automation takes care of it without any problem, since it suffices to use the induction hypothesis in each case and do some simplification and/or some easy instantiation.

Other interesting, but a little more annoying structures are **tuples** and **record**. Why annoying? Well, because those structures are built upon some collection of terms (therefore, on list or sets of terms) and this will make the proofs and the De Bruijn representation a bit more complicate.

Indeed, the next constructions, **except for the references(pointers)**, will require some *new* operators on lists and sets with related lemmas. *New* means that there is nothing similar defined neither in the common library for lists of Isabelle/HOL nor in the AFP archive (or I might have missed them). Therefore, I proved in the process of my work a lot of useful facts (also some that ends up to be not necessary at end). Those are in the file called, List_extra and a list of all lemmas, with a proof in Isabelle for some of them, is provided in Appendix A.

The global framework for records and tuples is as stands below. The definitions are similar up to list operators to the definitions in the book [1].

- Types: $\llparenthesis$ **TL** $\rrparenthesis$ for tuples and $\llparenthesis$ **L** |:| **TL** $\rrparenthesis$ for records,
  where **L** is a list of strings (fields' name) and **TL** is a list of arbitrary types

- Terms: Tuple **LT**, Π **i t1**, the projection operator for tuples,
  Record **L LT** and ProjR **$\mathfrak{l}$ t1**, the projection for record,
  where **t1** is a term, **i** a natural number, **$\mathfrak{l}$** a string (field name), **L** a list of strings and **LT** a list of terms

- De Bruijn representation: shift-L and subst-L replace the lists of terms by its mapping with the function
  (example: shift-L d c (Tuple L) = Tuple (map (shift-L d c) L)) and are recursively called on subterms for the projections
  **Free variables** are also mapping with FV of the lists of terms for records and tuples and free variables of of subterms for the projections.

The predicate **distinct** on a list L, used in next rules, means that any value in the list is unique (no duplicates).

$$\text{value\_Tuple}\ \frac{\forall\ \text{i<length L. is\_value\_L (L!i)}}{\text{is\_value\_L (Tuple L)}} \qquad \text{value\_record}\ \frac{\forall\ \text{i<length LT. is\_value\_L (LT!i)}}{\text{is\_value\_L (Record L LT)}}$$

$$\text{ProjTuple}\ \frac{1\leq \text{j} \leq \text{length L} \qquad \text{is\_value\_L (Tuple L)}}{\text{eval1\_L } (\Pi\ \text{j (Tuple L)) (L!(j-1))}} \qquad \text{Proj}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L } (\Pi\ \text{i t1}) (\Pi\ \text{i t1')}}$$

$$\text{ProjRCD}\ \frac{\ell \in \text{set LT} \qquad \text{is\_value\_L (Record L LT)}}{\text{eval1\_L (ProjR } \ell \text{ (Record L LT)) (LT!(index L } \ell\text{))}} \qquad \text{ProjR}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (ProjR } \ell \text{ t1) (ProjR } \ell \text{ t1')}}$$

$$\text{Tuple}\ \frac{1\leq \text{j} \leq \text{length L} \qquad \text{eval1\_L (L!(j-1)) t1} \qquad \text{is\_value\_L (Tuple (take (j-1) L))}}{\text{eval1\_L (Tuple L) (Tuple (replace (j-1) t1 L))}}$$

$$\text{RCD}\ \frac{\text{m<length LT} \qquad \text{eval1\_L (LT!m) t1} \qquad \text{is\_value\_L (Record (take m L) (take m LT))}}{\text{eval1\_L (Record L LT) (Record L (replace m t1 LT)}}$$

Figure 6: Evaluation rules and values for tuples and records

$$\text{Tuple}\ \frac{\text{length L} \neq [] \qquad \text{length L = length TL} \qquad \forall\ \text{i<length L.}\ \Gamma \vdash L!i \ |{:}|\ TL!i}{\Gamma \vdash TupleL \ |{:}|\ \llparenthesis\ \text{TL}\ \rrparenthesis}$$

$$\text{ProjT}\ \frac{1\leq \text{j} \leq \text{length L} \qquad \Gamma \vdash t \ |{:}|\ \llparenthesis\ \text{TL}\ \rrparenthesis}{\Gamma \vdash \Pi\, i\, t \ |{:}|\ TL!i}$$

$$\text{RCD}\ \frac{\text{length L} \neq [] \qquad \text{distinct L} \qquad \text{length L = length TL} \qquad \text{length LT = length TL} \qquad \forall\ \text{i<length L.}\ \Gamma \vdash LT!i \ |{:}|\ TL!i}{\Gamma \vdash Record LLT \ |{:}|\ \llparenthesis\ \text{L}\ |{:}|\ \text{TL}\ \rrparenthesis}$$

$$\text{ProjR}\ \frac{\text{distinct L} \qquad \ell \in \text{set L} \qquad \text{length L = length TL} \qquad \Gamma \vdash t \ |{:}|\ \text{rcdTLTL}}{\Gamma \vdash ProjR\, l\, t \ |{:}|\ TL!(indexL\ l)}$$

Figure 7: Typing rules for pairs and unit

The canonical forms' lemmas for unit and pairs are the following:

$$\text{is\_value\_L v} \Rightarrow \Gamma \vdash v \ |{:}|\ \llparenthesis\ \text{TL}\ \rrparenthesis \Rightarrow \ \exists\ \text{L. is\_value\_L (Tuple L)} \wedge \text{v = Tuple L} \wedge \text{length TL = length L}$$

$$\text{is\_value\_L v} \Rightarrow \Gamma \vdash v \ |{:}|\ \llparenthesis\ \text{L}\ |{:}|\ \text{TL}\ \rrparenthesis \Rightarrow \ \exists\ \text{LT. is\_value\_L (Record L LT)} \wedge \text{v = (Record L LT)}$$

The common lemmas goes as followed (IH stands for induction hypothesis):

- **Weakening, shift-down and Substitution:**

  These cases only consist in applying the induction hypothesis and some extra lemma:
  $\forall$i < length L. (map (shift-L 1 n) L)!i = shift-L 1 n (L ! i)

- **Progress:** For tuple and record, progress can be proved by a nested induction on the list of term.
  For the projections, we need the canonical forms for tuple and record, in order to show that the term evaluates.

- **Preservation:**

  For these cases, preservation is quite easy since we don't have any substitution.
  Namely, we only need to use inversion on the evaluation predicate and we get all the information we need to determine the type of the next step.
  Since we are dealing with replace, I proved a sublemma nth_replace using some facts about the take and drop functions:

  $\forall$i < length L. (replace n t L)!i = (if i=n then t else L!i)

Finally, all definitions of FV, shift-L, subst-L and other functions related to the De Bruijn representation of terms can be found in Appendix B.

# III. Formalization

This part will present the main formalization results of my thesis.
All subsections will follow the same structure. I'll start by explaining the informal ideas and then I'll show its formal version. Since we are dealing with $\lambda$-calculi, all redundant rules will be omitted. To find those rule, please refer to the New notations and rules and Martin's work and notations sections.

Furthermore, the global structure of my work for each chapter will follow the pattern below with some optional entries:

1. new term(s) and type(s)
2. De Bruijn representation's changes (shifting, substitution, free variables) (optional)
3. new operators (filling, pvars, ...) (optional)
4. evaluation rules (inspired from the book [1] with some changes due to the representation)
5. typing rules (inspired from the book [1] with some changes due to the representation)
6. theorems and lemmas (not written down in the book [1], using Martin's proof for the cases already proven)

## III.i. Sequence and derived forms

In this subpart, we want to add sequence of instructions to our language (STLC with unit). Notice that we can already simulate the behaviour of a sequence, i. e. execute an instruction t1 then an instruction t2. In $\lambda$-calculus, a program is just a game of giving values to functions and repeating it until we don't have any function on the left-hand side any more.

($\lambda$ a. c) ($\lambda$ u. b) a = ($\lambda$ a. c) b = c is the general picture.
An instruction should, in the same fashion, give some kind of information to the next one. We will use the notation **t1 ;; t2** for a sequence of two instructions t1 and t2. Furthermore, in order to have the simpliest behaviour, we will use **unit** as final value returned by an instruction.
Now the first instruction reduces to unit and unit is given as argument to a function executing the second instruction.
This way to implement sequence is called **derived form**, since we don't define any new term and only formulate its behavior with the current terms.

Formally speaking, we get the following definitions:

- **Derived form:** t1 ;; t2 $\triangleq$ LApp (LAbs Unit (shift-L 1 0 t2)) t1
  Since we use De Bruijn representation, we have to adapt the function (executing t2), so that the evaluation behaves as expected. That is t1 reduces to unit: unit ;; t2 and this reduces to t2.
- Other option is to add a new term: Seq t1 t2, with all new rules and definitions below

When we define the new term constructor Seq, we need to increment our rules' set:

- shift-LE: shift-LE d c (Seq t1 t2) = Seq (shift-LE d c t1) (shift-LE d c t2)
- subst-LE: shift-LE d c (Seq t1 t2) = Seq (shift-LE d c t1) (shift-LE d c t2)
- Evaluation:

  Seq unit t2 $\rightarrow^E$ t2  t1 $\rightarrow^E$ t1' $\Rightarrow$ Seq t1 t2 $\rightarrow^E$ Seq t1' t2

- Typing: a sequence has type A if the first instruction has type UnitE and the second one has type A

$$\Gamma \vdash^E t1 \;|{:}|\; UnitE \Rightarrow \Gamma \vdash^E t2 \;|{:}|\; A \Rightarrow \Gamma \vdash^E Seq\, t1\, t2 \;|{:}|\; A$$
$$\Gamma \vdash t1 \;|{:}|\; Unit \Rightarrow \Gamma \vdash t2 \;|{:}|\; A \Rightarrow \Gamma \vdash t1 \,;;\, t2 \;|{:}|\; A$$

Finally, assuming equivalence between a language with internal sequence (using derived forms) $\lambda^I$ and one with external sequence (new term Seq) $\lambda^E$ seems to be correct. But this is **not completely true**. Indeed, the $\lambda^E$ language contains more terms than $\lambda^I$, which also implies more information. Therefore, translating a term from $\lambda^E$ to $\lambda^I$ brings up ambiguities (Seq and derived forms in $\lambda^E$ become derived forms in $\lambda^I$). This translation is represented by a function e that converts terms from $\lambda^E$ to $\lambda^I$ (replacing recursively Seq by corresponding derived form).

**Remark :** all terms of $\lambda^E$ are built with the same constructor as $\lambda^I$ with E at the end or with the constructor Seq
(example: LVar becomes LVarE).

Then, we define **e**:

```
e t   :=   match t with
           | LVarE x        ⇒   LVar x
           | LAppE t1 t2    ⇒   LApp (e t1) (e t2)
           | LAbsE A t      ⇒   LAbs A (e t)
           | unitE          ⇒   unit
           | Seq t1 t2      ⇒   (e t1 ;; e t2)
```

Now the expected results are:

1. $\forall$ t t1. (e t $\rightarrow^E$ e t1) $\Longleftrightarrow$ (t $\rightarrow$ t1)
2. $\forall$ t A. $\Gamma \vdash^E e\, t \;|{:}|\; A \Longleftrightarrow \Gamma \vdash t \;|{:}|\; A$

Well, that's a shame but **No**, because it has an easy counter-example:

Let t1, t2 be arbitrary terms such that:

e (Seq unitE (LAppE (LAbsE Unit (shift-LE 1 0 t1)) t2)) → e t1 ;; et2 ≜ unit ;; (e t1;; e t2) → (e t1;; e t2),
but e t1 ;; et2 can be **either** e (LAppE (LAbsE Unit (shift-LE 1 0 t1)) t2) **or** e (Seq t2 t1)
Seq unitE (LAppE (LAbsE Unit (shift-LE 1 0 t1)) t2) doesn't reduce to the second possibility.

The fact 2 can be proved without any difficulties, but even through we have 2, we can't derive 1.
With 2 and Preservation, we could have tried to prove:

$$3.\ \forall\, \text{t t1}.\ \Gamma \vdash^{E} e\ t\ |{:}|\ A \Rightarrow (\text{e t} \rightarrow^{E} \text{e t1}) \Longleftrightarrow (\text{t} \rightarrow \text{t1})$$

But once more, in the previous counter-example,
LAppE (LAbsE Unit (shift-LE 1 0 t1)) t2) and Seq t2 t1 have the same type by definition, so the type doesn't give us any further informations, that could make e injective in this particular case.
Namely, e being a bijection is required (since it would justify the existence of an equivalence). But if we consider both languages as sets of terms, they have obviously different cardinalities, which means that no bijection exists between them. The only hope would be to come up with a clever restriction.
This concludes this introducing parts, we will now dive into the biggest part of my work (the long chapter 11 of the book).

## III.ii. From let to lists

In this part, I will present the lemmas and formalisms, that I developped in details. Namely, the biggest part is obviously about the changes in the context for typing and the new predicates. This part of my work corresponds to the chapter 11 in the original book [1]

But, the De Bruijn representation and the proofs associated FV_subst, FV_shift, ... still remain pretty tricky and required some thinking. Furthermore, Isabelle seems not so at ease with complex set theory (replacement,
set comprehension, $\bigcup$, ...), which leads to horrible proofs (more than 200 lines).

In this part, I will also introduce some leads to solve an exercise in the book [1] (without solution), asking to add a particular pattern feature to the language. For convenience, the language definition(includes the De Bruijn representation and associated functions (shift-L, ...)) is in a different file Lambda_calculus.thy. This subpart will present incrementially all required features, while the Isabelle file proves that all features are compatible(simulation of some small language).

Normalization chapter of the book [1] introduces inconsistency with some unprecise definitions, that I replaced by correct ones. The proof sketch remains pretty much similar, but I will explained the main differences.

Finally, formalizing references(pointers) was some piece of cake compared to some structures, that I expected to be easy(example: pattern matching on disjoint sums). Last part also presents a formalization of exception handling(Java-like structure, exercise) with some limitations and the development based on the results in the book (dummy term error).

For the moment, let us start with Let binder and some surprisingly vicious structures(pattern matching on sum and variant types).

## ii.a. Let binder, disjoint sums and variants

In the popular languages like ML, there is always a system that allows us to do some pattern matching on terms. This goes from simple example, like the instructions let: let f = 1+2 in f − 5 + f, to more advanced matchings (pairs, record, ...): let (a,b) = (f 5,g 3) in (b+a), b−a).
Even through the difference between the two previous examples is subtle, there is one. Indeed, the first let only does some replacement in another term (no structural analysis). For this reason, it can be modelled with a **derived form**:
($\lambda$ x. t1) t2 $\triangleq$ let x = t2 in t1. But for the second one, we need to make sure that the term on the right-hand side of the = has the expected structure.

Let us begin with the easier case, the Let instruction, which behaves like its derived forms. Therefore, the rules for **Let** are similar to the rules for
the abstraction and the beta rule (apply a function(abstraction) to an argument). Now, with the De Bruijn representation, there is a difference, since we don't give a name to variables anymore. That's why the **binder index x** will be more important.

**Formal definitions:**

- Types: no changes
- Terms: Let var x := t1 in t2, with x an index (natural number), t1 and t2 arbitrary terms
- De Bruijn representation:

**shift-L**:

sL d c (Let var x:= t1 in t2) = (if (x>c) (Let var (x+d):= sL d c t1 in sL d c t2)
(Let var x:= sL d c t1 in sL d Suc c t2))

**subst-L**:

suL j s (Let var x:= t1 in t2) = (Let var x := (suL j s t) in (suL (if (j≥ x) (Suc j) j) (sL 1 x s) t1))

**FV**:

FV (Let var x := t1 in t2) = ($\lambda$ y. if (y>x) then y - 1 else y)@(FV t2 - $\{x\}$) ∪ FV t1

$$\text{Let}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (Let var x := t1 in t2) (Let var x := t1' in t2)}}$$

$$\text{LetV}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (Let var x := v in t2) (shift-L (-1) x (subst x (shift-L 1 x v) t2))}}$$

Figure 8: Evaluation rules for Let

$$\text{Let}\ \frac{\Gamma \ \vdash \ t1 \ \ |:| \ \ A \qquad \text{insert-nth}\ x\ A\ \Gamma \ \vdash \ t2 \ \ |:| \ \ B}{\Gamma \ \vdash \ \ Let\ var\ x \ \ := t1\ in\ t2 \ \ |:| \ \ B}$$

Figure 9: Typing rules for Let

Since the instruction Let doesn't add any value, there is no new canonical form. Now, everything seems easy. But that's only the case for **Weakening** and **Progress**. Indeed, when we need to use statements about free variables, things become a bit more complicate.

The first case is in **shift-down**. Since we have two subcases to study, thanks to the shifting function: **x≤c** and **x>c**. In the case **x>c**, we need to show that a lemma H1: ∀ y∈FV t2. x≠n, which is not difficult, if we have the right equation for Let terms in the definition of FV.
Now, our goal is to show:

Γ ⊢ Let var (x-1) := (shift-L (-1) n t1) in (shift-L (-1) n t2) |:| $B$

To prove it, we will use the induction hypotheses, that requires H1 and
∀ y∈FV t1. x≠n, which is derived by simplification of our hypothesis:
∀ y∈FV (Let var x := t1 in t2). x≠n. Finally, we only apply the **introduction rule Let** of the typing's predicate.

We would hope the other case to be symmetric, which is almost the case. Indeed, we will need another lemma H2: ∀ y∈FV t2. x≠Suc n, since our goal is now:

$$\Gamma \ \vdash \ \ Let\ var\ x \ \ := \ \ (shift-L\ (-1)\ nt1)\ in\ (shift-L\ (-1)\ (Sucn)\ t2) \ \ |:| \ \ B$$

At this point, everybody would expect the definition of FV to be: FV(Let var x := t1 in t2) = FV t1 ∪ (FV t2 - $\{x\}$), but then there is now way to prove H2.

To be more precise, H1 is direct and H2 unprovable, because we don't have any information about the correlation between FV t2 and x. The same problem occurs in the LAbs case, if we don't take the image by the predecessor operator. Now, in our case, we have x, that is binded, not 0. So, we need to shift our free variables depending on x.

FV (Let var x := t1 in t2) = ($\lambda$ y. if (y>x) then y - 1 else y)@(FV t2 - $\{x\}$) ∪ FV t1. @ is used as notation in this report, but in Isabelle, the notation for the replacement operator is different. It can be found in the Appendix B, containing the Isabelle version of the functions' definition.

**This intuition comes from the LAbs case**. Actually, we could have written ($\lambda$ y. if (y>0) then y - 1 else y) as predecessor function. Since **natural number are always greater or equal than 0**, this will let the case 0, but we apply this function to FV t - {0}, so everything is fine.

From the previous proof description and explanation, it becomes clear that the biggest difficulty comes from doing proofs with sets and replacement. Actually, it is not so hard on paper, but if we want to formalize it, that's painful. It takes more than 250 lines to formalize a generalized version of the reasoning. This lemma can be found in the corresponding theory file(check picture after the table of contents) under the name **Binder_FV_shift**. This generalization is pretty helpful, since next pattern matching on disjoint sums use exactly the same reasoning with two pairs index, term, instead of one in the case of Let. The same process is done to prove FV_subst with **Binder_FV_subst**, which requires as much effort.
These lemmas are required to be able to prove preservation. Indeed, when we want to prove that the type is preserved a function application, for example, we will have to prove that an expression of the form: shift-L (-1) c (subst-L c (shift-L 1 c t1) t2, for some c t1 t2, has the same type as the term before application. Now, we know that we can get the type of substituted term with the **lemma Substitution**. So what is left is proving shift-L (-1) c t', where t' is a substitution, has the correct type. For this task, we need **the lemma Shift-down**, but remember that it requires knowledges about the free variables contained in the term:
∀ y∈FV t'. y≠c.

But we don't know the form of t', outside the fact that it is subst-L a ta tb, for some a ta tb. So we need arbitrary knowledges about free variables in shifted and substituted terms. That's the detailed reason why we need FV_shift and FV_susbt, even through it consists in most of the workload (interactions between definitions of FV and shifting, substitution functions imply that changing the rules of one function might help to prove one lemma, but makes the next one unprovable most of the time). It delays obviously the formalization process a lot.

We can then remove the head of the context, most of the time (also what we need to do). For the case **Let**, we get by induction hypotheses and the lemma Substitution, that the subst-L part of the term is well-typed in some arbitrary context Γ with a type A inserted at the xth position, but our goal is to have a proof that after applying shift-L (-1) x to it, the new term obtained is **well-typed** in Γ. With FV_shift and FV_subst, we can easily check that x is not free in the subst-L part and then apply **Shift-down**.

The previous reasoning can also be applied to more **specific matchings**, namely for **disjoint sums and variants**. **Sum elements** are term that may store a term from two different types. That is, for example, a boolean and a natural number: inl LTrue and inr 5 may have the same type. Well, since we can only infer one information about the type of such term, we would loose, as described in the book [1], **uniqueness of types**: "*every well-typed term has a unique type*". Why?
Just have a look at the previous example: inl LTrue, contains a boolean, but we could put any other type as second part. To get rid of this ambiguity, Professor Pierce introduces Ascription: 1 as Bool is not well-typed while 1 as Nat would be. So, inl LTrue as Bool| + |Nat can only be typed as a sum of boolean and natural number and then inr 5 as Bool| + |Nat has the same type. With this type, we know that we can do some branching (pattern distinction):
Case t of Inl x ⇒ t1 | Inr y ⇒ t2, for arbitrary t, t1, t2 terms and x,y indexes(ℕ).
The formal definition follows the same principle as Let with the same difficulties. For instance, proofs for FV_shift and FV_subst are pretty short thanks to the generalized lemmas quoted in the Let case (Binder_FV...).
To make the implementation of the De Bruijn representation more readable, the case constructor above is replaced by **CaseS** (that takes the same arguments).

**Formal definitions:**

- Types: A| + |B with arbitrary types A and B
- Terms: t as A, inl t as A, inr t as A,
  Case t of Inl x ⇒ t1 | Inr y ⇒ t2,

  with t, t1, t2 arbitrary terms and x,y arbitrary indexes(ℕ) and A an arbitrary type.
- De Bruijn representation:

**shift-L**: sL d c (t as A) = (sL d c t) as A
sL d c (inl t as A) = inl (sL d c t) as A
sL d c (inr t as A) = inr (sL d c t) as A

sL d c (CaseS t x t1 y t2) = CaseS (sL d c t) (sV x c) (sL d (sC x c) t1)
(sV y c) (sL d (sC y c) t2)

**subst-L**: sL d c (t as A) = (sL d c t) as A
suL j s (inl t as A) = inl (suL j s t) as A
suL j s (inr t as A) = inr (suL j s t) as A

suL j s (CaseS t x t1 y t2) = CaseS (suL j s t) x (suL (sC x j) (sL 1 0 s) t1)
y (suL (sC y) (sL 1 0 s) t2)

**FV**: FV (t as A) = FV t
FV (inl t as A) = FV t
FV (inr t as A) = FV t

FV (CaseS t x t1 y t2) = $\{x, y\}$ ∪ FV t ∪ ($\lambda$ z. if (z>x) then z - 1 else z)@(FV t1 - $\{x\}$)∪ ($\lambda$ z. if (z>y) then z - 1 else z)@(FV t2 - $\{y\}$)

$$\text{Ascribe1}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (t1 as A) (t1' as A)}} \qquad \text{Ascribe}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (v as A) v}}$$

$$\text{Val\_Inl}\ \frac{\text{is\_value\_L v}}{\text{is\_value\_L (inl v as A)}} \qquad \text{Val\_Inr}\ \frac{\text{is\_value\_L v}}{\text{is\_value\_L (inr v as A)}}$$

$$\text{inl}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (inl t1 as A) (inl t1' as A)}} \qquad \text{inr}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (inr t1 as A) (inr t1' as A)}}$$

$$\text{CaseInl}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (CaseS (inl v as A) x t1 y t2) (sL (-1) x (subst-L x (sL 1 x v) t1))}}$$

$$\text{CaseInr}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (CaseS (inr v as A) x t1 y t2) (sL (-1) y (subst-L y (sL 1 y v) t2))}}$$

$$\text{CaseS}\ \frac{\text{eval1\_L t t'}}{\text{eval1\_L (CaseS t x t1 y t2) (CaseS t' x t1 y t2)}}$$

Figure 10: Evaluation rules for As and disjoint sums

$$\text{Ascribe}\ \frac{\Gamma \vdash t\ \ |{:}|\ \ A}{\Gamma \vdash t\ as\ A\ \ |{:}|\ \ A}$$

$$\text{Inl}\ \frac{\Gamma \vdash t\ \ |{:}|\ \ A}{\Gamma \vdash inl\ t\ as\ A\,|+|\,B\ \ |{:}|\ \ A\,|+|\,B} \qquad \text{Inr}\ \frac{\Gamma \vdash t\ \ |{:}|\ \ B}{\Gamma \vdash inr\ t\ as\ A\,|+|\,B\ \ |{:}|\ \ A\,|+|\,B}$$

$$\text{CaseS}\ \frac{\Gamma \vdash t\ \ |{:}|\ \ A\,|+|\,B \qquad \text{insert-nth}\ x\ A\ \Gamma \vdash t1\ \ |{:}|\ \ C \qquad \text{insert-nth}\ y\ B\ \Gamma \vdash t2\ \ |{:}|\ \ C}{\Gamma \vdash \text{CaseS}\ t\ x\ t1\ t2\ \ |{:}|\ \ C}$$

Figure 11: Typing for As and disjoint sums

The canonical forms' lemmas for disjoint sums is the following:

is_value_L v ⇒ $\Gamma \vdash v\ \ |{:}|\ \ A\,|+|\,B \Rightarrow$
(∃ v1. is_value_L v1 ∧ v = inl v1 as A| + |B) ∨ (∃ v1. v = inr v1 as A| + |B ∧ is_value_L v1)

Proofs for the cases **As** and **disjoint sums** are really easy. In contrast to these cases, the matching on disjoint sums **CaseS** follows the same reasoning as **Let**, but with two indexes, this time.

Finally, the last structure of this kind is called variant. It's a structure that can store any element of an arbitrary list of types. Actually, it looks like a record, that contains only one value among all possible ones and it is a more general version of disjoint sums. Therefore, there is also a structure, that matches on a variant and return some term depending on the actual content of the variant.

Example: <'x':=5> as < ['x','cdt'] |,| [Nat, Bool] > and
<'cdt':=LTrue> as < ['x','cdt'] |,| [Nat, Bool] > are both variants.

This kind of element is really helpful, when we want to use only one element that changes along the execution. Money conversion is an example presented in the book [1] (page 139). The matching structure will be abbreviated as **CaseV**.

**Formal definitions:**

- Types: <L|, |TL> with arbitrary types' list L and types' list TL
- Terms: <l:=t> as A where l is an arbitrary string (field's name), t an arbitrary term and A an arbitrary type

  Case t of <L⇒B>

  with t, an arbitrary term, L an arbitrary strings' list and B an arbitrary list of pairs composed by an index and a term.
- De Bruijn representation:

  **shift-L**: sL d c (<l:=t> as A) = <l:=(sL d c t)> as A

sL d c (CaseV t L B) = CaseV (sL d c t) L (map ($\lambda$ p. (sV (fst p) c, shift-L d (sC (fst p) c) (snd p))) B)

**subst-L**: suL j s (<l:=t> as A) = <l:=(suL j s t)> as A

suL j s (CaseV t L B) = CaseV (suL j s t) B (map ($\lambda$ p. (fst p, suL (sC (fst p) j) (sL 1 (fst p) s) (snd p))) B)

**FV**: FV (<l:=t> as A) = FV t

FV (CaseV t L B) = FV t ∪
foldl ($\lambda$ x r. x ∪ r) ∅ (map ($\lambda$p. image ($\lambda$ y. if (y>(fst p)) (y - 1) y) (FV (snd p) - {$(fstp)$})) B)

$$\text{Val\_Variant}\ \frac{\text{is\_value\_L v}}{\text{is\_value\_L (<l:=v> as A)}}$$

$$\text{Variant}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (<l:=t1> as A) (<l:=t1'> as A)}}$$

$$\text{CaseVar}\ \frac{\text{i<length L} \qquad \text{is\_value\_L v}}{\text{eval1\_L (CaseV (<L!i:=v> as A) L B) (sL (-1) (fst(B!i)) (subst-L (fst(B!i)) (sL 1 (fst(B!i)) v) (snd(B!i))))}}$$

$$\text{CaseV}\ \frac{\text{eval1\_L t t'}}{\text{eval1\_L (CaseV t L B) (CaseV t' L B)}}$$

Figure 12: Evaluation rules and value for variants

$$\text{Variant}\ \frac{\text{i<length L} \qquad \text{distinct L} \qquad \text{length L=length TL} \qquad \Gamma \vdash t1 \ |{:}|\ TL!i}{\Gamma \vdash <(L!i) := t1 > \ as\ <L\,|,|\,TL> |{:}| <L\,|,|\,TL>}$$

P abbreviates ∀ i<length L. insert-nth $(fst(B!i))\ (TL!i)\ \Gamma \vdash\ snd(B!i)\ |{:}|\ C \wedge$ fst(B!i)≤length Γ

$$\text{CaseV}\ \frac{\Gamma \vdash\ t\ |{:}| <L\,|,|\,TL> \qquad \text{distinct L} \qquad \text{length L=length B} \wedge \text{length L=length TL} \qquad \text{P}}{\Gamma \vdash\ CaseV\ t\ L\ I\ LT\ \ |{:}|\ C}$$

Figure 13: Typing for variants

Professor Pierce doesn't define a value for the variant type. But in this case, then we can't prove **Progress**. Indeed, when t is a a variant, if the inner term is a value, the term t doesn't evaluate anymore and with Professor Pierce setting, it can't be a value. So **we have to** define a value of type variant and the following **canonical form**:

is_value_L v ⇒ $\Gamma \vdash\ v\ |{:}| <L\,|,|\,TL> \Rightarrow$
(∃ v1 i. is_value_L v1 ∧ v = <L!i:=v1> as <L|, |TL> ∧ i<length L)

Once more, the statement is the same, there is a big contrast in term of difficulty between proofs for the variant and for its matching. Namely, the matching has even longer and harder proofs than the disjoint sums. The reason is obvious, we deal with a list of possibilities and the set obtain with (foldl ...) is some kind of $\bigcup$ over smaller sets, which introduces a lot of existential quantifier in our proofs. Unfortunately, since we end up with $\bigcup$ operator, we cannot use our lemmas (Binder_FV...). Actually, we have a goal that contains (($\bigcup$ ...) ∪ ...) on both sides of the equality. So we can't just push the part after the ∪ sign under the $\bigcup$. So the only solution is to start back from scratch.

To make the problem even more annoying, since we have to reduce to first-order premises and goals, when we deal with big sets equality, we are confronted to DNF-factorisation problems. Namely, when confronted to equality between unions of sets, human will try to associate sets together without unfolding all definition until reaching a first-order statements where the only set operator remaining is membership.
To win time, I chose to go down to the deeper simplifications (complete first-order formula, with only membership remaining (from set theory)). But, even with this method, the work is long and painful.

This will conclude this part on expected easy structures, that ends up with having *complicate* proofs on computer, namely *thanks to* their De Bruijn representation. In next section, I'll present an exercise proposed by the book [1]: general pattern matching, that generalizes what has been done in this part. Well, we would expect it, since the generalization from disjoint sums to variants brings up a lot of difficulties, further generalization will bring up further troubles (namely a lot of new formalisms).

## ii.b. Patterns

General patterns follows all the same principle, so why don't define it once and then add new patterns(separated definition) only?
The answer is that it is indeed a possibility. This is given as exercise in the book. Wolfram Kahl presented a solution with a slightly different definition in one of his publication [4]. In order to have an intuition of what we want to do, let us first have a look at some example.

Let (a,b)=(d,c) in a + b is a classic pattern matching on pairs, but what are we doing in the most general sense. First we have **a pattern** (a,b), which always mimics the constructor of the matched structure(pairs, in this case). Then we analyse a term namely (d,c), which is a pair, but could have been anything else (it would only be well-typed if it reduces to a pair of values). Finally, we have a term containing or not **references** to the term matched (**patterns variables**). The evaluation will only determine some **filling function**, a substitution function, and apply it to a + b.

So if we want to generalize, a **pattern matching** is composed of so-called **pattern constructors** and will substitute variables. The instruction **Let** is a good example and the easiest one. But, the naive intuition is to say: let us use variables and we will extend the definition of Let or disjoint sum. That's probably a good idea, but we end up then with pretty long term with a lot of index and furthermore, shifting and substitution rules are going to be really complicate. The trick is to say, what about doing something like inlining in C++. The goal is to have **tokens** in the code, that will be only affected by **pattern filling**.
Let us have a look at the previous example, but with this strategy:

Let ($\star$1,$\star$2)=(d,c) in $\star$1 + $\star$2.

Now, $\star$1 and $\star$2 are not variables in the sense of the language(term), but only in the sense of the matching system. Shifting this term is easy, because like mentioned before, pattern variables are not affected by all operations except **filling**. How can it work? The pattern variables will only be filled when the **matched term** is a value and this term is affected by all operations.

To summarize, a **pattern matching** can be formalized as **a type for patterns**, **a wrapper for pattern variables** and a predicate to determine the filling function to apply. Even through this method seems easy, it requires a lot of work and especially a lot of attention when it comes to how to handle the filling function. In this sense, the version with variables of the language is certainly unreadable, for really complicate terms, but is based on an already clearly defined framework, so some tricks can be copied from already known cases (Let, disjoint sums, variants).

The cost will also be the same, while this method might avoid some really unpleasant proofs (FV_shift, FV_subst cases, remember more than 200 lines for CasesV).
Formally, we define first the type Lpattern:

Lpattern ::= **V** k **as** A (with k a natural number and A a type)
| **RCD** L PL (with L a list of strings (fields' name) and
PL a list of Lpattern)

We could have used the same formulation as Wolfram Kahl in his paper [4], with an abstract constructor, but the exercise was to define a pattern matching for variables and records.
Then we need a **wrapper**(reminder: new constructor taking only one argument): <| p |>, where p is a **Lpattern**.
The pattern matching structure is defined as follows: Let pattern p := t1 in t2, with t1, t2 terms and p a Lpattern. Finally, the wrapper is not affected by shifting and substitution and it is applied recursively with same arguments to the subterms in the case of the new matching structure.

But, we need some way to determine the substitution to apply during the reduction, based on a Lpattern p, a matched term t1 and a substitution partial function for indexes $\sigma$:

$$\text{M\_Var}\ \frac{\text{is\_value\_L t1}}{\textbf{Lmatch}\ \text{(V k as A) t1}\ [k \mapsto t1]}$$

**P1** abbreviates length L = length LT ∧ length F = length PL ∧
length PL = length LT and
**P2** abbreviates ∀ i<length PL. **Lmatch** (PL!i) (LT!i) (F!i)

$$\text{M\_RCD}\ \frac{\text{distinct L} \qquad \textbf{P1} \qquad \text{is\_value\_L (Record L LT)} \qquad \textbf{P2}}{\textbf{Lmatch}\ \text{(RCD L PL) (Record L LT)}\ (\bigodot \text{F})}$$

Furthermore, we need to define a new context to track the type of each pattern variable, which is also a partial function. Then, it is easy to deduce that we need some other predicate to extract this context from the pattern specified in the branching term:

$$\text{M\_Var}\ \frac{}{\textbf{Lmatch\_Type}\ \text{(V k as A) t1}\ [k \mapsto A]}$$

**P1** abbreviates length Tx = length PL ∧ length L = length PL
**P2** abbreviates ∀ i<length PL. **Lmatch_Type** (PL!i) (Tx!i)

$$\text{M\_RCD}\ \frac{\text{distinct L} \qquad \textbf{P1} \qquad \textbf{P2}}{\textbf{Lmatch\_Type}\ \text{(RCD L PL)}\ (\bigodot_T \text{Tx})}$$

The rules for evaluation and typing are then defined, such that only pattern variables can be instantiated in terms (RCD should only appear after Let pattern). Since **Lmatch** provides a substitution function for index, we defined a function **fill** that applies it properly on terms. See Appendix B for more details.

$$\text{LetP}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (Let pattern p:=t1 in t2) (Let pattern p:=t1' in t2)}}$$

$$\text{LetPV}\ \frac{\text{Lmatch p v } \sigma \qquad \text{is\_value\_L v}}{\text{eval1\_L (Let pattern p:=v in t2) (fill } \sigma \text{ t2)}}$$

Figure 14: Evaluation rules for Let pattern

coherent p B is a predicate stating that all pattern variables in p are distinct and if p is RCD L PL for some L and PL, then all patterns p' in PL are such that coherent p' (TL!i) and B = ⦇ L |:| TL ⦈, for some list of types TL.

$$\text{LetPattern}\ \frac{\text{coherent p B} \qquad \text{Lmatch\_Type p t1 } \sigma 1 \qquad \Gamma \vdash ⦇\, t1 \mid ; \mid \sigma\, ⦈ \mid : \mid B \qquad \Gamma \vdash ⦇\, t2 \mid ; \mid \sigma + + \sigma 1\, ⦈ \mid : \mid B}{\Gamma \vdash ⦇\, Let\ pattern\ p := t1\ in\ t2 \mid ; \mid \sigma\, ⦈ \mid : \mid B}$$

$$\text{PatternVar}\ \frac{\text{k} \in \text{dom } \sigma \qquad \sigma \text{ k} = \text{A}}{\Gamma \vdash ⦇ <\mid\text{V k as A}\mid> \mid ; \mid \sigma\, ⦈ \mid : \mid A}$$

Figure 15: Typing for Let pattern and pattern variables

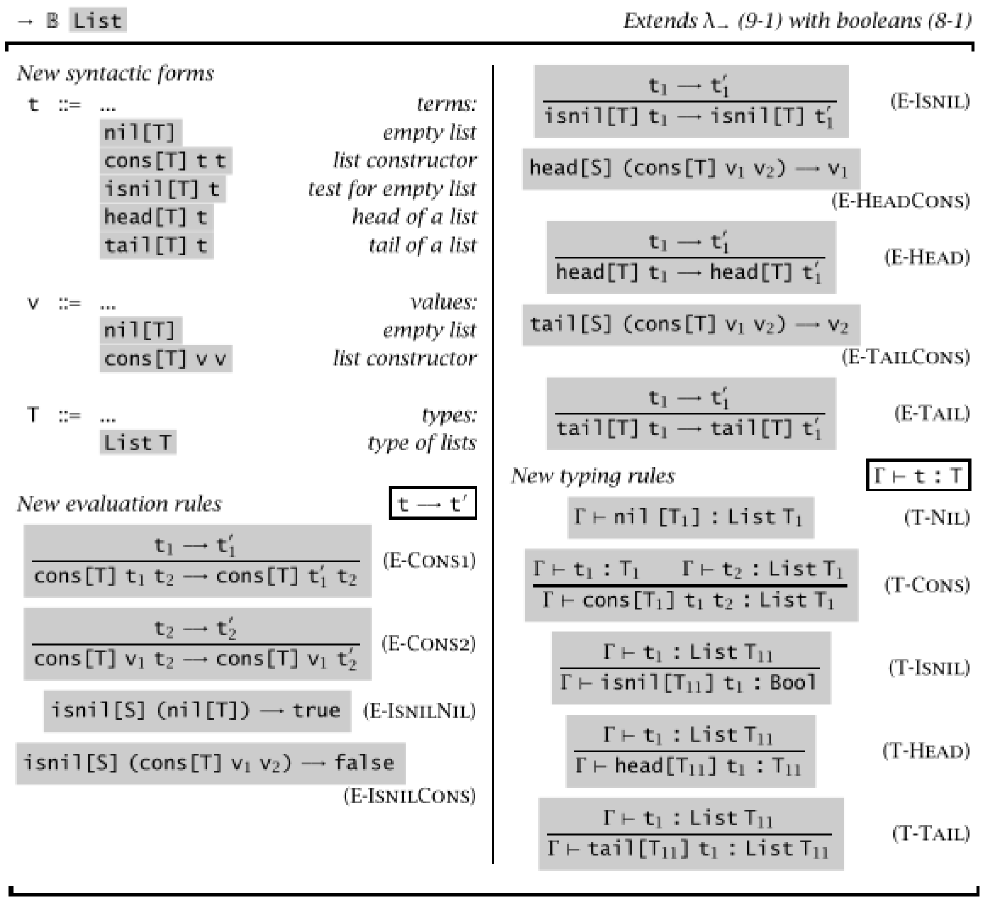
This is unfortunately unfinished work, since now we need to come up with lemma for **Substitution** (find a precondition such that you can substitute some term s, well-typed with $\sigma$ in some term t well-typed with $\sigma 1$, such that the result is well-typed with $\sigma 1$). The same goes for **Weakening**, since we need to know that we can use **some other context with a bigger domain**, but we need to be careful, since it must not coincide with the context generated by Let pattern. I propose my development until this point, allowing someone to try to prove it with some lemmas.

This concludes this subpart on pattern matchings, we will know talk about lists.

## ii.c. List

List is among the most common features in our favorite programming languages today, but it remains a bit tricky when we look at its implementation in details. Especially, the best way to define lists is to use more advanced language (with recursive types), in which it is only a recursion with parameter $\alpha$ on a disjoint sum of Unit and the product of the type of elements of the list and the parameter $\alpha$. I won't go into details, but just know that $\alpha$ can be replaced by the whole expression, recursive pattern. The formal definition: list A $\triangleq \mu\ \alpha$. Unit| + |(A| × | $\alpha$).

Obviously, recursive types should be defined formally first. In an attempt to avoid using those advanced technics, Professor Pierce proposes the following definitions:



$\rightarrow\ \mathbb{B}$ `List` *Extends $\lambda_{\rightarrow}$ (9-1) with booleans (8-1)*

*New syntactic forms*

| | | |
|---|---|---|
| `t ::=` | ... | *terms:* |
| | `nil[T]` | *empty list* |
| | `cons[T] t t` | *list constructor* |
| | `isnil[T] t` | *test for empty list* |
| | `head[T] t` | *head of a list* |
| | `tail[T] t` | *tail of a list* |
| `v ::=` | ... | *values:* |
| | `nil[T]` | *empty list* |
| | `cons[T] v v` | *list constructor* |
| `T ::=` | ... | *types:* |
| | `List T` | *type of lists* |

*New evaluation rules* $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{cons[T]}\ t_1\ t_2 \longrightarrow \texttt{cons[T]}\ t_1'\ t_2} \quad \text{(E-CONS1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\texttt{cons[T]}\ v_1\ t_2 \longrightarrow \texttt{cons[T]}\ v_1\ t_2'} \quad \text{(E-CONS2)}$$

$$\texttt{isnil[S] (nil[T])} \longrightarrow \texttt{true} \quad \text{(E-ISNILNIL)}$$

$$\texttt{isnil[S] (cons[T]}\ v_1\ v_2) \longrightarrow \texttt{false} \quad \text{(E-ISNILCONS)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{isnil[T]}\ t_1 \longrightarrow \texttt{isnil[T]}\ t_1'} \quad \text{(E-ISNIL)}$$

$$\texttt{head[S] (cons[T]}\ v_1\ v_2) \longrightarrow v_1 \quad \text{(E-HEADCONS)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{head[T]}\ t_1 \longrightarrow \texttt{head[T]}\ t_1'} \quad \text{(E-HEAD)}$$

$$\texttt{tail[S] (cons[T]}\ v_1\ v_2) \longrightarrow v_2 \quad \text{(E-TAILCONS)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{tail[T]}\ t_1 \longrightarrow \texttt{tail[T]}\ t_1'} \quad \text{(E-TAIL)}$$

*New typing rules* $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{nil}\ [T_1] : \texttt{List}\ T_1 \quad \text{(T-NIL)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : \texttt{List}\ T_1}{\Gamma \vdash \texttt{cons}[T_1]\ t_1\ t_2 : \texttt{List}\ T_1} \quad \text{(T-CONS)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{isnil}[T_{11}]\ t_1 : \texttt{Bool}} \quad \text{(T-ISNIL)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{head}[T_{11}]\ t_1 : T_{11}} \quad \text{(T-HEAD)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{tail}[T_{11}]\ t_1 : \texttt{List}\ T_{11}} \quad \text{(T-TAIL)}$$

**Figure 11-13: Lists**

Figure 16: List implementation scanned from the book [1]

Since there are no substitutions, there is actually no change due to the De Bruijn representation and functions like shift-L are just called recursively on subterms, see Appendix B.

But now, this definition is obviously not intend for proving type safety. Why? Because there is some problem that might occur during execution. What happens when we have term head or tail of a term nil? Well, no further step and it's not a value, so **Progress** can't be proved.
A solution would be, like it is done in all programming language, to use exceptions' managing: head A (nil A) reduces to some predefined exception error of the language and same goes for tail A (nil A). Then since an exception appears, all upper-terms will reduce to the exception's raise, i. e. given any constructor C of the type term, C (head A (nil A)) will reduce to C (raise e) and then to (raise e), with e some predefined exception. See chapter on Referencing and exceptions, for more intuition.

This conclude this part on new structures and the formalization of chapter 11. We will now make a break in the improvement of our language to talk about normalization.

## III.iii. Normalization

In this part we are going to reformulate the chapter 12 of the original book [1]. An important property of a programming language is that it should *ensure termination*. That's not completely true, since we can write silly well-typed programs, that never ends (loops). What is meant by **termination** is **convergence** of evaluation for the well-typed terms of our **call-by-value** language.
This convergence is defined as the existence of a so-called **normal form** (the state reached during evaluation where we cannot reduce a term any more).In the previous chapter $\lambda$-calculus, background and common notations, I talked at some point about the reason why we are using values (endpoint of computation). That's also the reason for this name **call-by-value**.

There are several ways to prove this fact, but Professor Pierce *tried* to explain some short and elegant technic, called **logical relation** [11] due to William Walker Tait in the 1967. This part will follow the sketch of Professor Pierce, but use a lot of lemmas not mentioned in the book [1]. We will suppose that the common facts like preservation or canonical forms are valid. Required sublemmas will always be precised. We will use the STLC language, without LIf and booleans. Bool type is replaced by some **generic type A**, that should contain at least one value (**valA**, in my formalization). Otherwise, since we have only variables, applications and abstractions, termination is obvious: beta reduction (function application) never happens (since there exists no value in our language), so there is no computation.

First, we need some easy vocabulary:

- **closed term** refers to a term with free variables (FV t= ∅)
- **multiple step computation** or star eval1_L, the **transitive reflexive closure** of eval1_L
  (computes in several steps(0 or more) to):
  star eval1_L t t (reflexivity) and if (eval1_L x y) and (star eval1_L y z) then star eval1_L x z
- **halts t** is a predicate stating that t computes until some endpoint (no further computation) or already doesn't compute:
  ∃ t'. star eval1_L t t' ∧ (∀ t1. ¬ eval1_L t' t1)

Our goal is to show: ∀ t T. $\vdash\ t\ \ |{:}|\ \ T \Leftarrow$ halts t. But we want to use relations to prove this fact.

In mathematics, relations are often characterized by sets. So, let start by defining our elements. We are reasoning on closed terms, since we can prove by induction on t that $\vdash\ t\ |{:}|\ T$ **is equivalent to** closed t.

We define then (C T), the set of closed terms of type T with empty context. We will start our reasoning on this set. Now, our goal is to show that the elements of (C T) halt for any arbitrary T (can only be a function type or A).

We define those elements as a set as well. This set is R T and it contains all elements of (C T) that halts. We can define R T as a set or a function, but it will not be possible in Isabelle to define it as a predicate, since this relation is not monotonic (P A $\leq$ P B, with $\leq$ a well-founded order).

The final recursive definition (functional definition giving back a boolean, closed to an inductive predicate) on the type T is :

$$\mathrm{t} \in_R \mathrm{A} = (\mathrm{t} \in \mathrm{C\ A} \wedge \mathrm{halts\ t})$$

$$\mathrm{t} \in_R (\mathrm{T1}\rightarrow\mathrm{T2}) = (\ \mathrm{t} \in \mathrm{C}\ (\mathrm{T1}\rightarrow\mathrm{T2}) \wedge \mathrm{halts\ t} \wedge (\forall\ \mathrm{s.}\ \ \mathrm{s} \in_R \mathrm{T1} \Rightarrow (\mathrm{LApp\ t\ s}) \in_R \mathrm{T2}))$$

Notice that we have a special rule for function types (reason why the predicate would not be monotonic).
Informally, the second equation means: "*t is in the realtion of (T1→T2) if and only if it is of type (T1→T2) with empty context and given any term s in the relation of T1, (LApp t s) is in the relation of T2*".

Why such a definition? Because we can only consider termination for a function if its arguments have to terminate. In the book [1], we can find a definition (page 150): "$\mathrm{R}_A$(t) $\Longleftrightarrow$ halts t". This is clearly not what we want, since it implies that, for example, (LAbs A valA) would fulfil $\mathrm{R}_A$, which is the same as saying t $\in_R$ A. But this means that (LAbs A valA) should have type A, contradiction. At the end, assuming such a definition is the same as assuming that we can prove falsity, which means that our proofs would be wrong.
Now, we have only the terms that interest us, i. e. t$\in_R$T. What we want is that those terms always terminate (halt predicate).

**Lemma** R_halts: $\forall$ t T. t$\in_R$ T $\Rightarrow$ halts t.

*Proof.*

Let t and T, be respectively a term and a type.

Let us proceed by induction on T. Since both equation of t$\in_R$ T contains halts t as clause and are CNF (conjunctive normal form). We get halts t for free.

*Qed.*

Since R is a function, we need some kind of **inversion lemma**
R_def: t$\in_R$T $\Rightarrow$ closed t $\wedge$ $\vdash$ $t$ |:| $T$ and R_A: t $\in_R$ A $\Rightarrow$ halts t.

With this lemmas, we can now start the main reasoning. We first know that a term t, such that t$\in_R$T, halts (but we need to prove it) and our final goal is to show that a well-typed term in empty context halts. So we need to show that well-typed terms t in empty concepts are such that t$\in_R$T.
Reasoning with empty context is not a good idea, because our hypothesis becomes too weak.

What happens then if we consider an arbitrary context? Let us go back to the definition of a context: a stack of pairs (variable's name, type of the variable), with De Bruijn representation we don't have names any more, just type correponding to an index but the idea is the same. So given a context $\Gamma$ and any set of values V, such that each values in V corresponds to the type at the same position in $\Gamma$, if we substitute all variables by the corresponding value in V, the obtained term is closed and well-typed in empty context.
Example: $\Gamma$ = [Nat, Bool] and V= [5, LTrue]

$\Gamma \vdash LIf\ (LVar 1)\ (LVar 0)\ 10$ |:| $Nat$

Now if we substitute V, $\vdash LIf\ (LTrue)\ 5\ 10$ |:| $Nat$.

From this observation, we notice that we can prove a more general lemma than the one using empty context, but we need to define first an operator for simultanous multiple substitution:

**Definition** subst-all(n:ℕ)(V: list of terms)(t:term).

Equations defining the function:

**Recursion on t**,
"subst-all n V ValA = ValA"
"subst-all n V (LApp s t) = LApp (subst-all n V s) (subst-all n V t)"
"subst-all n V (LAbs T t) = LAbs T (subst-all (Suc n) (map (shift-L 1 0) V) t)"
"subst-all n V (LVar k) = (if (k≥n ∧ (k-n)<length V) then (V!(k-n)) else LVar k)"

Since Isabelle is defined in a classical setting, excluded middle is assumed, which allows such definiton (condition on proposition). For instance, since order on natural numbers is decidable, this function can also be defined in a constructive setting without excluded middle.
The common usage of this function is namely subst-all 0, since we want to substitute the elements that have the same index as those in V. To complete this definition, we need to prove two extra facts, namely the simplification when V is empty and the simplification of composition with single substitution subst-L.

**Lemma** subst_all_empty: ∀ n t. subst-all n [] t = t.

*Proof.*

Let t and n be respectively a term and a natural number.
Obvious by induction on t, since in the case LVar, no natural number is less than 0 (reason for this extra condition (staying within the list V)).

*Qed.*

The lemma subst_comp_subst_all:

$$\forall\ \text{n V t. subst-L n v (subst-all (Suc n) (map (shift-L (n + 1) 0) V) t)} = \\ \text{subst-all n (v\#map (shift-L (n + 1) 0) V) t}$$

can be proved using the fact that: ∀ x t t1. x∉FV t ⇒ subst-L x t1 t = t.

Then, we can state our lemma subst_R:

$$\forall\,\Gamma\ \text{t V T.}\ \Gamma\ \vdash\ t\ \ |{:}|\ \ T \Rightarrow \text{length V} = \text{length}\ \Gamma\ \ \Rightarrow \\ (\forall\ \text{i. i<length V} \Rightarrow \text{is\_value\_L (V!i)} \wedge \text{(V!i)} \in_R (\Gamma\text{!i})) \Rightarrow \\ \text{subst-all 0 V t} \in_R \text{T}$$

Since we are once more dealing with substitution and typing, there is no real surprise. We need a lemma like Substitution but for subst-all, substitution_all (be careful @ stands for list concatenation in this lemma and not replacement of set theory):

$$\forall\,\Gamma\ \text{t k V T.}\ \Gamma\ \vdash\ t\ \ |{:}|\ \ T \Rightarrow \text{k+length V} = \text{length}\ \Gamma\ \ \Rightarrow \\ (\forall\text{i. i<length V} \Rightarrow\ \vdash\ V!i\ \ |{:}|\ \ \Gamma!(i+k)) \Rightarrow \\ take\ k\ \Gamma\ @\ drop\ (k + length\ V)\ \Gamma\ \ \vdash \text{subst-all k V t}\ |{:}|\ \ T$$

The only difficulty encountered in the proof of this fact, done by induction on the typing predicate, is that we need to show at some point ⊢ shift-L 1 0 (V!i) |:| $\Gamma!(i + k)$ and we don't have any lemma or information allowing us to prove it. Now, remember that we that (V!i) $\in_R$ (Γ!i), so we know that ⊢ $V!i$ |:| $\Gamma!i$. This will be enough, since we can correlate free variables and context length, thanks to indexation.

In fact, just prove: ∀ Γ t T. Γ ⊢ $t$ |:| $T$ ⇒ FV t ⊆ {k. k<length Γ}. Now since we have some term well-typed in empty context, it means that it doesn't contains any free variable. This can be proved by induction on the typing predicate.

Well, everything seems fine, but there also other lemmas required to prove subst_R, namely in the only *long* case (LAbs), we will need to know that multiple step computation (star eval1_L) preserves the fact of being in a relation ($\in_R$ T, for some type T). To prove these facts, we will need the previous lemma about FV and context, but also preservation and others (see file Normalization.thy for more details).

Finally, the proof of Normalization is then really easy:

**Lemma** Normalization: ∀ t T. $\emptyset \vdash t$ |:| $T \Rightarrow$ halts t.

*Proof.*

Let t and T, be respectively a term and a type.
Assume $\emptyset \vdash t$ |:| $T$.

By applying the lemma subst_R to the assumption, we get that $t \in_R T$ and so we can conclude by applying R_halts.

*Qed.*

This conclude this part on Normalization, next chapter in the book [1] and in this report is about **referencing** and **exceptions**, which will conclude my work.

## III.iv. Referencing and exceptions

This section will corresponds to chapter 13 and 14 in the book [1]. In all generic programming languages, **references (pointers)** are critical elements, since they are deeply related to the **memory model of our language** as well. **The memory model** is also a formalism that aims at reproducing what really happens in term of allocation and deallocation. This part will not focus on it, since our goal is only to show **Progress** and **Preservation**, while memory model would bring up problems like memory corruption (writting on already allocated space). Therefore, we will suppose that we have a perfect model (infinite memory). The memory will be represented as a **partial function** that gives back a term for each **allocated *address***. Since we are on a computer, once more using function is probably too large. Since we only care about **the domain of the function** and **its values**, we will replace the function by a list. Since we use infinite memory, the memory addresses' representation should be infinite as well. Therefore, we will use natural number as **memory adress** or more specifically **location**. All the theoretical results of this part follows the book [1], but the representation with a list, is inspired from the De Bruijn representation. It is already mentioned in the book [1] that garbage collection can produce problem and since it is only quoted in the book [1], it will not be formalized in this thesis.

Furthermore, for convenience, we will restrict our study to a language containing only: **unit, variables, applications, abstractions (functions)** and the memory related features: **referencing (ref t), dereferencing (!t) and assignment (t1::=t2)**.

Since we can allocate memory, we need to keep track of the memory change during evaluation. So the current statement "eval1_L t t1" becomes "eval1_L t $\mu$ t1 $\mu$1", with $\mu$ a list of terms. It is not strange, that both sides don't have the same list, because an allocation will modify it. We will keep Pierce's notation for allocation in the memory "$\mu \leftarrow t$" but it will correspond to inserting t at the end of the list $\mu$.
In the same way, **typing** will also differ since now we need to track the types of elements in memory. For this purpose, we add another stack $\Sigma$ to the typing: $\Gamma|;|\Sigma \vdash t \ |:| \ T$. I define the same kind of membership operator as Martin ($|\in|$) for this new context, using notation $|\in_l|$.

With all this element, we will then define formally our language (like in the previous part, the STLC part and unit part (variables, abstraction, applications and unit) of the rules are omitted:

- Types: Ref T with arbitrary type T
- Terms: ref t, with t an arbitrary term
  !t for dereferencing, with an arbitrary term
  t1::=t2 for assignment, with t1 and t2 arbitrary terms
  L n for location, with n is an arbitrary index($\mathbb{N}$)
- De Bruijn representation:

  **shift-L**: shift-L d c (ref t) = ref (shift-L d c t)
  shift-L d c (!t) = !(shift-L d c t)
  shift-L d c (t1::=t2) = (shift-L d c t1)::=(shift-L d c t2)
  shift-L d c (L n) = L n

  **subst-L**: subst-L j s (ref t) = ref (subst-L j s t)
  subst-L j s (!t) = !(subst-L j s t)
  subst-L j s (t1::=t2) = (subst-L j s t1)::=(subst-L j s t2)
  subst-L j s (L n) = L n

  **FV**: FV (ref t) = FV t
  FV (!t) = FV t
  FV (t1::=t2) = (FV t1) $\cup$(FV t2)
  FV (L n) = $\emptyset$

$$\text{Val\_Loc}\ \frac{}{\text{is\_value\_L (L n)}} \qquad \text{Ref}\ \frac{\text{eval1\_L t1 }\mu\text{ t1' }\mu 1}{\text{eval1\_L (ref t1) }\mu\text{ (ref t1') }\mu 1}$$

$$\text{RefV}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (ref v) }\mu\text{ (L (length }\mu\text{)) }\mu 1} \qquad \text{Deref}\ \frac{\text{eval1\_L t1 }\mu\text{ t1' }\mu 1}{\text{eval1\_L (!t1) }\mu\text{ (!t1') }\mu 1}$$

$$\text{DerefV}\ \frac{\text{n<length }\mu}{\text{eval1\_L (!(L n)) }\mu\text{ (}\mu\text{!n) }\mu} \qquad \text{Assign1}\ \frac{\text{eval1\_L t1 }\mu\text{ t1' }\mu 1}{\text{eval1\_L (t1::=t2) }\mu\text{ (t1'::=t2) }\mu 1}$$

$$\text{Assign2}\ \frac{\text{is\_value\_L v} \qquad \text{eval1\_L t2 }\mu\text{ t2' }\mu 1}{\text{eval1\_L (v::=t2) }\mu\text{ (v::=t2') }\mu 1} \qquad \text{Assign1}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (L n::=v) }\mu\text{ (unit) }\mu[n := v]}$$

Figure 17: Evaluation rules and value for references

Please be careful !t is dereferencing, while $\mu$!n is taking the nth value of the list $\mu$. Furthermore, the notation $\mu[n := v]$ means replacing the nth element by the value v.

$$\text{Loc}\ \frac{(\text{n,T1})\ |\in_l|\ \Sigma}{\Gamma|;|\Sigma\ \vdash\ Ln\ \ |{:}|\ \ Ref\ T1} \qquad \text{Deref}\ \frac{\Gamma|;|\Sigma\ \vdash\ t\ \ |{:}|\ \ Ref\ T1}{\Gamma|;|\Sigma\ \vdash\ !t\ \ |{:}|\ \ T1}$$

$$\text{Ref}\ \frac{\Gamma|;|\Sigma\ \vdash\ t\ \ |{:}|\ \ T1}{\Gamma|;|\Sigma\ \vdash\ ref\,t1\ \ |{:}|\ \ Ref\ T1} \qquad \text{Assign}\ \frac{\Gamma|;|\Sigma\ \vdash\ t1\ \ |{:}|\ \ Ref\ T1 \qquad \Gamma|;|\Sigma\ \vdash\ t2\ \ |{:}|\ \ T1}{\Gamma|;|\Sigma\ \vdash\ t1\ ::=t2\ \ |{:}|\ \ Unit}$$

Figure 18: Typing for references

Since locations are values, we have a **new canonical form**:

$$\text{is\_value\_L v} \Rightarrow \Gamma|;|\Sigma\ \vdash\ v\ \ |{:}|\ \ RefT1 \Rightarrow (\exists\ \text{n. n<length}\ \Sigma \wedge \text{v} = \text{L n} \wedge \Sigma!\text{n=T1})$$

Notice that the context for memory is never modified, namely since it is specified directly and then combined with the following proposition, named welltyped_store:

$$\Gamma\ |;|\ \Sigma \vDash\ \mu \triangleq (\text{length}\ \Sigma\ = \text{length}\ \mu) \wedge (\forall\ \text{i<length}\ \Sigma.\ \Gamma|;|\Sigma\ \vdash\ \mu!i\ \ |{:}|\ \ \Sigma!i)$$

Since we have now to consider the state of the memory, our main theorems (**Progress and Preservation** are going to change:

**Progress**: $\forall\,\Gamma\,\Sigma\,\text{t T1}.\ |;|\Sigma\ \vdash\ v\ \ |{:}|\ \ T1 \Rightarrow \text{is\_value\_L t} \vee$
$(\forall\ \mu.\ \Gamma\ |;|\ \Sigma \vDash\ \mu\ \Rightarrow (\exists\ \text{t'}\ \mu 1.\ \text{eval1\_L t}\ \mu\ \text{t'}\ \mu 1))$

**Preservation**: $\forall\,\Gamma\,\Sigma\,\mu\,\mu 1\ \text{t t' T}.\ \Gamma|;|\Sigma\ \vdash\ t\ \ |{:}|\ \ T \Rightarrow\ \Gamma\ |;|\ \Sigma \vDash\ \mu\ \Rightarrow$
$\text{eval1\_L t}\ \mu\ \text{t'}\ \mu 1\ \ \Rightarrow \exists\,\Sigma 1.\ \Gamma\ |;|\ \Sigma@\Sigma 1 \vDash\ \mu 1 \wedge \Gamma|;|\Sigma@\Sigma 1\ \vdash\ t'\ \ |{:}|\ \ T$

Now, we will need some lemmas involving welltyped_store:

- **store_updt:**

  $\forall\,\Gamma\,\Sigma\,\mu\ \text{i t v T}.\ \Gamma\ |;|\ \Sigma \vDash\ \mu \Rightarrow \text{i<length}\ \Sigma \Rightarrow \Sigma!\text{i} = \text{T} \Rightarrow$
  $\Gamma|;|\Sigma\ \vdash\ v\ \ |{:}|\ \ T \Rightarrow \Gamma\ |;|\ \Sigma \vDash\ \mu[i := v]$

- **store_weakening:**

  $\forall\,\Gamma\,\Sigma\,\Sigma 1\ \text{t T}.\ \Gamma|;|\Sigma\ \vdash\ t\ \ |{:}|\ \ T \Rightarrow \Gamma|;|\Sigma@\Sigma 1\ \vdash\ t\ \ |{:}|\ \ T$

store_weakening can be prove by induction on the typing predicate and store_updt requires some simple lemma about replacement(in lists).
All proofs are pretty much straightforward, so please refer to the book [1] and the file (Referencing;thy). For instance, **Weakening** and **Substitution** are the same with the extra context($\Sigma$) in the typing predicate.

This concludes the referencing part of this chapter. We will now look quickly at exceptions and error managing.

The book [1] presents two kinds of error managing, once more in a restricted language (STLC without Bool). The first one is a language with an instruction error. This language is well-suited to define and prove that Lists can be added to our language, without breaking Progress and Preservation. Therefore, by adding the following evaluation rules and the rules presented by Professor Pierce in the book, we can still prove type safety:

$$\text{Ltail\_nil}\ \frac{}{\text{eval1\_L (Ltail A (Lnil B)) error}}$$

$$\text{Lhead\_nil}\ \frac{}{\text{eval1\_L (Lhead A (Lnil B)) error}}$$

Figure 19: Additional evaluation rules for lists

The evaluation rules are trivial, namely error stops the evaluation, i. e. each time it appears under a constructor, the term reduces to error. This behaviour can be generalized to a language with more instructions and abide programming rules. When it comes to typing, having an error in our program doesn't mean that it is not well-typed. Since we want to be able later to use this information (there was an error somewhere), error should be able to take anytime (rule proposed by Professor Pierce).

The **Progress** and **Preservation** for this language is easy, but **Progress** means that "*either t is a value or it evaluates or it is an error*". Well now we have an error, but we cannot do anything with it, so let us formalize the **try and catch** structure (here try t1 with t2).

But now, like presented in the book [1], we just end up doing copy paste and adding the following rules for an **exception type Texn**, which must contain at least one value. I will omit rules for the De Bruijn representation, because the function are only applied recursively for shifting and substitution and we take the union of the recursive calls of FV for the free variables.

The formal rules and the terms' definition are the following (given a fixed Texn, with at least one value):

Terms: raise t, raises an exception, with t an arbitrary term
try t1 with t2, with t1 and t2 arbitrary terms

$$\text{AppRaise1}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (LApp (raise v) t2) (raise v)}} \qquad \text{AppRaise2}\ \frac{\text{is\_value\_L v} \qquad \text{is\_value\_L v1}}{\text{eval1\_L (LApp v (raise v1)) (raise v1)}}$$

$$\text{Raise}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (raise t1) (raise t1')}} \qquad \text{RaiseRaise}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (raise (raise v)) (raise v)}}$$

$$\text{tryV}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (try v with t2) v}} \qquad \text{tryE}\ \frac{\text{eval1\_L t1 t1'}}{\text{eval1\_L (try t1 with t2) (try t1' with t2)}}$$

$$\text{TryRaise}\ \frac{\text{is\_value\_L v}}{\text{eval1\_L (try (raise v) with t2) (shift-L (-1) 0 (subst-L 0 (shift-L 1 0 v) t2))}}$$

Figure 20: Evaluation rules and value for try and raise

$$\text{Raise}\ \frac{\Gamma \vdash t \ |:|\ Texn}{\Gamma \vdash raise\, t \ |:|\ T1}$$

$$\text{Try}\ \frac{\Gamma \vdash t1 \ |:|\ T1 \qquad \Gamma \vdash t2 \ |:|\ Texn \to T1}{\Gamma \vdash try\, t1\, with\, t2 \ |:|\ T1}$$

Figure 21: Typing for try and raise

If Texn fulfils the requirements: having one value at least and well-typed expressions fulfilling **Progress**, then we can prove Progress and Preservation for such a language. Since the exercise proposed in the book is a generalization of this language, I formalized a slightly stricter version.

The exercise consists in proving type safety for a Java-like setting (exceptions are variant types and LAbs takes an additional variant type as argument). I didn't manage to finish this exercise, since I restricted the try catch behavior and the LAbs definition. Indeed, my language don't allow functions with a set of raisable errors E to call other functions with a set of raisable errors E1, if E≠E1. Furthermore, **try** is considered to capture only errors of specified type.
Example: LAbs T < [Err],[Nat>] (try (LAbs T1 < [Err0],[Nat>] ...)) with ... is not allowed.
The reason is easy, in the definition above, **Texn** is a static type, that the user can't define. In my language, the only difference is that it can change for each top lambdas. Let us have a look how to formalize the rule, then we will discuss the difficulty of trying to be larger with the definition (try can catch any errors, while the with part raises only allowed errors and takes arguments with the same type as raised errors).

We know that we need somehow to retrieve the type of allowed exceptions with the typing predicate, in the LAbs case. So the conventional idea is to add a *new context*. Since we are only caring about one type of exceptions (variant type), there is no need to use a list. We can stick with a single variant type, i. e. a pair L TL of lists of strings and types, respectively. The rule for typing LAbs looks then like:

$$\forall\,\Gamma\ \mathrm{t\ L\ TL\ A\ B}.\ < L, TL > |*| A\#\Gamma\ \vdash\ t\ \ |:|\ \ B \Rightarrow \\ < [], [] > |*| \Gamma\ \vdash\ LAbs\ A\ \ < L, TL >\ t\ \ |:|\ \ A \to B$$

**Problem:** In Java, void A(int a, int b) throws ArithmeticException is a correct syntax. But in lambda-calculus, we have a binder for each argument: LAbs Nat <L1,TL1> (LAbs Nat <L2,TL2> (body of the function)). We cannot force in the term construction (L1=L2 and TL1=TL2), which is what I avoided with my restriction.

There is a solution to solve this problem:

- **subtyping:**

  If our type system supports subtyping, then we can define a rule with <L2,TL2>⊑<L1,TL1> as premise, where ⊑ is the types' ordering relation.
  The rule will looks like:

$$\forall\,\Gamma\ \mathrm{t\ L\ TL\ A\ B}.\ < L1, TL1 > |*| B\#A\#\Gamma\ \vdash\ t\ \ |:|\ \ B \Rightarrow \text{<L,TL>} \sqsubseteq \text{<L1,TL1>} \Rightarrow \\ < L1, TL1 > |*| A\#\Gamma\ \vdash\ LAbs\ A\ \ < L, TL >\ t\ \ |:|\ \ A \to B$$

  But **this solution is time-expensive and requires a lot of work** (introducing subtyping and checking correctness of the system).

In conclusion, my restricted version is **type-safe**, but pretty **restrictive**. But it requires a lot of extra definitions and lemmas, if you want to formalize a nice version with subtyping and probably other features. The main difficulty is probably designing the typing rule of the try with structure. To be completely rigorous, we know that t2 is only allowed to raise exceptions that have the type specified in the exception context. But it could take **any arbitrary exception's type(variant type)** as argument(like mentioned above), and t1 could raise any type of exception if it corresponds to the type of the argument of t2.
Furthermore, t1 is allowed to be LAbs or some code converging to a raise instruction of arbitrary type. This allows differences between the exception context for typing t1 and type of argument of t2.

Formally, we would like to write:

$$\begin{aligned}\forall\,\Gamma\ \text{t1 t2 L TL L' TL' A.} &< L', TL' > |*| \Gamma \vdash t1\ |:|\ A \Rightarrow \\ &< L, TL > |*| \Gamma \vdash t2\ |:| < L', TL' > \rightarrow A \Rightarrow \\ &< L, TL > |*| \Gamma \vdash try\, t1\ with\, t2\ |:|\ A\end{aligned}$$

But that's clearly not working this way, there are some preconditions missing. This concludes the report of my work, I will give some further conclusion remark after talking about related works and relevance of this subject.

# Related work

After explaining my own work, I will now explain the pertinence of such thesis by describing related works. Namely, just formalizing a book [1] in itself is not the endpoint.

The first related work is the POPLmark challenge [12], proposed in 2005. The purpose of this challenge was to have a look at formalization of concrete programming languages (challenges) and try to prove them on a computer. The real aim is obviously to see what kind of formalisms is used and if we can managed depending on the tools (proof assistants, provers, ...) to give a solution, that any user (who doesn't even know about automation of proof checking) would be able to find with a small introduction to tool.
Well, that's probably not realistic, but the questions were why and what has to be done, so that it becomes possible.

This is an example of what has been done before, but the idea of having a tools fitting for any user still remains. Yearly, another conference is given on subjects related to improvement of interactive theorem provers. Called ITP, this conference will gather researchers around the world to discuss of the current results and improvement of tools. But researchers are not the only one, who contributes to improvement.

It is true, that development of the tool itself and new mechanisms is pretty much the privilege of researchers. But what will always be needed is background-theories. While lambda-calculus is then a good example of formalized background theory, the formalizations are restricted and most of the time belongs only to one proof assistant library. Furthermore, commonities develop a lot of theories, that before remains in the dark. Nowadays, AFP or Coq website allow to check theories and publish them, so that anyone can profit from the background-theories of others. This contributes to avoid unnecessary and/or tedious work and to the development of the proof assistant.
Why? Because Coq proposed since a long time and Isabelle since 2015, a language for tactics allowing to develop methods for other users. Given those specific automations, some common user may be able to prove *difficult* theorems easily. The difficulties, that is mentioned here, is unfortunately not a theoretical one. It's the difficulty coming from lack of experience with the tools.
A lot of mathematicians feel hesitant about trying to do proofs on computer since it requires to learn a new language, while they are well-versed in the mathematical one.

Even through Isabelle proposed a syntax closed to the mathematical language, usage of automation imposes to know the language of tactics. For those people, having nice automatic tactics, like inversion on inductive definition, like tower reasoning for ZF set theory and others, is a blessing. They know what kind of proof is being done, and the computer does it without showing it or asking for some specific tactics or instruction.

To go back on the main topic: programming languages' formalization, there are new language with powerful feature that would require some background or generic methods to be formalized easily. For example, there is a project: "RustBelt: Logical Foundations for the Future of Safe Systems Programming" [13] on formalization of the language Rust in Coq, started in 2015 under the direction of Derek Dreyer. But there is also what we could call the ghost of the past that also emerges like unsoundness of java and scala's type system [14]. Furthermore, some groups in Paris works on formalization of data model and associated languages (SQL and IEEE) in Coq. They published in particular a paper about javascript specification [15]. Finally, Mezzo has also a partial formalization in Coq (see the paper [16]).

To conclude, formalization of programming languages and tools to make this task easier are the main subjects of a lot of workshops like:

- CoqPL on a yearly base (last on the 21th January 2017)
- POPL, which was quoted as first example, still exists (last on January 2017)
- PiP, Principles in practice, which is about applying current formalisms to real-world languages for testing, analysis and verification
- N40AI, Next 40 years of Abstract Interpretation is about impact of Abstract Interpretation during the last 40 years and new challenges involving it
- OBT, Off the Beaten Track is about unfamiliar problems in conferences but that shows up a lot in practice

Obviously, this list is not exhaustive, but this just show that the subject is monochrome. It is subdivided first between famous problems and practical problems, then into semantic models' impact and automation of tasks(here is meant *proving soundness of a language*, *checking correctness of a program* and *investigating behaviour of a program*). New languages don't appear so frequently, but research in the field is what makes them appear, since investigation and especially formalization put us in front of theoretical problems and solutions that may have different and really challenging applications.

# Conclusion

This master thesis allowed me to deepen my understanding of programming languages and the proof assistant Isabelle/HOL. Now, I know a lot of useful technics and touch a bit to the backstage of the application (ML programming), even through it was not mentioned in this report (no concrete result).
Furthermore, being confronted with all kind of proofs and reasonings helped me to mature as a person. Indeed the difficult parts of this formalization allows us to realize the limitations of our tools(proof assistants), but also the difficulty of the task of designing such languages consistently. The **De Bruijn representation** is obviously some nightmare, you can't escape from, and using **nominals** would probably have pushed difficulties on a different side (reason why it wasn't used in this work). Therefore, I suggest greatly to try it out and verify my words. There is still a lot of aspects of programming languages described in the book written by Professor Pierce and probably some imprecisions and misleading parts (normalization part), that will show up during formalization. But this remains eventual future work for my successors, together with the incomplete parts(general Java-like exceptions and general pattern matching). Finally, what will remain from my work is a formalization of a language with a lot of interesting structures (exceptions, disjoint sum, ...) and some Eisbach methods that can be reused easily.

# **Appendix A:** Extra list functions and lemmas

This theory contains all references to list iterators and extra functions. Please see original file List_extra.thy for not displayed proofs.

**fun** *list-iter* $::('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ list \Rightarrow 'b$ **where**
*list-iter f r* $[] = r$ |
*list-iter f r* $(a\#xs) = f\ a$ (*list-iter f r xs*)

**fun** *replace* $::nat \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
*replace n x xs* =
  (*if length xs* $\leq$ *n then xs*
    *else* (*take n xs*) @ [*x*] @ (*drop* (*Suc n*) *xs*))

**abbreviation** *fst-extract*$::('a \times 'b)\ list \Rightarrow 'a\ list$ **where**
*fst-extract L* $\equiv$ (*list-iter* ($\lambda p\ r.\ fst\ p\ \#\ r$) [] *L*)

**abbreviation** *snd-extract*$::('a \times 'b)\ list \Rightarrow 'b\ list$ **where**
*snd-extract L* $\equiv$ (*list-iter* ($\lambda p\ r.\ snd\ p\ \#\ r$) [] *L*)

**abbreviation** *update-snd*$::('b \Rightarrow 'c) \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'c)\ list$ **where**
*update-snd f L* $\equiv$ *zip* (*fst-extract L*) (*map f* (*snd-extract L*))

**fun** *BigCirc*$::('a \Rightarrow 'a)\ list \Rightarrow ('a \Rightarrow 'a)$ ($\bigodot$ (-) [75] 55) **where**
$\bigodot\ [] = id$ |
$\bigodot\ (f\#fs) = f \circ (\bigodot\ fs)$

**fun** *BigCircT*$::('a \rightharpoonup 'b)\ list \Rightarrow ('a \rightharpoonup 'b)$ ($\bigodot_T$ (-) [75] 55) **where**
$\bigodot_T\ [] = empty$ |
$\bigodot_T\ (f\#fs) = f\ {+}{+}\ (\bigodot_T\ fs)$

**lemma** *replace-inv-length*[*simp*]:
  *length* (*replace n x S*) = *length S*
**by**(*induction S arbitrary*: *x n*, *auto*)

**lemma** *nth-replace*[*simp*]:
  $i{<}length\ L \Longrightarrow (replace\ n\ x\ L)!i = (if\ i{=}n \wedge n{<}length\ L\ then\ x\ else\ (L!i))$

**lemma** *insert-nth-comp*:
  $n \leq$ *length* $L \Longrightarrow n \leq n1 \Longrightarrow$ *insert-nth n S* (*insert-nth n1 S1 L*) = *insert-nth* (*Suc n1*) *S1* (*insert-nth n S L*)
  $n \leq$ *length* $L \Longrightarrow n \geq n1 \Longrightarrow$ *insert-nth* (*Suc n*) *S* (*insert-nth n1 S1 L*) = *insert-nth* (*n1*) *S1* (*insert-nth n S L*)

**lemma** *rep-ins*:
  $n \leq n1 \Longrightarrow n \leq$ *length* $W \Longrightarrow$ *insert-nth n S* (*replace n1 A W*) = *replace* (*Suc n1*) *A* (*insert-nth n S W*) (**is** $?P \Longrightarrow\ ?R \Longrightarrow\ ?Q$)
**proof** –
  **assume** *H*: *?R ?P*
  **have** *1*:$n1 \geq$ *length* $W \Longrightarrow\ ?Q$
    **by** (*simp add*: *min-def*)
  **have** $n1 <$ *length* $W \Longrightarrow\ ?Q$
    **proof** –
      **assume** *H1*: $n1 <$ *length W*
      **have** (*Suc* (*Suc n1*) − *n*) = *Suc* (*Suc n1* − *n*)
          **using** *H*
          **by** *fastforce*
      **with** *H* **show** *?thesis*
        **by** (*simp add*: *H1 min-def*)(*simp add*: *H Suc-diff-le take-drop*)
    **qed**
  **with** *1* **show** *?Q* **by** *linarith*
**qed**

**lemma** *rep-ins2*:
  $n < n1 \Longrightarrow n1 \leq$ *length* $W \Longrightarrow$ *insert-nth n1 S* (*replace n A W*) = *replace n A* (*insert-nth n1 S W*) (**is** $?P \Longrightarrow\ ?R \Longrightarrow\ ?Q$)
**proof** –
  **assume** *H*: *?R ?P*
  **have** *Suc n* $\leq$ *n1*
      **by** (*metis* (*no-types*) *H*(*2*) *Suc-leI*)
  **with** *H* **show** *?Q*
    **by** (*simp add*: *drop-Cons′ drop-take take-Cons′ min-def*)
**qed**

**lemma** *len-fst-extract*[*simp*]:
  *length* (*fst-extract L*) = *length L*
**by** (*induction L*, *auto*)

**lemma** *len-snd-extract*[*simp*]:
  *length* (*snd-extract L*) = *length L*
**by** (*induction L*, *auto*)

```
lemma fst-extract-zip1:
  length L = length L1 ⟹ fst-extract (zip L L1) = L
proof(induction L arbitrary: L1)
  case (Cons a L')
    obtain b L1' where L1 = b#L1'
      using Cons(2)
      by (metis length-Suc-conv)

    with Cons
      show ?case
        using fst-conv length-Cons list-iter.simps(2) nat.simps(1) zip-Cons-Cons
          by auto
qed auto

lemma zip-comp:
  L = zip (fst-extract L) (snd-extract L)
by(induction L,auto)

lemma inset-find-Some:
  x ∈ set (fst-extract L) ⟹ ∃ p. find (λe. fst e = x) L = Some (x,p) ∧ (x,p) ∈
set L
by (induction L arbitrary: x, auto)

lemma incl-fst:
  set L ⊆ set L1 ⟹ set (fst-extract L) ⊆ set (fst-extract L1)
using zip-comp[of L1]
      set-zip-leftD[of - - fst-extract L1 snd-extract L1]
by (induction L arbitrary: L1, auto)

lemma incl-snd:
  set L ⊆ set L1 ⟹ set (snd-extract L) ⊆ set (snd-extract L1)
using zip-comp[of L1]
      set-zip-rightD[of - - fst-extract L1 snd-extract L1]
by (induction L arbitrary: L1, auto)

lemma find-zip1:
  distinct L ⟹ length L = length L1 ⟹ length (L@L3) = length L2 ⟹ j<
length L ⟹ find (λp. fst p = k) (zip L L1) = Some ((zip L L1) ! j)
   ⟹ find (λp. fst p = k) (zip (L@L3) L2) = Some ((zip (L@L3) L2) ! j)
```

```
lemma list-iter-nil:
list-iter op @ [] L = [] ⟹ i<length L ⟹ L!i = []
proof (induction L arbitrary: i)
  case (Cons a L')
    show ?case
      proof (cases i>0)
        case True
          from this obtain j where i = Suc j
            by (metis Suc-pred)
          thus ?thesis
            using Cons(1)[of j]
                  Cons(2,3)
            by force
      next
        case False
          thus ?thesis
            using Cons(2)
            by auto
      qed
qed auto

lemma list-map-incl:
  set (list-iter op @ [] (map f L)) ⊆ S ⟹  i<length L ⟹ set(f (L!i)) ⊆ S
proof (induction L arbitrary: f S i)
  case (Cons a L')
    from Cons show ?case
      by (induction i arbitrary: L', auto)
qed auto

lemma list-map-incl2:
  (⋀i. i<length L ⟹ set(f (L!i)) ⊆ S) ⟹ set (list-iter op @ [] (map f L)) ⊆ S
proof (induction map f L arbitrary: L f S)
  case (Cons a L')
    obtain b L1 where L = b#L1
      using Cons(2)
      by blast
    thus ?case
      using Cons(1)[of f L1]
            Cons(3)[of Suc -]
            Cons(3)[of 0]
            Cons(2)
      by auto
qed auto

lemma fst-extract-app[simp]:
  fst-extract (L@L1) = fst-extract L @ fst-extract L1
by (induction L arbitrary: L1, auto)
```

**lemma** *snd-extract-app*[*simp*]:
  *snd-extract* (*L*@*L1*) = *snd-extract L* @ *snd-extract L1*
**by** (*induction L arbitrary*: *L1*, *auto*)

**lemma** *fst-extract-fst-index*[*simp*]:
  *i*< *length L* $\Longrightarrow$ *fst-extract L* ! *i* = *fst* (*L* ! *i*)
**proof** (*induction L arbitrary*: *i*)
  **case** (*Cons a L'*)
    **thus** *?case*
      **by** (*induction i arbitrary*: *L'*, *auto*)
**qed** *auto*

**lemma** *snd-extract-snd-index*[*simp*]:
  *i*< *length L* $\Longrightarrow$ *snd-extract L* ! *i* = *snd* (*L* ! *i*)
**proof** (*induction L arbitrary*: *i*)
  **case** (*Cons a L'*)
    **thus** *?case*
      **by** (*induction i arbitrary*: *L'*, *auto*)
**qed** *auto*

**lemma** *fst-updt-snd-is-fst*[*simp*]:
  *fst-extract* (*update-snd f L*) = *fst-extract L*
**by** (*induction L arbitrary*:*f*, *auto*)

**lemma** *fst-ext-com-list-it-app*:
  *fst-extract* (*list-iter op* @ [] *L*) = *list-iter op* @ [] (*map fst-extract L*)
**by**(*induction L*, *auto*)

**lemma** *snd-ext-com-list-it-app*:
  *snd-extract* (*list-iter op* @ [] *L*) = *list-iter op* @ [] (*map snd-extract L*)
**by**(*induction L*, *auto*)

**lemma** *map-com-list-it-app*:
  *map F* (*list-iter op* @ [] *L*) = *list-iter op* @ [] (*map* (*map F*) *L*)
**by** (*induction L arbitrary*: *F*, *auto*)

**lemma** *zip-com-list-it-app*:
  ($\bigwedge$*L1*. *length* (*f L1*) = *length* (*g L1*)) $\Longrightarrow$ *zip* (*list-iter op* @ [] (*map f L*)) (*list-iter op* @ [] (*map g L*)) =
    *list-iter op* @ [] (*map* ($\lambda$*p*. *zip* (*f p*) (*g p*)) *L*)
**by** (*induction L arbitrary*: *f g*, *auto*)

**lemma** *list-it-map-app-map*:
  *list-iter op* @ [] (*map* (*update-snd F*) *L*) = *update-snd F* (*list-iter op* @ [] *L*)
**using** *fst-ext-com-list-it-app*[*of L*] *snd-ext-com-list-it-app*[*of L*]
    *map-com-list-it-app*[*of F map snd-extract L*]
    *zip-com-list-it-app*[*of fst-extract map F* $\circ$ *snd-extract L*]
**by** *force*

```
lemma update-snd-subset:
  set L ⊆ set L1 ⟹ set (update-snd F L) ⊆ set (update-snd F L1)
proof (induction L arbitrary: L1 F)
  case (Cons a L′)
    obtain j where H:j<length L1 L1 ! j = a
      using Cons(2)
            set-conv-nth[of L1]
      by auto
    have (fst a, F (snd a)) ∈ set (update-snd F L1)
      using H in-set-zip
      by force
    then show ?case
       using Cons
        by simp
qed auto

lemma set-foldl-app[simp]:
  set(foldl op @ L1 L) = (UN l : set L. set l) ∪ set L1
by (induction L arbitrary: L1, auto)

lemma set-foldl-union[simp]:
  foldl op ∪ S L = (UN l : set L. l) ∪ S
by (induction L arbitrary: S, auto)

lemma update-snd-rewrite-fun:
  (∀ i<length L. f (snd (L!i)) = g (snd (L!i))) ⟹ update-snd f L = update-snd g
L
by (induction L, force+)

lemma update-snd-comp:
  (update-snd F ∘ update-snd G) L = update-snd (F ∘ G) L
by (induction L, force+)

lemma fst-map:
  fst-extract L = map fst L
by (induction L, auto)

lemma snd-map:
  snd-extract L = map snd L
by (induction L, auto)

lemma count-rem1:
  x ∈ set L ⟹ count-list L x = count-list (remove1 x L) x + 1
by(induction L arbitrary: x, auto)
```

**lemma** *count-Suc*:
  *count-list L x = Suc n* $\Longrightarrow$ *x* $\in$ *set L*
**proof**(*induction L arbitrary*: *n x*)
  **case** (*Cons a L1*)
    **have** *x* $\notin$ *set L1* $\Longrightarrow$ *x = a*
      **using** *count-notin*[*of x L1*]
            *Cons*(*2*)
      **by** (*cases a=x*,*auto*)
    **then show** *?case*
      **by** (*cases x* $\in$ *set L1*, *auto*)
**qed** *auto*

**lemma** *count-inset*:
   *x* $\in$ *set L* $\Longrightarrow$ $\exists$ *n*. *count-list L x = Suc n*
**proof**(*induction L arbitrary*: *x*)
  **case** (*Cons a L1*)
    **then show** *?case*
      **by** (*cases x* $\in$ *set L1*, *auto*)
**qed** *auto*

**lemma** *count-rem1-out*:
  *x*$\neq$*l1* $\Longrightarrow$ *count-list* (*remove1 l1 L*) *x = count-list L x*
**by** (*induction L arbitrary*: *l1*, *auto*)

**lemma** *same-count-eq*:
  ($\forall$ *x*$\in$*set L* $\cup$ *set L1*. *count-list L x = count-list L1 x*) $\Longrightarrow$ *length L1 = length L*
$\Longrightarrow$ (*set L1 = set L*)
**proof** –
  **assume** *count*:$\forall$ *x*$\in$*set L* $\cup$ *set L1*. *count-list L x = count-list L1 x*
         **and** *len* : *length L1 = length L*
  **have** *1*:$\bigwedge$*x*. *x*$\in$ *set L* $\Longrightarrow$ *x* $\in$ *set L1*
    **proof** –
      **fix** *x*
      **assume** *H*: *x*$\in$*set L*
      **have** *H1*: *count-list L x = count-list L1 x*
        **using** *count H*
        **by** *auto*
      **with** *len H* **show** *x*$\in$ *set L1*
        **proof** (*induction L arbitrary*: *L1 x*)
          **case** (*Cons a L'*)
            **obtain** *b L1'* **where** *H*:*L1 = b#L1'*
              **using** *Cons*(*2*) *length-Suc-conv*
              **by** *metis*
            **have** *C*: *x = a* $\vee$ *x = b* $\vee$ (*x*$\neq$*a* $\wedge$ *x* $\neq$ *b*)
              **by** *auto*
            **have**   *x*$\neq$*a* $\Longrightarrow$ *x*$\neq$*b* $\Longrightarrow$ *x*$\in$ *set L1*
              **using** *Cons*(*2*–*4*) *H*
                    *Cons*(*1*)[*of L1' x*]
              **by** *fastforce*

```
          **with** *C* **show** $x \in$ *set L1*
            **using** *Cons*(*3*,*4*) *count-Suc H*
            **by** *fastforce+*
        **qed** *auto*
    **qed**
  **have** $\bigwedge x.\ x \in$ *set L1* $\Longrightarrow x \in$ *set L*
    **proof** –
      **fix** *x*
      **assume** *H*: $x \in$*set L1*
      **have** *H1*: *count-list L x* = *count-list L1 x*
        **using** *count H*
        **by** *auto*
      **with** *len H* **show** $x \in$ *set L*
        **proof** (*induction L1 arbitrary*: *L x*)
          **case** (*Cons b L1'*)
            **obtain** *a L'* **where** *H*:*L* = *a#L'*
              **using** *Cons*(*2*) *length-Suc-conv*
              **by** *metis*
            **have** *C*: $x = a \lor x = b \lor (x \neq a \land x \neq b)$
              **by** *auto*
            **have**   $x \neq a \Longrightarrow x \neq b \Longrightarrow x \in$ *set L*
              **using** *Cons*(*2*–*4*) *H*
                    *Cons*(*1*)[*of L' x*]
              **by** *fastforce*
            **with** *C* **show** $x \in$ *set L*
              **using** *Cons*(*3*,*4*) *count-Suc H*
              **by** *fastforce+*
        **qed** *auto*
    **qed**
  **with** *1* **show** *?thesis*
    **by** *auto*
**qed**
```

**lemma** *count-conv-length*:
($\forall x \in$*set L*. *count-list L x* = *count-list L1 x*) $\Longrightarrow$ *set L* $\subseteq$ *set L1* $\Longrightarrow$ *length L* $\leq$ *length L1*

```
**proof** (*induction L arbitrary*:*L1*)
  **case** (*Cons l L'*)
    **have** *A*:$\forall x \in$*set L'*. *count-list L' x* = *count-list* (*remove1 l L1*) *x*
      **using** *Cons*(*2*,*3*) *count-rem1*[*of l L1*] *count-rem1-out*
     **by** (*metis add-diff-cancel-right' count-rem1 remove1.simps*(*2*) *set-subset-Cons subsetCE*)
    **have** *set L'* $\subseteq$ *set* (*remove1 l L1*)
      **using** *Cons*(*3*) *A count-notin count-rem1*
      **by** *fastforce*
    **then show** *?case*
      **using** *Cons*(*1*)[*of remove1 l L1*] *A length-remove1*[*of l L1*]
            *subsetD*[*OF Cons*(*3*), *of l*, *simplified*]
     **by** (*metis One-nat-def Suc-le-mono Suc-pred length-Cons length-pos-if-in-set*)
```

**qed** *force*

**lemma** *same-count-set-length*:
  $(\forall x{\in}set\ L.\ count\text{-}list\ L\ x = count\text{-}list\ L1\ x) \Longrightarrow set\ L = set\ L1 \Longrightarrow length\ L = length\ L1$
**by** (*metis order-refl count-conv-length le-antisym*)

**lemma** *count-list-app*[*simp*]:
  $count\text{-}list\ (L@L1)\ x = count\text{-}list\ L\ x + count\text{-}list\ L1\ x$
**by** (*induction L arbitrary*: *L1*, *auto*)

**lemma** *distinct-fst-imp-count-1*:
  $distinct\ (fst\text{-}extract\ L) \Longrightarrow (\forall x{\in}set\ L.\ count\text{-}list\ L\ x = 1)$

**lemma** *same-count-swap*:
  $\forall x{\in}set\ L.\ count\text{-}list\ L\ x = count\text{-}list\ L1\ x \Longrightarrow set\ L1 \subseteq set\ L \Longrightarrow \forall x{\in}set\ L1.\ count\text{-}list\ L\ x = count\text{-}list\ L1\ x$
**proof** (*rule*+)
  **fix** *x*
  **assume** *hyp*: $\forall y{\in}set\ L.\ count\text{-}list\ L\ y = count\text{-}list\ L1\ y\ set\ L1 \subseteq set\ L\ x{\in}set\ L1$
  **thus** $count\text{-}list\ L\ x = count\text{-}list\ L1\ x$
    **by** *auto*
**qed**

**lemma** *inset-rem1*:
  $x{\in}\ set\ L \Longrightarrow \exists L1\ L2.\ L = L1@[x]@L2 \wedge remove1\ x\ L = L1@L2$
**proof**(*induction L arbitrary*: *x*)
  **case** (*Cons x1 L'*)
    **have** $x \in set\ L' \Longrightarrow x{\neq}x1 \Longrightarrow\ ?case$
      **proof** −
        **assume** $H{:}x{\in}\ set\ L'\ x{\neq}x1$
        **obtain** *L1 L2* **where** $H1{:}L' = L1\ @\ [x]\ @\ L2 \wedge remove1\ x\ L' = L1\ @\ L2$
          **using** *Cons*(*1*)[*OF H*(*1*)]
          **by** *auto*
        **have** $x1\ \#\ L' = (x1\#L1)\ @\ [x]\ @\ L2 \wedge remove1\ x\ (x1\ \#\ L') = (x1\#L1)\ @\ L2$
         **using** *remove1-append H*(*2*) *H1*
         **by** *auto*
        **thus** *?case* **by** *blast*
      **qed**
    **thus** *?case* **using** *Cons*(*2*) **by** *auto*
**qed** *auto*

```
lemma in-set-conv-card-Suc:
  finite L ⟹ x ∈ L ⟹∃ k. card L = Suc k
proof –
  assume H:x∈L finite L
  have 1:L = insert x (L − {x}) ∧ x ∉ (L − {x}) using insert-Diff[OF H(1)]
by blast
  have card (L − {x}) = 0 ⟶ (L − {x}) = {}
    using card-eq-0-iff[of L−{x}] finite-Diff[OF H(2), of {x}]
    by meson
  with 1 show ∃ k. card L = Suc k
    using card-Suc-eq[of L card(L−{x})]
    by metis
qed

lemma set-zip-subset:
  set (zip L TL) ⊆ set (zip L′ TL′) ⟹ length L′ = length TL′ ⟹ length L =
length TL
    ⟹ set L ⊆ set L′ ∧ set TL ⊆ set TL′
proof (induction L arbitrary: TL TL′ L′)
  case (Cons l L)
    obtain t TL1 where TL = t#TL1
      using length-Suc-conv Cons(4)
      by metis
    then show ?case
      using Cons(1)[OF - Cons(3)] Cons(4,2)
            in-set-zip[of (l,t), simplified]
      by auto
qed auto

lemma set-zip-subset-app:
 length L=length L1 ⟹ length L′=length L1′ ⟹
       set (zip L L1) ⊆ A ⟹ set (zip L′ L1′) ⊆ A ⟹ set (zip (L@L′) (L1@L1′))
⊆ A
proof (induction L arbitrary: L1 L′ L1′ A)
  case (Cons l′ La)
    obtain l1 L1a where L1 = l1#L1a
      using Cons(2) length-Suc-conv
      by metis
    then show ?case
      using Cons(1)[OF - Cons(3) - Cons(5),of L1a]
            Cons(2,4)
      by simp
qed auto
```

# **Appendix B:** Extended calculus functions

```
fun shift-L :: int ⇒ nat ⇒ lterm ⇒ lterm where
  shift-L d c LTrue = LTrue |
  shift-L d c LFalse = LFalse |
  shift-L d c (LIf t1 t2 t3) = LIf (shift-L d c t1) (shift-L d c t2) (shift-L d c t3) |
  shift-L d c (LVar k) = LVar (if k < c then k else nat (int k + d)) |
  shift-L d c (LAbs T' t) = LAbs T' (shift-L d (Suc c) t) |
  shift-L d c (LApp t1 t2) = LApp (shift-L d c t1) (shift-L d c t2) |
  shift-L d c unit = unit |
  shift-L d c (Seq t1 t2) = Seq (shift-L d c t1) (shift-L d c t2) |
  shift-L d c (t as A) = (shift-L d c t) as A |
  shift-L d c (Let var x := t in t1) =
    (if x> c then Let var (nat (int x + d)) := (shift-L d c t) in (shift-L d c t1)
     else  Let var x := (shift-L d c t) in (shift-L d (Suc c) t1)
     )  |
  shift-L d c ({|t1,t2|}) = {| shift-L d c t1 , shift-L d c t2 |} |
  shift-L d c (π1 t) = π1 (shift-L d c t) |
  shift-L d c (π2 t) = π2 (shift-L d c t) |
  shift-L d c (Tuple L) = Tuple (map (shift-L d c) L) |
  shift-L d c (Π i t)   = Π i (shift-L d c t) |
  shift-L d c (Record L LT)  = Record L (map (shift-L d c) LT) |
  shift-L d c (ProjR l t) = ProjR l (shift-L d c t) |
  shift-L d c (<|p|>) = <|p|> |
  shift-L d c (Let pattern p := t1 in t2) = (Let pattern p := (shift-L d c t1) in
(shift-L d c t2)) |
  shift-L d c (inl t as T') =  inl (shift-L d c t) as T' |
  shift-L d c (inr t as T') =  inr (shift-L d c t) as T' |
  shift-L d c (Case t of Inl x ⇒ t1 | Inr y ⇒ t2) =
    (Case (shift-L d c t) of
        Inl (if x> c then (nat (int x + d)) else x) ⇒ shift-L d (if x≤ c then Suc c
else c) t1
      | Inr (if y> c then (nat (int y + d)) else y) ⇒ shift-L d (if y≤ c then Suc c
else c) t2) |
  shift-L d c (<l:=t> as A) = <l:= shift-L d c t> as A |
  shift-L d c (Case t of L ⇒ B) =
    (Case (shift-L d c t) of L ⇒
      map (λp.(if (fst p) > c then (nat (int (fst p) + d)) else fst p , shift-L d (if
(fst p)≤c then Suc c else c) (snd p))) B)|
  shift-L d c (Fixpoint t) = Fixpoint (shift-L d c t)
```

```
function subst-L :: nat ⇒ lterm ⇒ lterm ⇒ lterm where
  subst-L j s LTrue = LTrue |
  subst-L j s LFalse = LFalse |
  subst-L j s (LIf t1 t2 t3) = LIf (subst-L j s t1) (subst-L j s t2) (subst-L j s t3) |
  subst-L j s (LVar k) = (if k = j then s else LVar k) |
  subst-L j s (LAbs T′ t) = LAbs T′ (subst-L (Suc j) (shift-L 1 0 s) t) |
  subst-L j s (LApp t1 t2) = LApp (subst-L j s t1) (subst-L j s t2) |
  subst-L j s unit = unit |
  subst-L j s (Seq t1 t2) = Seq (subst-L j s t1) (subst-L j s t2) |
  subst-L j s (t as A) = (subst-L j s t) as A |
  subst-L j s (Let var x := t in t1) =
  ((Let var x := (subst-L j s t) in (subst-L (if j ≥ x then Suc j else j) (shift-L 1 x
s) t1))) |
  subst-L j s (⦃t1,t2⦄) = ⦃subst-L j s t1, subst-L j s t2⦄ |
  subst-L j s (π1 t) = π1 (subst-L j s t) |
  subst-L j s (π2 t) = π2 (subst-L j s t) |
  subst-L j s (Π i t) = Π i (subst-L j s t) |
  subst-L j s (Tuple L) = Tuple (map (subst-L j s) L) |
  subst-L j s (Record L LT) = Record L (map (subst-L j s) LT) |
  subst-L j s (ProjR l t) = ProjR l (subst-L j s t) |
  subst-L j s (<|p|>) = <|p|> |
  subst-L j s (Let pattern p := t1 in t2) = (Let pattern p := (subst-L j s t1) in
(subst-L j s t2)) |
  subst-L j s (inl t as T′) = inl (subst-L j s t) as T′ |
  subst-L j s (inr t as T′) = inr (subst-L j s t) as T′ |
  subst-L j s (Case t of Inl x ⇒ t1 | Inr y ⇒ t2) =
    (Case (subst-L j s t) of
        Inl x ⇒ (subst-L (if j ≥ x then Suc j else j) (shift-L 1 x s) t1)
      | Inr y ⇒ (subst-L (if j ≥ y then Suc j else j) (shift-L 1 y s) t2)) |
  subst-L j s (<l:=t> as T′) = <l:=subst-L j s t> as T′ |
  subst-L j s (Case t of L ⇒ B) =
    (Case (subst-L j s t) of L ⇒ map (λp. (fst p, subst-L (if j ≥ fst p then Suc j
else j) (shift-L 1 (fst p) s) (snd p))) B) |
  subst-L j s (Fixpoint t) = Fixpoint (subst-L j s t)
by pat-completeness auto

termination
  proof (relation measure (λ(j,s,t). size t), simp-all)
    fix t :: lterm and B :: (nat × lterm) list and x :: nat × lterm
    assume x ∈ set B
     then have ¬ Suc (size-list (size-prod (λn. 0) size) B + size t) ≤ size-prod
(λn. 0) size x
      by (meson less-add-Suc1 not-less size-list-estimation′)
    then have ¬ Suc (size-list (size-prod (λn. 0) size) B + size t) ≤ size (snd x)
      by (simp add: size-prod-simp)
    then show size (snd x) < Suc (size-list (size-prod (λn. 0) size) B + size t)
      using not-less by blast
  qed (metis size-list-estimation′ lessI not-less)+
```

**fun** *Pvars* :: *Lpattern* ⇒ *nat list* **where**
*Pvars* (*V n as A*) = [*n*] |
*Pvars* (*RCD L PL*) = (*foldl* (*λx r. x* @ *r*) [] (*map Pvars PL*))

**fun** *patterns*::*lterm* ⇒ *nat list* **where**
*patterns* (<|*p*|>) = *Pvars p* |
*patterns* (*LIf c t1 t2*) = *patterns c* @ *patterns t1* @ *patterns t2* |
*patterns* (*LAbs A t1*) = *patterns t1* |
*patterns* (*LApp t1 t2*) = *patterns t1* @ *patterns t2* |
*patterns* (*Seq t1 t2*) = *patterns t1* @ *patterns t2* |
*patterns* (*t1 as A*) = *patterns t1* |
*patterns* (*Let var x* := *t1 in t2*) = *patterns t1* @ *patterns t2* |
*patterns* (⦃*t1*,*t2*⦄) = *patterns t1* @ *patterns t2* |
*patterns* (*Tuple L*) = *foldl* (*λe r. e* @ *r*) [] (*map* (*patterns*) *L*) |
*patterns* (*Record L LT*) = *foldl* (*λe r. e* @ *r*) [] (*map* (*patterns*) *LT*) |
*patterns* ($\pi$*1 t*) = *patterns t* |
*patterns* ($\pi$*2 t*) = *patterns t* |
*patterns* ($\Pi$ *i t*) = *patterns t* |
*patterns* (*ProjR l t*) = *patterns t* |
*patterns* (*Let pattern p* := *t1 in t2*) = *patterns t1* @ *patterns t2* |
*patterns* (*inl t as T'*) = *patterns t* |
*patterns* (*inr t as T'*) = *patterns t* |
*patterns* (*Case t of Inl x* ⇒ *t1* | *Inr y* ⇒ *t2*) = *patterns t* @ *patterns t1* @ *patterns t2* |
*patterns* (<*l*:=*t*> *as T'*) = *patterns t* |
*patterns* (*Case t of L* ⇒ *B*) = *patterns t* @ *foldl* (*λe r. e* @ *r*) [] (*map* (*patterns*∘*snd*) *B*) |
*patterns* (*Fixpoint t*) = *patterns t*

**inductive** *is-value-L* :: *lterm* ⇒ *bool* **where**
*VTrue* : *is-value-L LTrue* |
*VFalse*: *is-value-L LFalse* |
*VAbs* :*is-value-L* (*LAbs T' t*) |
*VUnit* :*is-value-L unit* |
*VPair* :*is-value-L v1* ⟹ *is-value-L v2* ⟹ *is-value-L* (⦃*v1*,*v2*⦄) |
*VTuple*:(⋀*i*. *0*≤*i* ⟹ *i*<*length L* ⟹ *is-value-L* (*L*!*i*)) ⟹ *is-value-L* (*Tuple L*) |
*VRCD* :(⋀*i*. *0*≤*i* ⟹ *i*<*length LT* ⟹ *is-value-L* (*LT*!*i*)) ⟹ *is-value-L* (*Record L LT*)|
*VSumL* :*is-value-L v* ⟹ *is-value-L* (*inl v as A*)|
*VSumR* :*is-value-L v* ⟹ *is-value-L* (*inr v as A*)|
*VVa* :*is-value-L v* ⟹ *is-value-L* (<*l*:=*v*> *as A*)

```
function FV :: lterm ⇒ nat set where
  FV LTrue = {} |
  FV LFalse = {} |
  FV (LIf t1 t2 t3) = FV t1 ∪ FV t2 ∪ FV t3 |
  FV (LVar x) = {x} |
  FV (LAbs T1 t) = image (λx. x − 1) (FV t − {0}) |
  FV (LApp t1 t2) = FV t1 ∪ FV t2 |
  FV unit = {} |
  FV (Seq t1 t2) = FV t1 ∪ FV t2 |
  FV (t as A) = FV t |
  FV (Let var x := t in t1) = image (λy. if (y>x) then y−1 else y) (FV t1 −{x})
∪ FV t |
  FV (⦃t1,t2⦄) = FV t1 ∪ FV t2 |
  FV (π1 t) =  FV t |
  FV (π2 t) =  FV t |
  FV (Tuple L) = foldl (λx r. x ∪ r) {} (map FV L) |
  FV (Π i t) = FV t |
  FV (Record L LT) = foldl (λx r. x ∪ r) {} (map FV LT) |
  FV (ProjR l t) = FV t |
  FV (Pattern p) = {} |
  FV (Let pattern p := t1 in t2) = FV t1 ∪ FV t2 |
  FV (inl t as A) = FV t |
  FV (inr t as A) = FV t |
  FV (Case t of Inl x ⇒ t1 | Inr y ⇒ t2) = image (λy. if (y>x) then y − 1 else
y) (FV t1 −{x}) ∪
                                           image (λz. if (z>y) then z − 1 else z) (FV
t2 −{y}) ∪
                                           FV t  |
  FV (<L:=t> as A) = FV t |
  FV (Case t of L ⇒ B) = FV t ∪ foldl (λx r. x ∪ r) {} (map (λp. image (λy.
if (y>fst p) then y − 1 else y) (FV (snd p) − {fst p})) B) |
  FV (Fixpoint t) = FV t
by pat-completeness auto

termination
  proof (relation measure (λt. size t), simp-all)
    fix t :: lterm and B :: (nat × lterm) list and x :: nat × lterm
    assume x ∈ set B
     then have ¬ Suc (size-list (size-prod (λn. 0) size) B + size t) ≤ size-prod
(λn. 0) size x
      by (meson less-add-Suc1 not-less size-list-estimation')
    then have ¬ Suc (size-list (size-prod (λn. 0) size) B + size t) ≤ size (snd x)
      by (simp add: size-prod-simp)
    then show size (snd x) < Suc (size-list (size-prod (λn. 0) size) B + size t)
      by (meson not-less)
  qed (metis size-list-estimation' lessI not-less )+
fun p-instantiate::(nat ⇀ lterm) ⇒ Lpattern ⇒ lterm where
p-instantiate Σ (V k as A) = (case Σ k of Some t′ ⇒ t′ | None ⇒ <|V k as A|>)|
p-instantiate Σ (RCD L PL) = <|RCD L PL|>

fun fill::(nat ⇀ lterm) ⇒ lterm ⇒ lterm where
fill Σ (Pattern p)                 = p-instantiate Σ p |
fill Σ (LIf c t1 t2)               = LIf (fill Σ c) (fill Σ t1) (fill Σ t2) |
fill Σ (LAbs A t1)                 = LAbs A (fill Σ t1) |
fill Σ (LApp t1 t2)                = LApp (fill Σ t1) (fill Σ t2) |
fill Σ (Seq t1 t2)                 = Seq (fill Σ t1) (fill Σ t2) |
fill Σ (t1 as A)                   = (fill Σ t1) as A |
fill Σ (Let var x := t1 in t2)     = (Let var x := (fill Σ t1) in (fill Σ t2)) |
fill Σ (⦃t1,t2⦄)                   = ⦃(fill Σ t1), (fill Σ t2)⦄ |
fill Σ (Tuple L)                   = Tuple (map (fill Σ) L) |
fill Σ (Record L LT)               = Record L (map (fill Σ) LT) |
fill Σ (π1 t)                      = π1 (fill Σ t) |
fill Σ (π2 t)                      = π2 (fill Σ t) |
fill Σ (Π i t)                     = Π i (fill Σ t) |
fill Σ (ProjR l t)                 = ProjR l (fill Σ t) |
fill Σ (Let pattern p := t1 in t2) = (Let pattern p := (fill Σ t1) in (fill Σ t2)) |
fill Σ (inl t as A) = inl (fill Σ t) as A|
fill Σ (inr t as A) = inr (fill Σ t) as A|
fill Σ (Case t of Inl x ⇒ t1 | Inr y ⇒ t2) = (Case (fill Σ t) of Inl x ⇒ (fill Σ t1)
| Inr y ⇒ (fill Σ t2))|
fill Σ (<l:=t> as A) = <l:=(fill Σ t)> as A|
fill Σ (Case t of L ⇒ B) = (Case (fill Σ t) of L ⇒ map (λp. (fst p, fill Σ (snd
p))) B)|
fill Σ (Fixpoint t) = Fixpoint (fill Σ t) |
fill Σ t = t
```

# Bibliography

[1] Benjamin C. Pierce. ***Types and Programming Languages***. The MIT Press, 1st edition, 2002.

[2] Martin Desharnais. **Formalizing types and programming languages in isabelle/hol**. Bachelor's thesis, Ecole de technologie supérieure, 2014.

[3] Martin Desharnais, **Repository:**
https://github.com/authchir/log792-type-systems-formalization

[4] Wolfram Kahl. **Basic pattern matching calculi: Syntax, reduction, confluence, and normalisation**. SQRL Report 16, Software Quality Research Laboratory, McMaster Univ., 10 2003.

[5] Tobias Nipkow, **List_index.thy**, AFP.
https://www.isa-afp.org/browser_info/current/AFP/List-Index/List_Index.html

[6] **implementation**. pages 74–75. url:
https://isabelle.in.tum.de/dist/Isabelle2016-1/doc/implementation.pdf

[7] **datatypes**. url:
https://isabelle.in.tum.de/dist/Isabelle2016-1/doc/datatypes.pdf

[8] Kunihiko Chikaya.
**Proof 11: ab+bc+ca≤aa+bb+cc** .
**Original source:** A. Engel, **Problem-Solving Strategies**, 1998
http://www.cut-the-knot.org/m/Algebra/AbBcCaLeAaBbCc.shtml

[9] **eisbach**. url:
https://isabelle.in.tum.de/dist/Isabelle2016-1/doc/eisbach.pdf

[10] N.G de Bruijn. **Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem**. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[11] W. W. Tait. **Intensional interpretations of functionals of finite type i**. *Journal of Symbolic Logic*, 1967:–, 1967.

[12] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. **Mechanized metatheory for the masses: The poplmark challenge.** In *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[13] Derek Dreyer.

**Rustbelt: Logical foundations for the future of safe systems programming** .
`http://plv.mpi-sws.org/rustbelt/#project`

[14] Nada Amin and Ross Tate. **Java and scala's type systems are unsound: The existential crisis of null pointers**. *SIGPLAN Not.*, 51(10):838–848, October 2016.

[15] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. **A trusted mechanised javascript specification**. *SIGPLAN Not.*, 49(1):87–100, January 2014.

[16] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. **The design and formalization of mezzo, a permission-based programming language**. *ACM Trans. Program. Lang. Syst.*, 38(4):14:1–14:94, 2016.