M1 INTERNSHIP REPORT

September, 12th

Testing judgements of Martin-Löf type theory

Yann DUPLOUY
yduplouy@ens-cachan.fr

Internship done in the *Department of Computer Science and Engineering* at *Chalmers University of Technology*, under the supervision of Peter Dybjer

INTRODUCTION

WHERE THE INTERNSHIP TOOK PLACE

I've done this internship in the division of *Computing Science*, of the *Department of Computer Science and Engineering*, at *Chalmers University of Technology* (in the Johannesberg campus, in Göteborg), under the supervision of Peter Dybjer.

The division of *Computing Science* is composed of four different research groups :

- the Language Technology research group, working on natural language processing and compiler tools
- the Algorithms and Optimization research group,
- the Bioinformatics and System Biology research group,
- and finally, the Programming Logic research group, in which I've worked, and that is working on type theory, program verification and functional languages.

WHAT THE INTERNSHIP IS ABOUT

The Curry-Howard correspondence is the known direct relationship between computer programs and mathematical proofs. One of the main features of Martin-Löf's type theory is that there are "meaning explanations" of the judgments of the theory. These give an approach to the semantics of the theory where the meaning of propositions and logical connective are not seen in terms of a mathematical interpretation which assumes an existing mathematical framework (such as set theory), but directly in terms of computations to canonical form of the constituent expressions.

In this internship, as explained by Peter Dybjer in his article *Program Testing and the Meaning Explanations of Intuitionistic Type Theory* [1], we want to implement a testing procedure for all the judgements of Martin-Löf type theory.

CONTENTS

Page

I — PRELIMINARIES

In this section, we will first describe what a Krivine Abstract Machine is, then see previous work done by Rodolphe Lepigre [2] (who has written a Testing Krivine Abstract Machine for simpler type theories) and then see the definition of Martin-Löf type theory.

I.1. THE KRIVINE ABSTRACT MACHINE

A Krivine Abstract Machine is an abstract machine designed for computing a lambda expression into a normal form. We will define what a *KAM* is, but we need to present the simple language of the λ-calculus first. In this language, a term is either a variable, an abstraction or an application :

$$t := x \mid \lambda x.t \mid t\ t$$

In order to define what a *Krivine Abstract Machine* (KAM) is, we first need to define environments and closures. An environment is a function σ that maps variables to closures, and a closure a pair of a term and an environment.

The *KAM* will also contain a stack, which is a list of closures :

$$\pi := \pi_0 \mid (t, \sigma).\pi$$

A Krivine Abstract Machine state is defined by a term, an environment, and a stack. The computation is defined by a list of possible transitions. If there's no transition appliable to a specific state, then the machine stops and the current state becomes the final states. In order to normalize lambda expressions, the transition function of the *KAM* should be the following :

$$\begin{array}{rcl} \langle t_1 t_2, \pi, \sigma\rangle & \to & \langle t_1, (t_2, \sigma).\pi, \sigma\rangle \\ \langle \lambda x.t, \sigma_x.\pi, \sigma\rangle & \to & \langle t, \pi, \sigma + \{x \mapsto \sigma_x\}\rangle \\ \langle x, \pi, \sigma\rangle & \to & \langle \sigma_1(x), \pi, \sigma_2(x)\end{array}$$

(The first rule adds t_2 to the stack of terms to be worked on later, so the machine can work on t_1. The second rule tries to compute an abstraction which need an argument - that is given by an application - so it takes it from the stack and adds it to the environment. Finally the third rule uses the environment to replace variables by their value in the computation[1].)

We will run a specific example to show how the *KAM* works. We consider the term $(\lambda x.xx)(\lambda x.x)$. The starting state of the machine is the following :

$$\langle (\lambda x.xx)(\lambda x.x), \pi_0, \sigma\rangle$$

From this starting state, we can use the computation rules :

$$\begin{array}{rl} & \langle (\lambda x.xx)(\lambda x.x), \pi_0, \emptyset\rangle \\ \to & \langle \lambda x.xx, (\lambda x.x, \emptyset).\pi_0, \emptyset\rangle \\ \to & \langle xx, \pi_0, \{x \mapsto (\lambda x.x), \emptyset\}\rangle \\ \to & \langle x, (x, \{x \mapsto (\lambda x.x), \emptyset\}).\pi_0, \{x \mapsto (\lambda x.x), \emptyset\}\rangle \\ \to & \langle \lambda x.x, (x, \{x \mapsto (\lambda x.x), \emptyset\}).\pi_0, \emptyset\rangle \\ \to & \langle x, \pi_0, \{x \mapsto (\lambda x.x), \emptyset\}\rangle \\ \to & \langle \lambda x.x, \pi_0, \emptyset\rangle \end{array}$$

So we reach the identity function as the final state, as expected.

[1] σ maps variable names to closures, and σ_1 then maps variable names to the first argument of the closure, σ_2 to the other argument

I.2. TESTING *KAM* ON THE PCF SYSTEM

The work described in this subsection has been done by Rodolphe Lepigre [2] during his internship in Chalmers, also under the supervision of Peter Dybjer.

The *PCF* is a simple typed language based on λ-calculus. It has two atomic types, $\mathbb{B}$ (or *Bool*, booleans) and $\mathbb{N}$ (naturals), along with a function type - and a fixed point combinator. So the types are :

$$T ::= \mathbb{B} \mid \mathbb{N} \mid T \to T$$

and the terms are :

$$t ::= x \mid \lambda x.t \mid t\ t \mid \mathtt{Y}\ t \mid \mathtt{T} \mid \mathtt{F} \mid \mathrm{case}_{\mathbb{B}}\ t\ t\ t \mid \mathtt{Z} \mid \mathtt{S}\ t \mid \mathrm{case}_{\mathbb{N}} t\ t\ t$$

These terms are enough to define, for example, the primitive recursion on functions :

$$\mathrm{natrec} \stackrel{\mathrm{def}}{=} \mathtt{Y}(\lambda r, t, g, n.\ \mathrm{case}_{\mathbb{N}}\ n\ t(\lambda p.\ g\ p(r\ t\ g\ p)))$$

(where $\lambda r, t, g, n$ is a faster way to write $\lambda r \lambda t \lambda g \lambda n$)

I.2.1. EXTENDING THE *KAM* TO *PCF* SYSTEM

The rules we have used for the simple lambda-calculus are still used there, but as we have new terms added to our language, we need new transition rules to be able to handle these :

$$\begin{aligned}
\langle \mathtt{Y}\ M, \pi, \sigma\rangle &\overset{\Gamma}{\rightsquigarrow} \langle M, (\mathtt{Y}\ M, \sigma).\pi, \sigma\rangle \\
\langle \mathrm{case}_{\mathbb{B}}\ M\ N_{\mathtt{T}}\ N_{\mathtt{F}}, \pi, \sigma\rangle &\overset{\Gamma}{\rightsquigarrow} \langle M, (N_{\mathtt{T}}, \sigma).(N_{\mathtt{F}}, \sigma).\pi, \sigma\rangle \\
\langle \mathtt{T}, (N_{\mathtt{T}}, \sigma_{\mathtt{T}}).(N_{\mathtt{F}}, \sigma_{\mathtt{F}}).\pi, \sigma\rangle &\overset{\Gamma}{\rightsquigarrow} \langle N_{\mathtt{T}}, \pi, \sigma_{\mathtt{T}}\rangle \\
\langle \mathtt{F}, (N_{\mathtt{T}}, \sigma_{\mathtt{T}}).(N_{\mathtt{F}}, \sigma_{\mathtt{F}}).\pi, \sigma\rangle &\overset{\Gamma}{\rightsquigarrow} \langle N_{\mathtt{F}}, \pi, \sigma_{\mathtt{F}}\rangle \\
\langle \mathrm{case}_{\mathbb{N}}\ M\ N_{\mathtt{Z}}\ N_{\mathtt{S}}, \pi, \sigma\rangle &\overset{\Gamma}{\rightsquigarrow} \langle M, (N_{\mathtt{Z}}, \sigma).(N_{\mathtt{S}}, \sigma).\pi, \sigma\rangle \\
\langle \mathtt{Z}, (N_{\mathtt{Z}}, \sigma_{\mathtt{Z}}).(N_{\mathtt{S}}, \sigma_{\mathtt{S}}).\pi, \sigma, \tau\rangle &\overset{\Gamma}{\rightsquigarrow} \langle N_{\mathtt{Z}}, \pi, \sigma_{\mathtt{Z}}\rangle \\
\langle \mathtt{S}\ M, (N_{\mathtt{Z}}, \sigma_{\mathtt{Z}}).(N_{\mathtt{S}}, \sigma_{\mathtt{S}}).\pi, \sigma, \tau\rangle &\overset{\Gamma}{\rightsquigarrow} \langle N_{\mathtt{S}}\ M, \pi, \sigma_{\mathtt{S}}\rangle
\end{aligned}$$

This Krivine Abstract machine is now able to compute any term of the *PCF* language into a normal form.

I.2.2. LAZY *TKAM*

The *TKAM* used in Rodolphe Lepigre's work is a bit different from the original version defined by Pierre Clairambault. The aim of this Lazy TKAM is still to compute open expression of the *PCF* language, this time introducing metavariables in the calculation - which are called channels. When a value is needed for such a channel, we will generate an *atomic normal form* in order to be able to continue the computation.

We are extending the language with channels c that carry a context Γ and a type T (and are then written c^T_Γ). While *KAM* states were triples, *LTKAM* states are quadruples $\langle t, \pi, \sigma, \tau\rangle$, where (t, σ) is still a closure (t being a term, and σ an environment), π still being a stack of closures, and the newly introduced τ a partial function, mapping channels to *atomic normal forms*. We define the transition function of the machine (that computes *PCF*) as follows (we begin with the "continuation" states). It is the same as the one used for the *KAM*, except there is a new rule for channel instantiation.

$$
\begin{array}{rcll}
\langle x, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle \sigma_1(x), \pi, \sigma_2(x), \tau\rangle & \\
\langle \lambda x.M, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, \pi_0, \sigma + \{x \mapsto (c_\Gamma^{\Gamma(x)}, \sigma)\}, \tau\rangle & \\
\langle \lambda x.M, \sigma_N.\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, \pi, \sigma + \{x \mapsto \sigma_N\}, \tau\rangle & \\
\langle M\ N, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, (N, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathtt{Y}\ M, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, (\mathtt{Y}\ M, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathrm{case}_{\mathbb{B}}\ M\ N_{\mathtt{T}}\ N_{\mathtt{F}}, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, (N_{\mathtt{T}}, \sigma).(N_{\mathtt{F}}, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathtt{T}, (N_{\mathtt{T}}, \sigma_{\mathtt{T}}).(N_{\mathtt{F}}, \sigma_{\mathtt{F}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{T}}, \pi, \sigma_{\mathtt{T}}, \tau\rangle & \\
\langle \mathtt{F}, (N_{\mathtt{T}}, \sigma_{\mathtt{T}}).(N_{\mathtt{F}}, \sigma_{\mathtt{F}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{F}}, \pi, \sigma_{\mathtt{F}}, \tau\rangle & \\
\langle \mathrm{case}_{\mathbb{N}}\ M\ N_{\mathtt{Z}}\ N_{\mathtt{S}}, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, N_{\mathtt{Z}}, \sigma).(N_{\mathtt{S}}, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathtt{Z}, (N_{\mathtt{Z}}, \sigma_{\mathtt{Z}}).(N_{\mathtt{S}}, \sigma_{\mathtt{S}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{Z}}, \pi, \sigma_{\mathtt{Z}}, \tau\rangle & \\
\langle \mathtt{S}\ M, (N_{\mathtt{Z}}, \sigma_{\mathtt{Z}}).(N_{\mathtt{S}}, \sigma_{\mathtt{S}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{S}}\ M, \pi, \sigma_{\mathtt{S}}, \tau\rangle & \\
\langle c, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle \tau(c), \pi, \sigma, \tau\rangle & c \in \mathrm{dom}(\tau)
\end{array}
$$

When a state of the following form is reached, we need to do a channel instantiation :

$$\langle c_\Gamma^A, \pi, \sigma, \tau\rangle \text{ when } c \notin \mathrm{dom}(\tau)$$

In this case, an *atomic normal form* M, of type A within context Γ is generated, and the computation will resume with the following state :

$$\langle M, \pi, \sigma, \tau + \{c_\Gamma^A \mapsto M\}\rangle$$

The definition of such *atomic normal forms* of type T, in context Γ, for *PCF* is split into three cases :

- when T has the canonical form $\vec{A} \to G$ with $G \in \{\mathbb{N}, \mathbb{B}\}$:

$$\mathrm{anf}_\Gamma(\vec{A} \to G) = \{\lambda\vec{x}.M \mid M \in \mathrm{anf}_{\vec{x}:\vec{A}.\Gamma}(G)\}$$

- when $T = \mathbb{B}$:

$$
\begin{array}{rcl}
\mathrm{anf}_\Gamma(\mathbb{B}) = \{\mathtt{T}, \mathtt{F}\} & \cup & \{\mathrm{case}_{\mathbb{B}}\ (x\ \vec{c})\ d\ e \mid x : \vec{A} \to \mathbb{B} \in \Gamma, c_i \in C_\Gamma^{A_i}, d, e \in C_\Gamma^{\mathbb{B}}\} \\
& \cup & \{\mathrm{case}_{\mathbb{N}}\ (x\ \vec{c})\ d\ e \mid x : \vec{A} \to \mathbb{N} \in \Gamma, c_i \in C_\Gamma^{A_i}, d \in C_\Gamma^{\mathbb{B}}, e \in C_\Gamma^{\mathbb{N}\to\mathbb{B}}\}
\end{array}
$$

- and when $T = \mathbb{N}$:

$$
\begin{array}{rcl}
\mathrm{anf}_\Gamma(\mathbb{N}) = \{\mathtt{Z}\} & \cup & \{\mathtt{S}\ c \mid c \in C_\Gamma^{\mathbb{N}}\} \\
& \cup & \{\mathrm{case}_{\mathbb{B}}\ (x\ \vec{c})\ d\ e \mid x : \vec{A} \to \mathbb{B} \in \Gamma, c_i \in C_\Gamma^{A_i}, d, e \in C_\Gamma^{\mathbb{N}}\} \\
& \cup & \{\mathrm{case}_{\mathbb{N}}\ (x\ \vec{c})\ d\ e \mid x : \vec{A} \to \mathbb{N} \in \Gamma, c_i \in C_\Gamma^{A_i}, d \in C_\Gamma^{\mathbb{N}}, e \in C_\Gamma^{\mathbb{N}\to\mathbb{N}}\}
\end{array}
$$

where C_Γ^T is the set of channels of type T with context Γ.

I.2.3. TESTING JUDGMENTS OF *PCF*

We have talked about a testing *KAM* but haven't yet tested any judgements. We will do some changes to the *TKAM* in this part, so we can test judgements of the following form :

$$\Gamma \vdash a : T$$

We can see that testing the following the judgements is equivalent :

$$\Gamma \vdash f : A \to B \text{ and } \Gamma, x : A \vdash fa : B$$

so all judgements can be reduced to judgements of one of the following forms :

$$\Gamma \vdash a : \mathbb{N} \text{ or } \Gamma \vdash a : \mathbb{B}$$

We also need to make some changes to the transition function of the *TKAM* we've defined in the previous section, because we now need it to have different types of states :

- *continuation* states – which are exactly the same as the states wei've used until here
- *instantiation* states – when we need to instantiate a channel
- *value* states – when the computation has finished and returned the normal form
- *error* states – when the computation gets into a state that should not be reached

We begin by the *continuation* rules, leading to *continuation* states. They are essentially the same than in the Lazy TKAM (as this is another extension of what has been described before) :

$$\begin{array}{rcll}
\langle \mathrm{case}_{\mathbb{B}}\ M\ N_{\mathtt{T}}\ N_{\mathtt{F}}, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, (N_{\mathtt{T}}, \sigma).(N_{\mathtt{F}}, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathtt{T}, (N_{\mathtt{T}}, \sigma_{\mathtt{T}}).(N_{\mathtt{F}}, \sigma_{\mathtt{F}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{T}}, \pi, \sigma_{\mathtt{T}}, \tau\rangle & \\
\langle \mathtt{F}, (N_{\mathtt{T}}, \sigma_{\mathtt{T}}).(N_{\mathtt{F}}, \sigma_{\mathtt{F}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{F}}, \pi, \sigma_{\mathtt{F}}, \tau\rangle & \\
\langle M\ N, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, (N, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \lambda x.M, \sigma_N.\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, \pi, \sigma + \{x \mapsto \sigma_N\}, \tau\rangle & \\
\langle x, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle \sigma_1(x), \pi, \sigma_2(x), \tau\rangle & x \in \mathrm{dom}(\sigma) \\
\langle x, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle c_\Gamma^{\Gamma(x)}, \pi, \sigma + \{x \mapsto (c_\Gamma^{\Gamma(x)}, \sigma)\}, \tau\rangle & x \notin \mathrm{dom}(\sigma) \\
\langle c, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle \tau(c), \pi, \sigma, \tau\rangle & c \in \mathrm{dom}(\tau) \\
\langle \mathrm{case}_{\mathbb{N}}\ M\ N_{\mathtt{Z}}\ N_{\mathtt{S}}, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, N_{\mathtt{Z}}, \sigma).(N_{\mathtt{S}}, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathtt{Z}, (N_{\mathtt{Z}}, \sigma_{\mathtt{Z}}).(N_{\mathtt{S}}, \sigma_{\mathtt{S}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{Z}}, \pi, \sigma_{\mathtt{Z}}, \tau\rangle & \\
\langle \mathtt{S}\ M, (N_{\mathtt{Z}}, \sigma_{\mathtt{Z}}).(N_{\mathtt{S}}, \sigma_{\mathtt{S}}).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{S}}\ M, \pi, \sigma_{\mathtt{S}}, \tau\rangle &
\end{array}$$

The transition rules leading to *value* states are the following :

$$\begin{array}{rcl}
\langle \mathtt{T}, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{value } \mathtt{T} \\
\langle \mathtt{F}, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{value } \mathtt{F} \\
\langle \lambda x.M, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{value } \lambda x.M \\
\langle \mathtt{S}M, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{value } \mathtt{S}M \\
\langle \mathtt{Z}, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{value } \mathtt{Z}
\end{array}$$

The group of transition rules leading to *error* states are the following. When we have a boolean or a natural and something in the stack, we expect the stack to contain at least two elements (which are created by $\mathrm{case}_{\mathbb{B}}$ and $\mathrm{case}_{\mathbb{N}}$). When it contains only one element, the computation cannot continue – and we don't have a value (the stack of closures should be empty) : in this case, we have an error.

$$\begin{array}{rcl}
\langle \mathtt{T}, (N, \sigma_N).\pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{error} \\
\langle \mathtt{F}, (N, \sigma_N).\pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{error} \\
\langle \mathtt{Z}, (N, \sigma_N).\pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{error} \\
\langle \mathtt{S}\ M, (N, \sigma_N).\pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{error}
\end{array}$$

Now, if we want to test the following judgement :

$$\Gamma \vdash M : \mathbb{B}$$

we run the *TKAM* on M, by doing a full computation from the following state (until it reaches a state that's not a *continuation* state)

$$\langle M, \pi_0, \emptyset, \emptyset \rangle$$

If the result is an *error* states, then the test fails. If it is a *value* state of the correct type (here $\mathbb{B}$, so `T` or `F`) then the test succeeds. If it's a *value* state of the wrong type, the test fails. And if it's a *value* state which is a λ-abstraction, the test will also fail (because we are simplifying judgements when testing functions, as suggested at the beginning of this section). The method is essentially the same for naturals, except that when we have a term `S` T we have then to test whether T is a natural or not.

If the result is an *instantiation* state, we instantiate the channel with an *atomic normal form*, and then run the *TKAM* again on this newly generated state, and then act according to the result. There is no guarantee this computation will stop, and in this case we can't conclude on the issue of the test.

I.3. MARTIN-LÖF TYPE THEORY

Martin-Löf type theory, also called intuitionistic or constructive type theory was introduced in 1972, in [3]. It add new structures to the previous systems we've already described, such as dependent types. For example, $\mathrm{Vec}(\mathbb{R}, n)$, the type of n-tuples of real numbers, may be defined in this type theory.

Another point of interest is the fact that, in this type theory, types are also terms - and they can be typed by a Universe.

Martin-Löf type theory's terms are constructed as follows :

$$t := x \mid \lambda x.t \mid T \mid \mathrm{I}(T, a, b) \mid \mathtt{Y} \mid \mathtt{Z} \mid \mathtt{S}t \mid \mathtt{T} \mid \mathtt{F}$$

where

$$T := \mathbb{B} \mid \mathbb{N} \mid \prod_{x:T} T \mid \sum_{x:T} T$$

We are still defining primitive recursion the same way it has been done before :

$$\mathrm{natrec} \stackrel{\mathrm{def}}{=} \mathtt{Y}(\lambda r, t, g, n.\ \mathrm{case}_{\mathbb{N}}\ n\ t(\lambda p.\ g\ p(r\ t\ g\ p)))$$

I.3.1. TYPES AND TYPE FORMERS

We're using several basic types to implement this type theory. Usually, any finite type - such as the empty type $\bot$, the unit type $\top$ or the boolean type $\mathbb{B}$ are enough to define Martin-Löf type theory.

There are several types of type formers in this type theory, which are :

- $\prod_{x:A} B$, constructing product types, that are used to represent the type of functions, taking a term of type A and returning a term of type $B(x), x : A$ (as B may depend of some variable, typed by A). For example, $\prod_{n:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, n)$ designates the type of functions taking as a variable a natural, and returning a n-tuple of reals.
- $\sum_{x:A} B$, constructing sum types that can be seen as the usual Cartesian product, forming pairs where the type of the second component depends of the first. For example, the type $\sum_{n:\mathbb{N}} Vec(\mathbb{R}, n)$ may be used to model lists - with the first variable giving the length of the list, and the second variable the list itself.

- $I(A, a, b)$, is the type of the proofs that a is equal of b when $a, b \in A$ and is called an equality type. You may use that type former to type the proof that a type A is equal to a type B, as types are terms.

Simpler functions or sums are also written this way, so $\prod_{x:\mathbb{N}} \mathbb{R}$ is the way to write the type, usually written $\mathbb{N} \to \mathbb{R}$, of the functions taking a natural and returning a real. Also, $\sum_{x:\mathbb{N}} \mathbb{B}$ is the type of the pairs of a natural number and a boolean ($\mathbb{N} \times \mathbb{B}$).

Some types are defined inductively, such as the natural numbers that are generated starting by $0 : \mathbb{N}$ and using the function $S : \mathbb{N} \to \mathbb{N}$ to generate other naturals.

We can also define types using other functions, for example, the type designing the vectors of naturals $\mathrm{Vec}(A, n)$ may be defined as : $\mathrm{Vec}(A, n) = \mathrm{natrec}(\lambda T_1, T_2 \sum_{x:A} T_2)\ A \top\ n$

I.3.2. UNIVERSES

As we've briefly mentioned before, types are also terms – and they can be typed by universes. In this report and in the implementation, we are working on Russell-style universes (as we follow the terminology of [4]) where objects of a universe are types. We usually define $\mathcal{U}$ as the universe containing every type described before ($\mathbb{N}, \mathbb{B}$, and the type formed with the formers given in the previous section.

Moreover, we usually have a hierarchy of universes which is cumulative (every type in $\mathcal{U}_i$ is also in $\mathcal{U}_j$ for any $j \geq i$). As the universes are also types, they are typed by a universe of a higher order.

II — THE HASKELL MODULE FOR TESTING JUDGEMENTS OF MARTIN-LÖF'S TYPE THEORY

This section will describe a Haskell module, which is the implementation of a Testing Kriving Abstract Machine adapted for testing judgements of the form

$$\Gamma \vdash a : T$$

where a is a term of Martin-Löf type theory.

As any Haskell module, we need to give a name to the module and import the necessary libraries. We need `System.Random` for randomly generating atomic normal forms (it will be explained later more precisely)

```
module TestMartinLof where

import Data.Maybe ( fromJust )
import System.Random ( StdGen , next , newStdGen )
```

We have seen another Testing KAM in the previous section. It was used for testing judgements in *PCF*. In the next section, we will describe the new TKAM that will be used for judgements of Martin-Löf type theory. It is based on notes from Pierre Clairambault and Peter Dybjer.

II.1. USING TKAM TO TEST JUDGEMENTS OF MARTIN-LÖF TYPE THEORY

We are now testing directly any judgement of the form :

$$\Gamma \vdash a : T$$

so we can't use the same states as the *TKAM* that has been used for testing judgements in *PCF* – as we cannot return a value. We are now using the different types of states :

- *continuation* states – which are exactly the same as the states we've used until here
- *instantiation* states – when we need to instantiate a channel
- *success* states – when the TKAM has reached a simple judgement which can be directly tested and has been found correct
- *error* states

In order to simplify some rules, we're using a notation for missing parameters in a construction, which will be noted as _. It is only used in *continuation* states, so the *TKAM* can work on a specific parameter before working on the whole construction again. The main use of this specific constructor is to simplify both sides of a judgement before working on the whole judgement, as we can see below :

II.1.1. CONTINUATION RULES

The first rules are there for simplifying the whole $M : T$ when either the term or the type is not a value. We first simplify the term before simplifying the type, and then we continue the computation with the whole judgement.

$$\begin{aligned}
\langle M : T, \pi, \sigma, \tau\rangle &\overset{\Gamma}{\rightsquigarrow} \langle T, (M : _, \sigma).\pi, \sigma, \tau\rangle && \text{when } M \notin V, T \notin V\\
\langle M : V, \pi, \sigma, \tau\rangle &\overset{\Gamma}{\rightsquigarrow} \langle M, (_ : V, \sigma).\pi, \sigma, \tau\rangle && \text{when } M \notin V\\
\langle V, (M : _, \sigma).\pi, \sigma_N, \tau\rangle &\overset{\Gamma}{\rightsquigarrow} \langle M : V, \pi, \sigma, \tau\rangle\\
\langle V, (_ : T, \sigma).\pi, \sigma_N, \tau\rangle &\overset{\Gamma}{\rightsquigarrow} \langle V : T, \pi, \sigma, \tau\rangle
\end{aligned}$$

where V is the set of values (or already simplified terms)

For simplifying terms alone, we use again the rules used in the LTKAM used for testing *PCF*, with different formulations so we don't have any interference between the different kinds of constructors :

$$\begin{array}{rcll}
\langle \mathrm{case}_{\mathbb{B}}\ M\ N_{\mathtt{T}}\ N_{\mathtt{F}}, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, (\mathrm{case}_{\mathbb{B}}\ _\ N_{\mathtt{T}}\ N_{\mathtt{F}}, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathtt{T}, (\mathrm{case}_{\mathbb{B}}\ _\ N_{\mathtt{T}}\ N_{\mathtt{F}}, \sigma).\pi, \sigma_2, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{T}}, \pi, \sigma, \tau\rangle & \\
\langle \mathtt{F}, (\mathrm{case}_{\mathbb{B}}\ _\ N_{\mathtt{T}}\ N_{\mathtt{F}}, \sigma).\pi, \sigma_2, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{F}}, \pi, \sigma, \tau\rangle & \\
\langle \mathrm{case}_{\mathbb{N}}\ M\ N_{\mathtt{Z}}\ N_{\mathtt{S}}, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, (\mathrm{case}_{\mathbb{B}}\ _\ N_{\mathtt{Z}}\ N_{\mathtt{S}}, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \mathtt{Z}, (\mathrm{case}_{\mathbb{N}}\ _\ N_{\mathtt{Z}}\ N_{\mathtt{S}}, \sigma).\pi, \sigma_2, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{Z}}, \pi, \sigma, \tau\rangle & \\
\langle \mathtt{S}\ M, (\mathrm{case}_{\mathbb{N}}\ _\ N_{\mathtt{Z}}\ N_{\mathtt{S}}, \sigma).\pi, \sigma_2, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle N_{\mathtt{S}}\ M, \pi, \sigma, \tau\rangle & \\
\langle M\ N, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, \pi, \sigma + (N, \sigma).\pi, \sigma, \tau\rangle & \\
\langle \lambda x.M, \sigma_N.\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, \pi, \sigma + \{x \mapsto \sigma_N\}, \tau\rangle & \\
\langle x, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle \sigma_1(x), \pi, \sigma_2(x), \tau\rangle & x \in \mathrm{dom}(\sigma) \\
\langle x, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle c_\Gamma^{\Gamma(x)}, \pi, \sigma + \{x \mapsto (c_\Gamma^{\Gamma(x)}, \sigma)\}, \tau\rangle & x \notin \mathrm{dom}(\sigma) \\
\langle c, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle \tau(c), \pi, \sigma, \tau\rangle & c \in \mathrm{dom}(\tau)
\end{array}$$

These rules apply to new constructs such as the pair constructor and the two projections :

$$\begin{array}{rcl}
\langle \mathrm{p}_1(M), \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, \mathrm{p}_1(_).\pi, \sigma, \tau\rangle \\
\langle \mathrm{p}_2(M), \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M, \mathrm{p}_2(_).\pi, \sigma, \tau\rangle \\
\langle Pair(A, B), \mathrm{p}_1(_).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle A, \pi, \sigma, \tau\rangle \\
\langle Pair(A, B), \mathrm{p}_2(_).\pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle B, \pi, \sigma, \tau\rangle
\end{array}$$

Some specific judgements can be simplified as a whole, such as pairs or some lambda-abstractions. For the abstraction, we first have to test if M is of type B when the argument x is of type A (we then need to generate an atomic normal form of type A to be able to do this test). For the pair (M, N), we first need to verify if M is of type A and then to verify if N is of type B.

$$\begin{array}{rcl}
\langle \lambda x.M : \prod_{x:A} B, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M : B, \pi, \sigma + \{x \mapsto c_\emptyset^A\}, \tau\rangle \\
\langle \mathrm{Pair}(M, N) : \sum_{x:A} B, \pi, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M : A, (N : B, \sigma + \{x \mapsto (M, \sigma)\}).\pi, \sigma, \tau\rangle
\end{array}$$

II.1.2. Success and error states

As we have already mentioned before, the machine will directly test if the judgement is correct when the judgement is really simple enough. Therefore, most of the rules here are quite obvious, but still necessary for the *LTKAM* to work.

$$\begin{array}{rcll}
\langle \mathtt{S}\ M : \mathbb{N}, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \langle M : \mathbb{N}, \pi_0, \sigma, \tau\rangle & \\
\langle \mathtt{S}\ M : T, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{error} & \text{when } T \neq \mathbb{N} \\
\langle \mathtt{Z} : \mathbb{N}, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{success} & \\
\langle \mathtt{T} : \mathbb{B}, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{success} & \\
\langle \mathtt{F} : \mathbb{B}, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{success} & \\
\langle \mathtt{T} : T, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{error} & \text{when } T \neq \mathbb{B} \\
\langle \mathtt{F} : T, \pi_0, \sigma, \tau\rangle & \overset{\Gamma}{\rightsquigarrow} & \text{error} & \text{when } T \neq \mathbb{B}
\end{array}$$

II.2. Implementation of Martin-Löf type theory's language

The first step is to define the different possible terms of the language.

We have added a few more constructs that aren't part of the language but will be used in the implementation. It may be either directly used in the calculus, with the case of `MissingInfo` which is the constructor for the _ symbol used in the rules described before, or the case of `Channel` that is the constructor for the channels c_Γ^T. Or it may also be used indirectly, for giving the judgement: `Colon A B` stands for $A : B$.

```
type VName = String
type CName = String

data Term = Var VName                          -- Variable
          | Abs VName Term                     -- Lambda-abstraction
          | App Term Term                      -- Application
          | CaseB Term Term Term               -- Boolean condition
          | T                                  -- True
          | F                                  -- False
          | CaseN Term Term Term               -- Natural condition
          | Z                                  -- Zero
          | S Term                             -- Successor
          | Pair VName Term Term               -- Pairs (a,b)
          | FirstPair Term                     -- (a,b)->a
          | SecondPair Term                    -- (a,b)->b
          | Y Term                             -- recursion
          | Bool                               -- Boolean Type
          | Nat                                -- Natural Type
          | ProdType Term Term                 -- name may be changed. Denotes \Pi a b
          | SumType Term Term                  -- name may be changed. Denotes \Sigma a b
          | U                                  -- Universe Type
          | CaseU Term Term Term Term Term     -- caseU
          | Channel CName Term Context         -- Channel
          | Colon Term Term                    -- t:T
          | MissingInfo                        -- _ (Probably a very temporary name)
```

Channels carry a type and a context. As in this type theory, types are specific terms, we will have to make a function testing whether the term is a type or not.

We still need to define contexts in order to have channels defined completely.

```
newtype Context = Context [(VName, Term)]
```

We may need to read (at least for debugging) the different terms the TKAM constructs while processing. We now write an instance of Show for both terms and contexts :

```
instance Show Term where
 show (Var v)           = v
 show (Abs v e)         = "" ++ v ++ "." ++ show e
 show (App t1 t2)       = "(" ++ show t1 ++ ")␣(" ++ show t2 ++ ")"
 show (CaseB b t e)     = "caseB␣(" ++ show b ++ ")␣("
                                     ++ show t ++ ")␣("
                                     ++ show e ++ ")"
 show T                 = "tt"
 show F                 = "ff"
 show (CaseN n z s)     = "caseN␣(" ++ show n ++ ")␣("
                                    ++ show z ++ ")␣("
                                    ++ show s ++ ")"
 show Z                 = "Z"
 show (S t)             = "S(" ++ show t ++ ")"
```

```
 show (Pair v a b)       = "Pair_" ++ v ++ "(" ++ show a ++ "," ++ show b ++ ")"
 show (FirstPair a)      = "fst(" ++ show a ++ ")"
 show (SecondPair a)     = "snd(" ++ show a ++ ")"
 show (Channel n t c)    = "[" ++ n ++ ":" ++ show t ++ "," ++ show c ++ "]"
 show Bool               = "Bool"
 show Nat                = "Nat"
 show U                  = "U"
 show (ProdType a b)     = "␣(" ++ show a ++ ")␣(" ++ show b ++ ")"
 show (SumType a b)      = "␣(" ++ show a ++ ")␣(" ++ show b ++ ")"
 show (CaseU t b n s p) = "caseU␣(" ++ show t ++ ")␣("
                                    ++ show b ++ ")␣("
                                    ++ show n ++ ")␣("
                                    ++ show s ++ ")␣("
                                    ++ show p ++ ")"
 show (Colon a b)        = "(" ++ show a ++")␣:␣(" ++ show b ++ ")"
 show (MissingInfo)      = "_"

instance Show Context where
 show (Context c) = case c of
   [] -> ""
   ls -> "{" ++ showMapT ls ++ "}"
  where showMapT :: [(VName, Term)] -> String
        showMapT []          = ""
        showMapT [(n,t)]     = n ++ ":" ++ show t
        showMapT ((n,t):cs) = n ++ ":" ++ show t ++ "," ++ showMapT cs
```

We define what a state of the TKAM is :

```
newtype MState = MState (Term, Stack, VEnv, CEnv, Int)
```

The state contains a term, a stack (of closures), a variable environment and a channel environment. We have added another parameter to the definition : it is an integer, which will be used to avoid reusing variable names.

We now need to define each component of a state :

```
newtype Stack = Stack [Closure]

newtype Closure = Closure (Term, VEnv)

newtype VEnv = VEnv [(VName, Closure)]

newtype CEnv = CEnv [(CName, Term)]
```

For debugging purposes (as reading a state should not be necessary when testing a judgement), we write another instances of Show for displaying a state :

```
instance Show MState where
 show (MState (t, s, lv, env, _)) =
   "(␣" ++ show t   ++ ",\n␣␣"
        ++ show s   ++ ",\n␣␣"
        ++ show lv  ++ ",\n␣␣"
        ++ show env ++ "␣)"

instance Show Stack where
 show (Stack s) = case s of
   []   -> ""
   c:cs -> show c ++ "." ++ show cs
```

```
instance Show Closure where
 show (Closure (t, lv)) = "(" ++ show t ++ ",␣" ++ show lv ++ ")"

instance Show VEnv where
 show (VEnv []) = ""
 show (VEnv ls) = "{" ++ showMapV ls ++ "}"
  where showMapV :: [(VName, Closure)] -> String
        showMapV []          = ""
        showMapV [(n,c)]     = n ++ "=" ++ show c
        showMapV ((n,c):ls) = n ++ "=" ++ show c ++ ",␣" ++ showMapV ls

instance Show CEnv where
 show (CEnv []) = ""
 show (CEnv ls) = "{" ++ showMapE ls ++ "}"
  where showMapE :: [(CName, Term)] -> String
        showMapE []          = ""
        showMapE [(n,t)]     = n ++ "=" ++ show t
        showMapE ((n,t):ls) = n ++ "=" ++ show t ++ ",␣" ++ showMapE ls
```

II.3. TRANSITION FUNCTION

We will now define the transition function of Martin-Löf TKAM. First, we need to define the different possible states :

```
data Status = Error -- There was an error
            | Inst  -- Need for a channel instantiation
            | Value -- Value state
            | Success -- If the judgment has been properly verified.
            | DebugNeed -- When the state reached can't be processed
           -- deriving ( Show )

instance Show Status where
  show (Error) = "Error"
  show (Inst) = "Instantiation␣required"
  show (Value) = "Value?"
  show (Success) = "Success␣!"
  show (DebugNeed) = "Debug␣needed."
```

The following functions will be needed to write the different rules: the rules only apply under specific conditions (for example, we may need to verify if both term and type are values, or if there is no use of the _ construct). This will be verified by these functions :

```
isNotMissingInfo :: Term -> Bool

isNotMissingInfo MissingInfo = False
isNotMissingInfo _ = True

isValue :: Term -> Bool
isValue (Abs _ _) = False
isValue (App _ _) = False
isValue (CaseB _ _ _) = False
isValue T = True
isValue F = True
isValue (CaseN _ _ _) = False
isValue Z = True
isValue (S t) = isValue t
isValue (Pair v a b) = isValue a && isValue b
```

```
isValue (FirstPair a) = isValue a
isValue (SecondPair a) = isValue a
isValue (Channel _ _ _) = False
isValue Bool = True
isValue Nat = True
isValue U = True -- Not sure in fact
isValue (SumType a b) = isValue a && isValue b
isValue (ProdType a b) = isValue a && isValue b
isValue (CaseU _ _ _ _ _) = False
isValue (Colon _ _) = False
isValue MissingInfo = False
isValue _ = False
```

The type of the transition function is the following :

```
step :: Context -> MState-> Either MState (Status,MState)
```

We use the Either datatype in the return type in order to distinguish between a continuation state (represented by a new state of the TKAM), or a final state (which may be Success, Error, or instantiation)

II.3.1. Calculation rules

Booleans and naturals

The following rule is a rule for boolean conditions. The computation will continue with the condition term, and the two other terms are added to the top of the stack (just as if the TKAM was working on another function, with two parameters)

```
step _ (MState (CaseB c t e, Stack st, VEnv lv, env, n)) =
  let cl = (Closure ((CaseB MissingInfo t e), VEnv lv))
  in Left $ MState (c, Stack (cl:st), VEnv lv, env, n)
```

We can only apply the rules for true and false if there are at least two closures on the stack.

```
step _ (MState (T, Stack ((Closure ((CaseB MissingInfo t _),VEnv lv)):s), _, env, i)) =
  Left $ MState (t, Stack s, VEnv lv, env, i)

step _ (MState (F, Stack ((Closure ((CaseB MissingInfo _ f),VEnv lv)):s), _, env, i)) =
  Left $ MState (f, Stack s, VEnv lv, env, i)
```

We're now working on the case construct over natural numbers, which is quite similar to boolean conditions.

```
step _ (MState (CaseN m z s, Stack st, VEnv lv, env, n)) =
  let cl = (Closure ((CaseN MissingInfo z s),VEnv lv))
  in Left $ MState (m, Stack (cl:st), VEnv lv, env, n)
```

The difference with the true and false cases is that, in the successor case, the element inside the successor construct must be applied to the corresponding element on the stack.

```
step _ (MState (Z, Stack ((Closure ((CaseN MissingInfo z _),VEnv lv)):s), _, env, i)) =
  Left $ MState (z, Stack s, VEnv lv, env, i)

step _ (MState (S t, Stack ((Closure ((CaseN MissingInfo _ s),VEnv lv)):st), _, env, i)) =
  Left $ MState (App s t, Stack st, VEnv lv, env, i)
```

II.3.2. Application and abstractions

For the application, if we are working on a simple term (and not a complete statement), we only need to add the argument term to the top of stack, and then continue the computation directly with the function.

```
step _ (MState (App f a, Stack s, lv, env, i)) =
  let cl = Closure (a, lv)
  in Left $ MState (f, Stack (cl:s), lv, env, i)
```

When working on terms, if we have a lambda abstraction and the stack is not empty, the variable in the lambda is mapped to the top of the stack in the variable environment, to be used later in the computation.

```
step _ (MState (Abs v e, Stack (cl:s), VEnv lv, env, i)) =
  Left $ MState (e, Stack s, VEnv ((v,cl):lv), env, i)
```

It is a bit more complicated when we are working on a full statement ($\lambda x.M : \Pi_{x:A}B$). We create a channel for generating an argument of the good type, and proceed with verifying if M is of the good type.

```
step _ (MState (Colon (Abs v m) (ProdType a b), Stack s, VEnv lv, env, i)) =
  let channel = Channel ("ch" ++ (show i)) a (Context [])
      cl = Closure (channel, VEnv lv)
      in Left $ MState (Colon m b, Stack s, VEnv ((v,cl):lv), env, i+1)
```

II.3.3. Recursion

In the case of the Y combinator, the term is added to the stack, and the computation resumes with the term inside the Y constructor.

```
step c (MState (Y t, Stack s, lv, env, n)) =
  let s' = Closure (Y t, lv):s
  in Left $ MState (t, Stack s', lv, env, n)
```

II.3.4. Pairs

As the first term of the pair is needed to be able to compute the second term, we will compute it first. The first rule is there to force the computation to continue on the first term when it is not a value, and the second rule will reconstruct the pair once we've computed the first pair.

```
step _ (MState (Pair v a b, Stack s, lv, env, i)) |
(isNotMissingInfo a) && (not (isValue a)) =
  let cl = Closure ((Pair v MissingInfo b), lv)
  in Left $ MState (a, (Stack (cl:s)), lv, env, i)

step _ (MState (t, Stack ((Closure ((Pair v MissingInfo a), lv)):s), lvtwo, env, i)) |
isValue t =
  Left $ MState ((Pair v t a), Stack s, lv, env, i)
```

The following rules compute the projections : we put the projection in the stack to be processed once we find a Pair in the computation.

```
step _ (MState (FirstPair t, Stack s, lv, env, i)) =
  let cl = Closure (FirstPair MissingInfo,lv)
  in Left $ MState (t, Stack (cl:s), lv, env, i)
```

```
step _ (MState (Pair v a b, Stack (Closure ((FirstPair MissingInfo), lv):s), lvtwo, env, i)) =
  Left $ MState (a, Stack s, lv, env, i)

step _ (MState (SecondPair t, Stack s, lv, env, i)) =
  let cl = Closure (SecondPair MissingInfo,lv)
  in Left $ MState (t, Stack (cl:s), lv, env, i)

step _ (MState (Pair v a b, Stack (Closure ((SecondPair MissingInfo), VEnv lv):s),
lvtwo, env, i)) =
  let cln = Closure (a, VEnv lv)
  in Left $ MState (b, Stack s, VEnv ((v,cln):lv), env, i)
```

The last rule we need about pairs is the rule to be able to compute type-checking on pairs : we first need to check if the first argument is of the correct type, and then need to check if the second argument is of the correct type when the variable is replaced by the first argument.

```
step _ (MState (Colon (Pair v a b) (SumType ta tb), Stack s, VEnv lv, env, i)) =
  let cln = Closure (a, VEnv lv)
  in let cl = Closure (Colon b tb, VEnv ((v,cln):lv))
     in Left $ MState (Colon a ta, Stack (cl:s), VEnv lv, env, i)
```

II.3.5. Variables and channels

For variables, if the variable is mapped to something in the variable environment, the computation resumes with this closre.

```
step _ (MState (Var v, s, VEnv lv, env, i)) | v `isIn` lv =
  let Closure cl = fromJust $ lookup v lv -- safe since v is in lv
  in Left $ MState (fst cl, s, snd cl, env, i)
 where isIn :: VName -> [(VName, Closure)] -> Bool
       isIn v lv = case lookup v lv of
                     Nothing -> False
                     _       -> True
```

However, if the variable is not mapped to something, then we map it to a new channel of the right type in the context and continue the computation with this channel. As the context should give the type of any variable used in the lambda-term, if the variable isn't mapped to anything, there is an error.

```
step ctx@(Context c) st@(MState (Var v, s, VEnv lv, env, i)) = -- v not in lv
  case lookup v c of
    Nothing -> Right (Error,st)
    Just tv -> let channel = Channel ("ch" ++ (show i)) tv ctx
                   cl = Closure (channel, VEnv lv)
               in Left $ MState (channel, s, VEnv ((v,cl):lv), env, i+1)
```

II.3.6. Statement simplification rules

If either the term or the type is not a value yet, we continue the calculation by only working on the type (if both are not values), or the value.

```
step _ (MState (Colon a b, Stack s, lv, env, i)) |
(isNotMissingInfo b) && (not (isValue a)) && (not (isValue b)) =
  let cl = Closure ((Colon a MissingInfo), lv)
  in Left $ MState (b, (Stack (cl:s)), lv, env, i)
```

```
step _ (MState (Colon a b, Stack s, lv, env, i)) | (isNotMissingInfo a) && (not (isValue a)) =
  let cl = Closure ((Colon MissingInfo b), lv)
  in Left $ MState (a, (Stack (cl:s)), lv, env, i)
```

In some cases, both term and type are filled by values but they can't be tested directly. For example, $S(t)$ is a value when t is. The following rules force a deeper calculation :

```
step _ (MState (S t, Stack s, lv, env, i)) | not (isValue t) =
  let cl = Closure ((S MissingInfo), lv)
  in Left $ MState (t, Stack (cl:s), lv, env, i)

step _ (MState (t, Stack ((Closure ((S MissingInfo),lv)):s), lvtwo, env, i)) | isValue t =
  Left $ MState (S t, Stack s, lv, env, i)
```

II.3.7. Already instantiated channels

If there is a channel on top of the stack, then if it has been instantiated already, we replace it with the corresponding term in the channel environment (we have to do that to keep the language pure).

```
step _ (MState (Channel c _ _, s, lv, CEnv env, i)) | c `isIn` env =
  let t = fromJust $ lookup c env -- safe since c is in env
  in Left $ MState (t, s, lv, CEnv env, i)
 where isIn :: CName -> [(CName,Term)] -> Bool
       isIn c lc = case lookup c lc of
                     Nothing -> False
                     _       -> True
```

If the channel has not been instantiated yet, then it must be instantiated.

```
step _ st@(MState (Channel c _ _, _, _, _, _)) = -- c not in env
  Right (Inst,st)
```

II.3.8. Type-checking rules

The following rules are checking types, and returning success if they're successfully checked :

```
step _ (MState ((Colon (S term) Nat), s, lv, env, i)) | isNotMissingInfo term =
  Left $ MState (Colon term Nat, s, lv, env, i)

step _ st@(MState ((Colon (S term) _), s, lv, env, i)) = Right (Error, st)
-- The successor function may only return a natural
step _ st@(MState ((Colon (Z) Nat), Stack [], lv, env, i)) = Right (Success,st)
step _ st@(MState ((Colon (Z) _), Stack [], lv, env, i)) = Right (Error, st)
step _ st@(MState ((Colon (T) Bool), Stack [], lv, env, i)) = Right (Success, st)
step _ st@(MState ((Colon (F) Bool), Stack [], lv, env, i)) = Right (Success, st)
step _ st@(MState ((Colon (T) _), Stack [], lv, env, i)) = Right (Error, st)
step _ st@(MState ((Colon (F) _), Stack [], lv, env, i)) = Right (Error, st)
```

Also, in some cases we want to verify other judgements (for example, when we're testing a SumType). The following rules are basically the same, except they'll make the computation continue on the following judgement in the stack :

```
step _ mast@(MState ((Colon (Z) Nat), Stack ((Closure ((Colon a b), VEnv lv)):st),
_, env, i)) =
  Left $ MState (Colon a b, Stack st, VEnv lv, env, i)
```

```
step _ mast@(MState ((Colon (T) Bool), Stack ((Closure ((Colon a b), VEnv lv)):st),
_, env, i)) =
  Left $ MState (Colon a b, Stack st, VEnv lv, env, i)

step _ mast@(MState ((Colon (F) Bool), Stack ((Closure ((Colon a b), VEnv lv)):st),
_, env, i)) =
  Left $ MState (Colon a b, Stack st, VEnv lv, env, i)

step _ mast@(MState ((Colon (T) _), Stack ((Closure ((Colon a b), VEnv lv)):st), _, env, i)) =
  Right (Error, mast)

step _ mast@(MState ((Colon (F) _), Stack ((Closure ((Colon a b), VEnv lv)):st), _, env, i)) =
  Right (Error, mast)
```

II.3.9. Forming back $M : T$ statements

When the terms have been computed into a value, we reconstruct the complete judgement.

```
step _ (MState (t, Stack ((Closure ((Colon a MissingInfo), lv)):s),
lvtwo, env, i)) | isValue t =
  Left $ MState ((Colon a t), Stack s, lv, env, i)

step _ (MState (t, Stack ((Closure ((Colon MissingInfo a), lv)):s),
lvtwo, env, i)) | isValue t =
  Left $ MState ((Colon t a), Stack s, lv, env, i)
```

Some of the rules for typechecking require a bit more attention, and require the TKAM to compute a specific term :

```
step _ (MState ((Colon (CaseB c t e) ty, Stack [], lv, env, i))) =
  let cl = Closure ((Colon MissingInfo ty), lv)
  in Left $ MState (CaseB c t e, (Stack ([cl])), lv, env, i)
```

If we happen (and we hope not) to have a case that is not handled by any of the previous rules, we return a `DebugNeed` state. We may use the entire state of the KAM for debugging :

```
step _ msta = Right (DebugNeed, msta)
```

II.3.10. The transition function

The following function will make a full transition between two steps of the run of the TKAM :

```
steps :: Context -> MState -> (Status,MState)
steps c m = case step c m of
              Left m' -> steps c m'
              Right r -> r
```

II.4. Atomic Normal Forms

At some point of the calculation, we may need to generate atomic normal forms of a certain type, and a certain context. The function generating these ANF will have the following type :

```
atomicNFs :: Term          -- Type of the ANF (which is a term)
          -> Context       -- Context
          -> Int           -- Next index for a fresh name
          -> [(Int,Term)] -- Returns couples next fresh name / term
```

The integer is used to generate new fresh variable and channel names. As we see in the return type, the atomic normal form that has been created this way is paired with an integer which can be used to have fresh variable names for the TKAM computation.

As the first parameter is the type of the ANF, we are going to pattern-match this parameter when we generate ANFs. If the type is a natural, then either it is zero, or it is the successor of a natural. (Notice that we don't generate infinite atomic normal forms since Haskell implements lazy computation).

```
atomicNFs Nat ctx@(Context c) i =
  let rest = map (adaptVarType ctx Nat i) c
      sch = S (Channel ("ch" ++ (show i)) Nat ctx)
  in (i,Z):(i+1,sch):rest
```

For the booleans, there are two simple ANFs (True and False).

```
atomicNFs Nat ctx@(Context c) i =
  let rest = map (adaptVarType ctx Bool i) c
  in (i,T):(i,F):rest
```

You may have noticed a function called `adaptVarType` was used in the two previous cases. It is used to make atomic normal forms that are more complicated, specifically using the $\text{case}_{\mathbb{B}}$ and $\text{case}_{\mathbb{N}}$ functions :

```
adaptVarType :: Context -> Term -> Int -> (VName,Term) -> (Int,Term)
adaptVarType ctx tc i (v,tv) =
  let ts = components tv -- Types to apply to v to get a ground type
      cn = [ "ch" ++ show n | n <- [i..(i + length ts - 1)] ]
      ch = zipWith (\n t -> Channel n t ctx) cn ts -- Channels to apply to v
      t = foldl App (Var v) ch -- Term : application of the channels to v
      i' = i + length ts
      ch1 = "ch" ++ (show i')
      ch2 = "ch" ++ (show (i'+1))
      term = case baseType tv of -- Patern-match on the type of t
               Bool -> CaseB t (Channel ch1 tc ctx) (Channel ch2 tc ctx)
               Nat  -> CaseN t (Channel ch1 tc ctx) (Channel ch2 (ProdType Nat tc) ctx)
  in (i' + 2, term)
 where components :: Term -> [Term]
       components (ProdType ta tb) = ta : components tb
       components _           = []
       baseType :: Term -> Term
       baseType (ProdType _ t) = baseType t
       baseType t         = t
```

Now we have a function that can compute all ANFs of a certain type, in a certain context, we need a function that picks one at random (so we can use this one to continue the computation).

```
atomicNF :: StdGen              -- Random seed
         -> Term                -- Type of the ANF (which is a term)
         -> Context             -- Context
         -> Int                 -- Next index for a fresh name
         -> (StdGen,(Int,Term)) -- New seed, new int for fresh name, term
atomicNF g t c i = oneOf g $ atomicNFs t c i
 where oneOf :: StdGen -> [a] -> (StdGen,a)
       oneOf g l = let sz = length l
```

```
                    (i,g') = next g
                    i' = i `mod` sz
                    i'' = if i' < 0 then i' + sz else i'
                in (g',l !! i'')
```

This following function instantiates a channel in a machine state. We can only use this function if the term in the state is a channel.

```
instantiateChannel :: StdGen           -- Random seed
                   -> MState           -- State
                   -> (StdGen,MState) -- New random seed and new state
instantiateChannel g (MState (Channel n t c, s, lv, CEnv env, i)) =
  let (g', (i', anf)) = atomicNF g t c i
      env' = CEnv ((n, anf) : env)
  in (g', MState (anf, s, lv, env', i'))
```

II.5. Testing

Now we have functions for computing the TKAM, making channel instantiations and do most of what is needed to test judgements. The last function we need is a function to actually start the computation, in order to be able to test a judgement.

```
test :: StdGen                  -- Random seed
     -> Context -> Term         -- Judgement
     -> Int                     -- Int for fresh variable names
     -> (StdGen,Bool)           -- New random seed and result
```

The following function will start the computation with a state only containing the judgement. The testing is already done by the TKAM, the only things we need to take care of are the instantiation states : in the case the TKAM computation returns a instantiation state, we run `instantiateChannel` on it so we can continue the computation.

```
test g c t _ = let st = MState (t, Stack [], VEnv [], CEnv [], 0)
  in loop g st c
  where loop :: StdGen -> MState -> Context -> (StdGen,Bool)
        loop g st c = let (k, st') = steps c st
                      in case k of
                           Error -> (g,False)
                           Success -> (g,True)
                           Value -> (g,False)
                           Inst -> let (g',st'') = instantiateChannel g st'
                                   in loop g' st'' c
                           DebugNeed -> (g,False)
```

II.6. Convenient functions for testing

When we are testing a judgement, testing with one specific instantiation will most of the time not be enough. The following version takes as a parameter a number of tests to perform, and will return the number of success before first failure (or 0 if there is no failure)

```
runTests :: StdGen                  -- Seed for random generation
         -> Int                     -- Number of distinct tests to run
         -> Context -> Term         -- Judgement
         -> Int                     -- Nb of success before first failure.
```

```
runTests g 0  c t = 0
runTests g nb c t = let (g',r) = test g c t 0
                    in if r then 1 + runTests g' (nb-1) c t
                            else 0
```

The following function will be more convenient for testing – we are using the IO monad here so we can have a random seed for the environment (which will be used when instantiating a channel)

```
quickTest :: Int                    -- Number of runs
          -> Context -> Term        -- Judgement
          -> IO ()                  -- IO monad for random seed generation.
quickTest nb c t = do
  g <- newStdGen
  let r = runTests g nb c t
  if r == nb
     then putStrLn $ "All the " ++ show nb ++ " tests passed!"
     else putStrLn $ "Test number " ++ show (r+1) ++ " failed ..."
```

II.7. Convenient functions for debugging

The following functions have been made to simplify the debugging of the implementation, and can be used to have a detailed list of steps in the computation. They are not really different from the testing functions that we have explained before : we are only adding print functions here and there to display the steps.

```
stepsIO :: Context -> MState -> IO () -- Status, MState
stepsIO c m = do case step c m of
                   Left m' -> do print m'
                                 stepsIO c m'
                   Right r -> do print r

testIO :: Context -> Term
       -> Int
       -> IO ()
testIO c t _ = let st = MState (t, Stack [], VEnv [], CEnv [], 0)
  in loop st c
  where loop :: MState -> Context -> IO ()
        loop st c = stepsIO c st


testGR :: StdGen                  -- Random seed
     -> Context -> Term           -- Judgement
     -> Int                       -- Int for fresh variable names
     -> IO ()                     -- mstate

testGR g c t _ = let st = MState (t, Stack [], VEnv [], CEnv [], 0)
  in loop g st c
  where loop :: StdGen -> MState -> Context -> IO ()
        loop g st c = let (k, st') = steps c st
                      in case k of
                           Inst -> let (g',st'') = instantiateChannel g st'
                                   in do print st''
                                         loop g' st'' c
                           _ -> print st'

quickTestGR :: Context -> Term          -- Judgement
            -> IO ()                    -- IO monad for random seed generation.
quickTestGR c t = do
```

```
  g <- newStdGen
  testGR g c t 0
  -- putStrLn $ "Test has given the following : " ++ show (testGR g c t 0)
```

II.8. EXAMPLES

Here are simple examples, that we use to check if the judgement testing is done correctly :

```
c1 = Context []
run1 = quickTest 1 c1 (Colon (S Z) Nat)
run2 = quickTest 1 c1 (Colon (S F) Nat)
run3 = quickTest 1 c1 (Colon (S F) Bool)
run4 = quickTest 1 c1 (Colon (F) Nat)
run5 = quickTest 1 c1 (Colon (F) Bool)
run6 = quickTest 1 c1 (Colon (CaseB F T (S (S Z))) Nat)
run7 = quickTest 1 c1 (Colon (CaseB F T F) Bool)
run8 = testIO c1 (Colon (CaseB F T F) Bool) 0
run9 = isValue F
run10 = testIO c1 (Colon (CaseB F T (S (S Z))) Nat) 0
run11 = isValue (S (S Z))
run12 = testIO c1 (Colon (App (Abs "x" (S (Var "x"))) (Z)) Nat) 0
run13 = testIO c1 (Colon ((Abs "x" (S (Var "x")))) (ProdType Nat Nat)) 0
run14 = quickTest 100 c1 (Colon ((Abs "x" (S (Var "x")))) (ProdType Nat Nat))
run15 = quickTestGR c1 (Colon ((Abs "x" (S (Var "x")))) (ProdType Nat Nat))
run16 = quickTest 1 c1 (Colon (Pair "x" T (Var "x")) (SumType Bool Bool))
run17 = testIO c1 (Colon (Pair "x" T (Var "x")) (SumType Bool Bool))
run18 = quickTest 1 c1 (Colon (Pair "x" T T) (SumType Bool Bool))
run19 = testIO c1 (Colon (Pair "x" T T) (SumType Bool Bool))
run20 = quickTestGR c1 (Colon (Pair "x" T T) (SumType Bool Bool))
run21 = quickTestGR c1 (Colon (Pair "x" T (Var "x")) (SumType Bool Bool))
run22 = quickTest 1 c1 (Colon (SecondPair (Pair "x" T (Var "x"))) Bool)
run23 = quickTest 1 c1 (Colon (FirstPair (Pair "x" T (Var "x"))) Bool)
```

Conclusion

We have presented an implementation of a framework used to test judgements of Martin-Löf type theory, based on aspects of the work of Peter Dybjer [1] and Pierre Clairambault. However, there is still a bit of work left as, for some types, the machine implemented in this report can't generate atomic normal terms.

This kind of testing may be helpful for the development of proof assistants : it can be used to find bugs, or incorrect types, and to help debug these.

Nonetheless, this internship helped me to have a better understanding of how "more complex" type theories work, especially Martin-Löf type theory that uses dependent types that are constructed out of other types.

Acknowledgements

First of all, I would like to thank Peter Dybjer for giving me the opportunity to do this internship, and for the very helpful and interesting talks I had with him throughout the internship.

I would also like to thank Cyril Cohen for the help he has given – having someone to talk to in French in the same subject was very helpful sometimes.

I'd also want to thank Nicolas Markey for helping me find this opportunity to do this really interesting internship.

Finally, I would like to thank people at Chalmers for making this a very nice internship; especially people of the research group I was in.

References

[1] Peter Dybjer. Program testing and the meaning explanations of intuitionistic type theory. In Peter Dybjer, Sten Lindström, Erik Palmgren, and Göran Sundholm, editors, *Epistemology versus Ontology*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 215–241. Springer, 2012.

[2] Rodolphe Lepigre. Testing judgements of type theory. Internship report, Chalmers and Université de Savoie, Chambéry, 2012.

[3] Per Martin-Löf. An intuitionistic theory of types. 1972.

[4] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.