Internship report

Testing judgements of type theory

Chalmers Tekniska Högskola, Göteborg

Rodolphe Lepigre
Université de Savoie, Chambéry
rodolphe.lepigre@etu.univ-savoie.fr

Under the supervision of Peter Dybjer
peterd@chalmers.se

August 23, 2012

Abstract

We investigate a testing framework for type theory. We first describe the *Krivine Abstract Machine* (*KAM*), and the *Testing KAM* (*TKAM*) which is a modified version of the *KAM* that allows the testing of terms of the *PCF* language. This follows notes by Pierre Clairambault ([1]).

We use two versions of the *TKAM*. The first one is restricted to the *Finite System T* language and the second one has lazy natural numbers.

We then use the virtual machines as the central part of a testing procedure for the typing of terms.

This work is an implementation of certain aspects of the testing manual for intuitionistic type theory described in Peter Dybjer's paper *Program Testing and Constructive Validity* [2].

Keywords: lazy instantiation / generation, transition function, testing the typing, implementation.

Acknowledgements

First of all I would like to thank Peter Dybjer for accepting me for this internship and for his help throughout the summer. It has been a very worthwhile experience which helped me confirm my interest in the field of type theory.

I would also like to thank all my other teachers at Chalmers for their interesting and valuable lessons which allowed me to really recognise the areas that are of the greatest interest for me.

I also wish to thank Pierre Hyvernat, my tutor for this project, for all his help and encouragement during my studies.

Thank you also to Laurence Vignollet for encouraging me to spend this year abroad, and for allowing me to choose a research internship rather than one in industry.

Finally I would like to thank all my teachers at the Université de Savoie for their help and support throughout my university career.

Contents

Introduction

Program testing is one way of checking that a program meets its specifications. There exists a few frameworks that allow testing with automatic input generation such as *QuickCheck* [3] or *SmallCheck* [4] for the language Haskell.

These tools allow the testing of predicates over the application of generated input to a function. The input may be generated in different ways, *QuickCheck* uses random generation, and *SmallCheck* generates the input in a systematic way.

The same kind of tool have been investigated for dependent types by Qiao Haiyan [5]. It is shown that testing might be used as a tool during the construction of a proof.

What we mean by testing is quite different from what has been described in *QuickCheck*-like programs. Our approach has been presented as a testing manual for intuitionistic type theory by Peter Dybjer in [2]. It is about testing the following form of judgement,

$$\Gamma \vdash a : T$$

(the term a has type T in the context Γ) as opposed to the testing of a predicate

$$\Gamma \vdash \mathcal{P}(a)$$

(the predicate $\mathcal{P}(a)$ is true in context Γ).

Content of the report. In the first section we present the *Krivine Abstract Machine* (*KAM*) together with one of its variations, the *Testing KAM*, described by Pierre Clairambault in [1]. We give two versions of the *TKAM*, the first one is restricted to the *Finite System T* language, and the second one is similar to the original *TKAM* (designed for *PCF*), but we extend it with lazy natural numbers.

The second section contains the definition of a testing framework for *Finite System T*, together with its implementation.

In the third section the testing framework and implementation are extended to the *PCF* language.

A full description of the implementations can be found in the appendix, in the form of a *Literate Haskell* documentation.

1 Preliminaries

1.1 Krivine Abstract Machine and λ-calculus

In this section we consider the very simple language of the λ-calculus. A term can be either a variable, a λ-expression or an application.

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

An environment is a function σ that maps variables to closures, and closures are couples of a term and an environment.

The *Krivine Abstract Machine* (*KAM*) also contains a stack, which is simply a list of closures.

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

The *KAM*'s state is a triple of a term, an environment and a stack. And the computation is defined in term of transitions. When there is no possible transition, the machine stops, and the state becomes the final state. The transition function $\rightarrow$ is defined by the following.

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

We will now try to run an example term using our definition of the *KAM*. We consider the term $(\lambda x.x\ x)\ (\lambda x.x)$. The starting state of the machine is the following.

$$\langle (\lambda x.x\ x)\ (\lambda x.x),\ \pi_0,\ \emptyset\rangle$$

The following is a full computation starting from this state.

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

The term in the final state is the identity function $\lambda x.x$, as it was to be expected.

1.2 Finite System T

We introduce a simple typed language based on the λ-calculus. The language has a single atomic type which is *Bool*, and a function type.

$$T\ ::=\ \text{Bool}\ \mid\ T \rightarrow T$$

The terms of the language are the following, including the constants *true* and *false*.

$$t\ ::=\ x\ \mid\ \lambda x.t\ \mid\ t\ t\ \mid\ \mathtt{T}\ \mid\ \mathtt{F}\ \mid\ \text{case}_{\mathtt{B}}\ t\ t\ t$$

The typing rules are the usual ones.

1.2.1 The Testing KAM restricted to Finite System T

We need to extend the language with an additional term, namely channels. Channels do not bellong to the language, they are metaterms. They play the role of metavariables, and will be eventually replaced by generated expressions during computation. A channel c always carry a type T and a context Γ. We write c_Γ^T.

The *TKAM* is a virtual machine, based on the *KAM*. Its state is a quadruple $\langle t\ , \pi,\ \sigma,\ \tau\rangle$ where $(t,\ \sigma)$ is a closure, π is a stack of closures and τ is a partial function that maps channels to *atomic normal forms* (which will be defined in the next section). This environment allows the machine to keep track of the instantiation of channels, and keep the language pure. The empty stack is denoted π_0.

The transition function $\overset{\Gamma}{\rightsquigarrow}$ for the TKAM is exhibited bellow. Note that it depends on a context Γ.

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

When a state of the following form is reached, there is a need for a channel instantiation.

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

An *atomic normal form* M of type A in context Γ is generated. Then the computation resumes with the following state.

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

Of course, there must be no collision between the names of the channels used in M and the channels in $\mathrm{dom}(\tau)$.

1.2.2 Atomic normal forms in Finite System T

An *Atomic normal form* always depends on a type T and a context Γ, we denote it $\mathrm{anf}_\Gamma(T)$.

There are two cases. Either T has the canonical form $\overrightarrow{A} \to \mathrm{Bool}$, and in that case we define

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

or $T = \mathrm{Bool}$, and we define

$$\begin{array}{rcl}
\mathrm{anf}_\Gamma(\mathrm{Bool}) & = & \{\mathtt{T},\mathtt{F}\} \ \cup \ \{\mathrm{case}_{\mathtt{B}}\ (x\ \overrightarrow{c})\ c_{\mathtt{T}}\ c_{\mathtt{F}} \mid \\
 & & \qquad x : \overrightarrow{A} \to \mathrm{Bool} \in \Gamma, \\
 & & \qquad c_i \in C_\Gamma^{A_i}, \\
 & & \qquad c_{\mathtt{T}}, c_{\mathtt{F}} \in C_\Gamma^{\mathrm{Bool}}\}
\end{array}$$

where C_Γ^A is the set of channels of type A in context Γ.

1.3 PCF

PCF is a more expressive language, it is basicaly Finite System T extended with natural numbers and a fixed point combinator.

The types are now the following.

$$T \ ::= \ \text{Bool} \ | \ \text{Nat} \ | \ T \to T$$

And the terms are the following.

$$t \ ::= \ x \ | \ \lambda x.t \ | \ t\ t \ | \ \mathtt{Y}\, t \ | \ \mathtt{T} \ | \ \mathtt{F} \ | \ \text{case}_{\mathtt{B}}\ t\ t\ t \ | \ \mathtt{Z} \ | \ \mathtt{S}\, t \ | \ \text{case}_{\mathtt{N}}\ t\ t\ t$$

Note that it is possible to define iszero, pred or natrec using the core language only.

$$\begin{aligned}
\text{iszero} &\overset{\text{def}}{=} \lambda t.\text{case}_{\mathtt{N}}\ t\ \mathtt{T}\ (\lambda x.\mathtt{F}) \\
\text{pred} &\overset{\text{def}}{=} \lambda t.\text{case}_{\mathtt{N}}\ t\ \mathtt{Z}\ (\lambda x.x) \\
\text{natrec} &\overset{\text{def}}{=} \mathtt{Y}\ (\lambda rtgn.\text{case}_{\mathtt{N}}\ n\ t\ (\lambda p.g\ p\ (r\ t\ g\ p)))
\end{aligned}$$

1.3.1 The Lazy TKAM

The *TKAM* that we present here is a little bit different from the original version defined by Pierre Clairambault in [1]. However the aim of the *Lazy TKAM* is still to computes open expressions of the *PCF* language by introducing metavariables in the expression (called channels). And when a value is needed for a channel, a so-called *atomic normal form* is generated.

We need to extend the language with channels (c_Γ^T) in the same way as in the previous section. Once again, channels are metaterms. They are introduced during computation and replaced by special terms called *atomic normal forms.*

Much like the *TKAM*, the *Lazy TKAM*'s state is a quadruple $\langle t\ ,\pi,\ \sigma,\ \tau\rangle$ where $(t,\ \sigma)$ is a closure, π is a stack of closures and τ is a partial function that maps channels to *atomic normal forms.*

The transition function of the machine is defined as follow.

$$\begin{aligned}
\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 \text{case}_{\mathtt{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 \text{case}_{\mathtt{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\ \ c \in \text{dom}(\tau) &\overset{\Gamma}{\rightsquigarrow} \langle \tau(c),\ \pi,\ \sigma,\ \tau\rangle
\end{aligned}$$

As for the *TKAM* in the previous section, when a state of the following form is reached, there is a need for a channel instantiation.

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

An *atomic normal form* M of type A in context Γ is generated. Then the computation resumes with the following state.

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

1.3.2 Atomic normal forms in PCF

The definition of an *atomic normal form* of type T in context Γ for *PCF* is split into three cases.

First, when T has the canonical form $\overrightarrow{A} \to G$ where $G \in \{\mathrm{Nat},\ \mathrm{Bool}\}$ we define

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

When $T = \mathrm{Bool}$ we have the following.

$$\begin{aligned}\mathrm{anf}_\Gamma(\mathrm{Bool}) \ = \ \{\mathtt{T}, \mathtt{F}\} \ &\cup \ \{\mathrm{case}_{\mathtt{B}}\ (x\ \overrightarrow{c})\ d\ e \mid x : \overrightarrow{A} \to \mathrm{Bool} \in \Gamma,\\ &\qquad c_i \in C_\Gamma^{A_i},\ d, e \in C_\Gamma^{\mathrm{Bool}}\}\\ &\cup \ \{\mathrm{case}_{\mathtt{N}}\ (x\ \overrightarrow{c})\ d\ e \mid x : \overrightarrow{A} \to \mathrm{Nat} \in \Gamma,\\ &\qquad c_i \in C_\Gamma^{A_i},\ d \in C_\Gamma^{\mathrm{Bool}},\ e \in C_\Gamma^{\mathrm{Nat}\to\mathrm{Bool}}\}\end{aligned}$$

When $T = \mathrm{Nat}$ we have the following.

$$\begin{aligned}\mathrm{anf}_\Gamma(\mathrm{Nat}) \ = \ \{\mathtt{Z}\} \ &\cup \ \{\mathtt{S}\ c \mid c \in C_\Gamma^{\mathrm{Nat}}\}\\ &\cup \ \{\mathrm{case}_{\mathtt{N}}\ (x\ \overrightarrow{c})\ d\ e \mid x : \overrightarrow{A} \to \mathrm{Nat} \in \Gamma,\\ &\qquad c_i \in C_\Gamma^{A_i},\ d \in C_\Gamma^{\mathrm{Nat}},\ e \in C_\Gamma^{\mathrm{Nat}\to\mathrm{Nat}}\}\\ &\cup \ \{\mathrm{case}_{\mathtt{B}}\ (x\ \overrightarrow{c})\ d\ e \mid x : \overrightarrow{A} \to \mathrm{Bool} \in \Gamma,\\ &\qquad c_i \in C_\Gamma^{A_i},\ d, e \in C_\Gamma^{\mathrm{Nat}}\}\end{aligned}$$

2 Testing Judgements of Finite System T

We have not done any testing so far. We only gave a definition of the *TKAM*, together with *atomic normal forms* of a given type, in a given context.

We are now going to present what it mean to test a judgement in *Finite System T*, but first we need to change a little bit the *TKAM*, so that we can use it as the central part of our testing framework. We will then show an implementation of the testing manual.

2.1 Using the TKAM to test judgements

The *TKAM* is going to be used for testing judgements of the following form.

$$\Gamma \vdash a : T$$

Remark that testing the following judgements is equivalent.

$$\Gamma \vdash f : T_1 \to T_2 \quad \Gamma, x : T_1 \vdash f\ x : T_2$$

Hence it is easy to reduce any judgement to one of the following form.

$$\Gamma \vdash a : \text{Bool}$$

We now need to make some changes to the transition function of the *TKAM*. In particular, we will need it to yield different kinds of states, that we call

- *continuation* states,
- *value* states,
- *error* states
- and *instantiation* states.

First we give the transition rules that lead to a *continuation* state. This mean that their result can be used as a starting point for a new transition.

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

The second group of transition rules is the group of rules giving *value* states as output. The *value* states are in one sense the end of the computation.

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

The third group is the group of transitions yielding an *error* state. This simply mean that such transition leads to failure.

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

The last group contains only one transition rule which leads to a state that expresses the need for the instantiation of a channel. This is an *instantiation* state.

$$\langle c,\ \pi,\ \sigma,\ \tau\rangle\ \ c \notin \text{dom}(\tau) \overset{\Gamma}{\rightsquigarrow} \text{need instantiation}$$

Now to test the following judgement

$$\Gamma \vdash M : \text{Bool}$$

we run the *TKAM* on M, which mean fully evaluating the following state.

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

Remark that what we mean by fully evaluating is to take a transition steps until the state is not a *continuation* state.

If the result state is an *error* state, then the test fails. If the result state is a *value* state that is `T` or `F` then the test succeeds. If it is a *value* state that is a λ-abstraction then the test fails (the result has a function type, it cannot be a boolean).

If on the other hand the result state contains an instantiation request, then we instantiate the channel with an *atomic normal form*. We then run the *TKAM* again on this state, act according to the result, and so on.

Note that there is no guarantee for the computation to stop, in which case we cannot conclude on the issue of the test.

2.2 Implementation

In this section we show the correspondance between the mathematical description and the implementation of the *TKAM*. We first present the data structures that are going to be used, then the transition function, the implementation of the *atomic normal forms* and finally the testing functions

2.2.1 Data structures

The *TKAM* can be modeled easily by using algebraic data types. First we define the data type *Term* which will be used to represent a term of the *Finite System T* language.

```
data Term = Var VName
          | Abs VName Term
          | App Term Term
          | T
          | F
          | CaseB Term Term Term
          | Channel CName Type Context
```

Note that *VName* and *CName* are defined to be type synonyms of *String*. This allows us to distinguish name of variables and name of channels.

Channels need to carry a type and a context, which are defined as follow.

```
data Type = Bool
          | Fun Type Type

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

Now that we have terms, we need to represent the central part of the virtual machine, namely the state. We define the type *State* as follow.

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

In order to have defined the state completely, we need to define the type *Stack*, which is a stack of closures, represented as a list. *VEnv* maps variable names to closures and *CEnv* maps channel names to terms. Finally, a *Closure* is a couple *(Term, VEnv)*.

```
newtype Stack = Stack [Closure]

newtype VEnv = VEnv [(VName, Closure)]

newtype CEnv = CEnv [(CName, Term)]

newType Closure = Closure (Term, VEnv)
```

Note that the *Int* in the type *State* is used to have fresh names for channels and variables. It could be possible to add an *Int* in the definition of *VEnv* and *CEnv*, but this is a design choice. Yet an other possibility would be not to carry any *Int*, list all names used in a state and find one that is not used.

2.2.2 Transition function

The *TKAM* being defined by its transition function $\overset{\Gamma}{\rightsquigarrow}$, we need to define a function to represent it. This function will take as input a state s of the machine and return a state t such that $s \rightsquigarrow t$. We will also need the output value to include information about the kind of the output state. The function *step* has the following type signature.

```
step :: Context -> State -> Either State (Kind,State)
```

In the return type we use the *Either* data type to distinguish between *continuation* states and the other kinds of states on which the transition must stop or pause. We define the type *Kind* as follow.

```
data Kind = Error
          | Inst
          | Value
```

Before defining the *step* function, we can define easily the *steps* function, which will take transition steps until it reach a state that is not a *continuation* state.

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

The definition of *step* consists of a patern-matching on the input state, according to the definition of the transition function. First we give the rules related to conditional operator $\text{case}_{\mathbb{B}}$ and the boolean constants.

```
step _ (MState (CaseB c t e, Stack s, lv, env, i)) =
  let ct = Closure (t, lv)
      cf = Closure (e, lv)
  in Left $ MState (c, Stack (ct:cf:s), lv, env, i)

step _ (MState (T, Stack (Closure ct:_:s), _, env, i)) =
  Left $ MState (fst ct, Stack s, snd ct, env, i)
step _ (MState (F, Stack (_:Closure cf:s), _, env, i)) =
  Left $ MState (fst cf, Stack s, snd cf, env, i)
```

```
step _ st@(MState (T, Stack [], _, _, _)) = Right (Value,st)
step _ st@(MState (F, Stack [], _, _, _)) = Right (Value,st)

step _ st@(MState (T, Stack (_:[]), _, _, _)) = Right (Error,st)
step _ st@(MState (F, Stack (_:[]), _, _, _)) = Right (Error,st)
```

Note that if the term is a boolean constant, the stack must have at least two elements for the computation to continue. If it has only one element this leads to an error, and if the stack is empty, the result is a *value* state.

Now we give the definition for function application and λ-abstraction.

```
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)

step _ st@(MState (Abs v e, Stack [], _, _, _)) = Right (Value,st)

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

Note that for a λ-abstraction, if the stack is empty, the result is a *value* state, otherwise the computation can go on.

The definition for variables is a bit trickier. If the variable is mapped to a closure in the environment, then the computation continues with this closure. Otherwise, we map the variable to a fresh channel (it has not yet been instantiated).

```
step _ (MState (Var v, s, VEnv lv, env, i)) | v `isIn` lv =
  let Closure cl = fromJust $ lookup v 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

step ctx@(Context c) st@(MState (Var v, s, VEnv lv, env, i)) =
  case lookup v c of
    Nothing -> Right (Error,st)
    Just tv ->
      let ch = Channel ("ch" ++ show i) tv ctx
          cl = Closure (ch, VEnv lv)
      in Left $ MState (ch, s, VEnv ((v,cl):lv), env, i+1)
```

The two last rules are about channels. The first one is for checking if the channel has already been instantiated, and the second one is to signal the need for an instantiation.

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

step _ st@(MState (Channel c _ _, _, _, _, _)) =
  Right (Inst,st)
```

We have now a function that allows us to run the *TKAM*, but we still cannot instantiate channels. To do so we need *atomic normal forms*.

2.2.3 Atomic normal forms

To compute the set of *atomic normal forms* of a certain type in a certain context we need a function with the following signature.

```
atomicNFs :: Type -> Context -> Int -> [(Int, Term)]
```

Note that once again, the parameter of type *Int* and the first field of the couples in the return list are to keep track of fresh names for channels and variables.

We patern-match on the type in the definition. When the type is a function taking an element of type t and returning an element of type t', we take a fresh variable v and compute the ANFs of type t' in the previous context extended with v : t. Finally we enclose these ANFs in an abstraction over v to obtain the desired result.

```
atomicNFs (Fun t t') (Context c) i =
  let var = "var" ++ show i
      anfs = atomicNFs t' (Context ((var,t):c)) (i+1)
  in map (\(i,e) -> (i, Abs var e)) anfs
```

The ANFs of boolean type are the basic cases true and false together with more complex cases handled by the auxiliary function adaptVarType. For each variable in the context, this function build an ANF based on application of arguments to the variables and case expressions. This function will be given latter.

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

We now describe the auxiliary function adaptVarType, required for the definition of the previous function.

```
adaptVarType :: Context -> Type -> Int -> (VName,Type)
             -> (Int,Term)
adaptVarType ctx tc i (v,tv) =
  let ts = components tv
      -- ts : 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
      -- ch : channels to apply to v
      t = foldl App (Var v) ch
      -- t : application of the channels to v
      i' = i + length ts
      ch1 = "ch" ++ show i'
      ch2 = "ch" ++ show (i'+1)
      term = CaseB t (Channel ch1 tc ctx) (Channel ch2 tc ctx)
  in (i' + 2, term)
 where components :: Type -> [Type]
       components (Fun ta tb) = ta : components tb
       components _           = []
```

Once we have got all the *atomic normal forms* of type A in context Γ, it is easy to generate one. To do so, we give the following function.

```
atomicNF :: StdGen -> Type -> Context -> Int -> (StdGen, Term, Int)
atomicNF g t c i = oneOf g $ atomicNFs t c i
```

Note that the oneOf function pick at random an element of a list. It has the following signature.

```
oneOf :: StdGen -> [a] -> (StdGen, a)
```

The type *StdGen* is a random seed that allow the *oneOf* function to be defined.

In order to have all the elements required for testing, we need one more function for intantiating a channel in a state.

```
instantiateChannel :: StdGen              -- Random seed
                   -> MState              -- State
                   -> (StdGen,MState) -- New random seed and 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'))
```

2.2.4 Testing function

We now give the core testing function. It has the following signature.

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

To test a function taking a type t and returning a type t', we test the function applied to a fresh variable v to have type t' in the context extended with v : t.

```
test g (Context c) e (Fun t t') i =
  let v  = "vart" ++ show i
      c' = (v,t):c
      e' = App e (Var v)
  in test g (Context c') e' t' (i+1)
```

If the type tested is the boolean type, then we wrap the term in an empty state, and start looping. In the loop, we run the TKAM using the steps function, and patern-match on the kind of the output state. If the output state is an error, then the test fails, we return False. If the result is a value that is T or F then the test is a success, we return True. The only other kind of value is an abstraction, and this leads to a failure. In the case of a need for instantiation, we instantiate the channel and loop.

```
test g c e Bool _ = let st = MState (e, 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)
              Value -> case term st' of
                         T -> (g,True)
                         F -> (g,True)
                         _ -> (g,False) -- Abstraction
              Inst  -> let (g',st'') = instantiateChannel g st'
                       in loop g' st'' c
```

3 Testing Judgements of PCF

Testing with *PCF* is very similar to testing with *Finite System T*, only a little bit more complex.

Very much like we did in the previous section, we will need to do change the transition function of the *Lazy TKAM* so that it can be used for testing judgements.

We will also extend our implementation with the new features needed for testing in the *PCF* language.

3.1 Using the LTKAM to test judgements

The remark that we made about testing judgements of *Finite System T* still holds for *PCF*, the following judgements are equivalent.

$$\Gamma \vdash f : T_1 \to T_2 \quad \Gamma, x : T_1 \vdash f\ x : T_2$$

Any judgement can be reduced to one of the following forms.

$$\Gamma \vdash a : \text{Nat} \;\; \Gamma \vdash a : \text{Bool}$$

We use the same system as for the transition of the *TKAM* to classify the transition rules in four categories.

The transition rules that lead to a *continuation* states are the following. Four of them are new.

$$\begin{aligned}
\langle \text{case}_{\mathtt{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\ \ x \in \text{dom}(\sigma) &\overset{\Gamma}{\rightsquigarrow} \langle \sigma_1(x),\ \pi,\ \sigma_2(x),\ \tau\rangle \\
\langle x,\ \pi,\ \sigma,\ \tau\rangle\ \ x \notin \text{dom}(\sigma) &\overset{\Gamma}{\rightsquigarrow} \langle c_{\Gamma}^{\Gamma(x)},\ \pi,\ \sigma + \{x \mapsto c_{\Gamma}^{\Gamma(x)}\},\ \tau\rangle \\
\langle c,\ \pi,\ \sigma,\ \tau\rangle\ \ c \in \text{dom}(\tau) &\overset{\Gamma}{\rightsquigarrow} \langle \tau(c),\ \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 \text{case}_{\mathtt{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{aligned}$$

The transition rules leading to *value* states are the following. There are two new rules in this group (natural number values).

$$\langle \mathtt{T},\ \pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{value } \mathtt{T}$$

$$\langle \mathtt{F},\ \pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{value } \mathtt{F}$$

$$\langle \lambda x.M,\ \pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{value } \lambda x.M$$

$$\langle \mathtt{S}\ M,\ \pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{value } \mathtt{S}\ M$$

$$\langle \mathtt{Z},\ \pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{value } \mathtt{Z}$$

The group of transition rules leading to *error* states has two new members..

$$\langle \mathtt{T},\ (N,\sigma_N).\pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{error}$$

$$\langle \mathtt{F},\ (N,\sigma_N).\pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{error}$$

$$\langle \mathtt{Z},\ (N,\sigma_N).\pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{error}$$

$$\langle \mathtt{S}\ M,\ (N,\sigma_N).\pi_0,\ \sigma,\ \tau\rangle \overset{\Gamma}{\leadsto} \text{error}$$

The last group of rules remains the same. There is only a single rule that lead to an *instantiation* state.

$$\langle c,\ \pi,\ \sigma,\ \tau\rangle\ \ c \notin \mathrm{dom}(\tau) \overset{\Gamma}{\leadsto} \text{need instantiation}$$

The difference with the testing framework for *Finite System T* is mainly the adding of a new testing procedure for natual numbers. The testing procedure for judgements with the type Bool is only slightly modified.

To test judgements of the form

$$\Gamma \vdash a : \text{Bool}$$

we proceed as in the previous section. However, when it comes to *value* states there are two more cases (for natural numbers). And obviously, in these two cases, the test must fail. Everything else remains the same.

Now for testing judgements of the form

$$\Gamma \vdash a : \text{Nat}$$

we run the *TKAM* on a, or in other words fully evaluate the following state.

$$\langle a,\ \pi_0,\ \emptyset,\ \emptyset\rangle$$

If the result state is an *error* state, then the test fails. If the result state is a *value* state that is T, F or a λ-abstraction then the test fails. If it is a *value* state that is a Z then the test succeeds, and if it is S M then we remove the S and resume the computation on M.

Note that S M has type Nat only if M has type Nat. That's why the computation needs to continue.

If the result state contains an instantiation request, then we instantiate the channel with an *atomic normal form*, run the *Lazy TKAM* on the new state and continue until the test fails or succeeds (or continue forever, in which case we cannot conclude).

3.2 Implementation

In this section we extend the previous implementation with the features specific to the *Lazy TKAM*, and extend the testing procedures.

3.2.1 Data structures

There are only two changes in the data structures used in the testing framework. The types *Type* and *Term* must be extended with the additional constructs of *PCF* and become the following.

```
data Term = Var VName
          | Abs VName Term
          | App Term Term
          | Y Term
          | T
          | F
          | CaseB Term Term Term
          | Z
          | S Term
          | CaseN Term Term Term
          | Channel CName Type Context

data Type = Bool
          | Nat
          | Fun Type Type
```

3.2.2 Transition function

We also need to add a few rules to the transition function. We need one rule for the fixed-point recursion operator. 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)
```

We also need rules for the case construct over natural numbers, zero and the successor. Note that these rules are very similar to the rules for the boolean case construct.

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

The zero and successor case are almost the same as the true and false case. The only difference 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 cz:_:s), _, env, i)) =
  Left $ MState (fst cz, Stack s, snd cz, env, i)
step _ (MState (S t, Stack (_:Closure cs:s), _, env, i)) =
  Left $ MState (App (fst cs) t, Stack s, snd cs, env, i)
```

If the stack is empty, then we have a value state.

```
step _ st@(MState (Z,   Stack [], _, _, _)) = Right (Value,st)
step _ st@(MState (S t, Stack [], _, _, _)) = Right (Value,st)
```

And if it has only one element then this is an error.

```
step _ st@(MState (Z,    Stack (_:[]), _, _, _)) = Right (Error,st)
step _ st@(MState (S t,  Stack (_:[]), _, _, _)) = Right (Error,st)
```

Every other rule remain the same.

3.2.3 Atomic normal forms

Now we need to change the function for computing the list of the *atomic normal forms* completely. However it still has the same signature, and the case for the function type remains the same too.

```
atomicNFs :: Type           -- Type of the ANF
          -> Context        -- Context
          -> Int            -- Next index for a fresh name
          -> [(Int,Term)]   -- Returns couples next fresh name / term
atomicNFs (Fun t t') (Context c) i =
  let var = "var" ++ show i
      anfs = atomicNFs t' (Context ((var,t):c)) (i+1)
  in map (\(i,e) -> (i, Abs var e)) anfs
```

The ANFs of boolean type are the basic cases true and false together with more complex cases handled by the auxiliary function adaptVarType. For each variable in the context, this function build an ANF based on application of arguments to the variables and case expressions. This function will be given latter.

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

The ANFs of natural number type are defined in a very similar way, only the base case are different. They are zero and the successor of a channel of type natural number type in the same context.

```
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
```

We now describe the auxiliary function adaptVarType, required for the definition of the previous function.

```
adaptVarType :: Context -> Type -> Int -> (VName, Type)
             -> (Int,Term)
adaptVarType ctx tc i (v,tv) =
  let ts = components tv
      -- ts : 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
      -- ch : channels to apply to v
      t = foldl App (Var v) ch
      -- t : 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 (Fun Nat tc) ctx)
 in (i' + 2, term)
where components :: Type -> [Type]
      components (Fun ta tb) = ta : components tb
      components _           = []
      baseType :: Type -> Type
      baseType (Fun _ t) = baseType t
      baseType t         = t
```

3.2.4 Testing function

The core testing function has one more case in its definition. The natural number case is very similar. The only different part is in the handleing of values. Z leads to success, and when we get S t as a value we must continue looping on the state with the S removed. In other cases the test is a failure.

```
test g c e Nat _ = let st = MState (e, 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)
             Value -> case term st' of
                        Z   -> (g,True)
                        S t -> loop g (remS st') c
                        _   -> (g,False) -- Bool or lambda
             Inst  -> let (g',st'') = instantiateChannel g st'
                      in loop g' st'' c
```

Not that the function remS is defined as follow. It only removes the successor which is in the term in the state.

```
remS :: MState -> MState
remS (MState (S t,a,b,c,d)) = MState (t,a,b,c,d)
```

Conclusion and future work

We presented an implementation of a testing framework based on some aspects of the recent work of Peter Dybjer ([2]). We started from the *TKAM* designed by Pierre Clairambault ([1]), and addapted it for the testing of judgements.

This new kind of testing gives other perspectives for the developement of proof assistants since it can be used to debug proofs.

This report was first intended to contain a part about testing judgements of dependent type theory, but adapting the *TKAM* for dependent types was harder than we expected and would have required more time. Peter Dybjer and Pierre Clairambault met recently and succeeded to adapt the *TKAM* to dependent types, and we will work on its implementation soon.

References

[1] Clairambault, P., "Testing semantics for PCF," .

[2] Dybjer, P., "Program Testing and Constructive Validity," 2010.

[3] Claessen, K. and Hughes, J., "QuickCheck: a lightweight tool for random testing of Haskell programs," 2000.

[4] C. Runciman, M. Naylor, F. L., "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values," 2008.

[5] Haiyan, Q., *Testing and Proving in Dependent Type Theory*, 2003.

Appendix A: Literate Haskell doc. (F. System T)

This module contains an implementation of the TKAM adapted for testing judgements of the form

$$\Gamma \vdash a : T$$

where a is a term of the Finite System T language.

First we give a name to the module and import a few things from the standard library.

```
module TestFSystemT where

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

Appendix A.1: Finite System T

We define what a term of the language is. We use an algebraic datatype.

```
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
          | Channel CName Type Context -- Channel
```

Note that channels carry a type and a context.

In order for the channels to be defined completely we need to have types and contexts.

```
data Type = Bool
          | Fun Type Type
        deriving ( Eq )
```

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

As it might be convenient for debuging, we define instances of Show for a term, a type and a context.

```
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 (Channel n t c) = "[" ++ n ++ ":" ++ show t
                            ++ "," ++ show c ++ "]"

instance Show Type where
 show Bool                 = "B"
 show (Fun s@(Fun _ _) t) = "(" ++ show s ++ ")->" ++ show t
 show (Fun s t)            = show s ++ "->" ++ show t

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

Appendix A.2: State of the TKAM

We now define what the state of the TKAM is.

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

As it is to be expected, the state contains a term, a stack, a variable environment and a channel environment. We also add an integer to the definition, this will be useful to get fresh variable names.

To have the definition in full we need to define what are a stack, a variable environment and a channel environment.

```
newtype Stack = Stack [Closure]

newtype Closure = Closure (Term, VEnv)

newtype VEnv = VEnv [(VName, Closure)]

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

As we did before, we define instances of Show for the state of the machine. This might be useful for debuging.

```
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
```

Appendix A.3: Transition function

We will now define the transition function, according to the modifications we did to the LTKAM for testing. We need some kind of flags to tell what kind of state the machine is in after a transtion.

```
data Kind = Error -- There was an error
          | Inst  -- Need for a channel instantiation
          | Value -- Value state
          deriving ( Show )
```

The signature of the transition funtion will be the following.

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

We use the Either datatype in the return type in order to distinguish between the continuation states and the final states (error, instantiation and value state).

We first give the rule for boolean condition. The computation continues with the condition term, and the two other terms are added to the top of the stack.

```
step _ (MState (CaseB c t e, Stack s, lv, env, i)) =
  let ct = Closure (t, lv)
      cf = Closure (e, lv)
  in Left $ MState (c, Stack (ct:cf:s), lv, env, i)
```

Then the rules for true and false can only be applied if there are at least two closures on the stack.

```
step _ (MState (T, Stack (Closure ct:_:s), _, env, i)) =
  Left $ MState (fst ct, Stack s, snd ct, env, i)
step _ (MState (F, Stack (_:Closure cf:s), _, env, i)) =
  Left $ MState (fst cf, Stack s, snd cf, env, i)
```

If there is no closure on the stack, the this is a value.

```
step _ st@(MState (T, Stack [], _, _, _)) = Right (Value,st)
step _ st@(MState (F, Stack [], _, _, _)) = Right (Value,st)
```

And if there is just one, then this is an error.

```
step _ st@(MState (T, Stack (_:[]), _, _, _)) = Right (Error,st)
step _ st@(MState (F, Stack (_:[]), _, _, _)) = Right (Error,st)
```

Now the rule for application only take the argument term to the top of the stack and computation continue 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)
```

In the case of a lambda-abstraction, if the stack is not empty, the the variable in the lambda is mapped to the top of the stack in the variable environment, and the computation keeps going with the body.

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

If the stack is empty however, then the state is a value.

```
step _ st@(MState (Abs v e, Stack [], _, _, _)) = Right (Value,st)
```

For variables, if the variable is mapped to something in the variable environment, the 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. If the context does not contain the variable then it is not exhaustive, this leads to an error.

```
step ctx@(Context c) st@(MState (Var v, s, VEnv lv, env, i)) =
  case lookup v c of
    Nothing -> Right (Error,st)
    Just tv ->
      let ch = Channel ("ch" ++ show i) tv ctx
          cl = Closure (ch, VEnv lv)
      in Left $ MState (ch, s, VEnv ((v,cl):lv), env, i+1)
```

There is an alternative possibility, which is an optimization. In this case we restrict the environment to the variables that are in scope.

```
step ctx@(Context c) st@(MState (Var v, s, VEnv lv, env, i)) =
  case lookup v c of
    Nothing -> Right (Error,st)
    Just tv ->
      let vinenv = map fst lv
          ch = Channel ("ch" ++ show i) tv
                 (Context (filter (\(n,_) -> n `elem` vinenv) c))
          cl = Closure (ch, VEnv lv)
      in Left $ MState (ch, s, VEnv ((v,cl):lv), env, i+1)
```

The latter version seem to always terminate, and the first one seems to diverge, even if it might just be that it is taking a very long time. Sometimes it terminates as expected in about 10 seconds.

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, l, CEnv env, i)) | c `isIn` env =
  let t = fromJust $ lookup c env -- safe since c is in env
  in Left $ MState (t, s, l, 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.

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

We have now a full definition of the transition function. We can define a new function that makes a full transition. Its definition is straight-forward.

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

Appendix A.4: Atomic normal forms

All the ANFs of a certain type in a certain context will be computed by a function with the following signature.

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

Note that the third parameter (which is an integer) is used to obtain fresh variable names and channel names. And similarly, in the return type, terms are coupled with integers to give the next fresh integer if this term is selected.

We patern-match on the type in the definition. When the type is a function taking an element of type t and returning an element of type t', we take a fresh variable v and compute the ANFs of type t' in the previous context extended with v : t. Finally we enclose these ANFs in an abstraction over v to obtain the desired result.

```
atomicNFs (Fun t t') (Context c) i =
  let var = "var" ++ show i
      anfs = atomicNFs t' (Context ((var,t):c)) (i+1)
  in map (\(i,e) -> (i, Abs var e)) anfs
```

The ANFs of boolean type are the basic cases true and false together with more complex cases handled by the auxiliary function adaptVarType. For each variable in the context, this function build an ANF based on application of arguments to the variables and case expressions. This function will be given latter.

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

We now describe the auxiliary function adaptVarType, required for the definition of the previous function.

```
adaptVarType :: Context -> Type -> Int -> (VName,Type)
             -> (Int,Term)
adaptVarType ctx tc i (v,tv) =
  let ts = components tv
      -- ts : 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
      -- ch : channels to apply to v
```

```
        t = foldl App (Var v) ch
        -- t : application of the channels to v
        i' = i + length ts
        ch1 = "ch" ++ show i'
        ch2 = "ch" ++ show (i'+1)
        term = CaseB t (Channel ch1 tc ctx) (Channel ch2 tc ctx)
  in (i' + 2, term)
 where components :: Type -> [Type]
       components (Fun ta tb) = ta : components tb
       components _           = []
```

Now that we can compute all the ANFs of a certain type in a certain context, we need a function to pick one at random.

```
atomicNF :: StdGen                   -- Random seed
         -> Type                     -- Type of the ANF
         -> Context                  -- Context
         -> Int                      -- Next index for a fresh name
         -> (StdGen,(Int,Term))      -- New seed, fresh int, 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'')
```

We also give a function that instantiate a channel in a machine state. This function can only be used if the term in the state is a channel.

```
instantiateChannel :: StdGen              -- Random seed
                   -> MState              -- State
                   -> (StdGen,MState)     -- New random seed and 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'))
```

Appendix A.5: Testing

We now give the core testing function. It has the following signature.

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

To test a function taking a type t and returning a type t', we test the function applied to a fresh variable v to have type t' in the context extended with v : t.

```
test g (Context c) e (Fun t t') i =
  let v  = "vart" ++ show i
      c' = (v,t):c
      e' = App e (Var v)
  in test g (Context c') e' t' (i+1)
```

If the type tested is the boolean type, then we wrap the term in an empty state, and start looping. In the loop, we run the TKAM using the steps function, and patern-match on the kind of the output state. If the output state is an error, then the test fails, we return False. If the result is a value that is T or F then the test is a success, we return True. The only other kind of value is an abstraction, and this leads to a failure. In the case of a need for instantiation, we instantiate the channel and loop.

```haskell
test g c e Bool _ = let st = MState (e, 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)
              Value -> case term st' of
                         T -> (g,True)
                         F -> (g,True)
                         _ -> (g,False) -- Abstraction
              Inst  -> let (g',st'') = instantiateChannel g st'
                       in loop g' st'' c
```

In the previous function we have used an auxiliary functions that helped us get the term inside a state.

```haskell
term :: MState -> Term
term (MState (t,_,_,_,_)) = t
```

Appendix A.6: Convenient functions for testing

We give a function that sequence a given number of tests. It takes as parameter the number of tests to perform, and returns the number of successful tests before failure.

```haskell
runTests :: StdGen                          -- Seed for random generation
         -> Int                             -- Number of distinct tests
         -> Context -> Term -> Type -- Judgement
         -> Int                             -- Nb of success without fail
runTests g 0  c e t = 0
runTests g nb c e t = let (g',r) = test g c e t 0
                      in if r then 1 + runTests g' (nb-1) c e t
                              else 0
```

We also give a function that is more convenient for testing. It is wraped in the IO monad to get access to a random seed from the environment.

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

Appendix A.7: Examples

We test a first term which corresponds to a test that appears in Pierre Clairambault's notes.

The term is the following. It is a function that take as input a function, it then applies T to the function, and if the result it T then it returns the function applied to F, and otherwise T.

```
t1 = Abs "f" (CaseB (App (Var "f") T) (App (Var "f") F) T)
```

The context is the following.

```
c1 = Context [ ("f", Fun Bool Bool) ]
```

And the term is expected to have the followin type in the context.

```
ty1 = Fun (Fun Bool Bool) Bool
```

Finaly we run a test.

```
run1 = quickTest 1000 c1 t1 ty1
```

Appendix B: Literate Haskell doc. (PCF)

This module contains an implementation of the TKAM extended with lazy natural numbers and adapted for testing. It can be used to test judgements of the form

$$\Gamma \vdash a : T$$

where a is a term of the PCF language.

First we give a name to the module and import a few things from the standard library.

```
module TestPCF where

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

Appendix B.1: PCF

We define what a term of the language is. We use an algebraic datatype.

```
type VName = String
type CName = String
data Term = Var VName                          -- Variable
          | Abs VName Term                     -- Lambda-abstraction
          | App Term Term                      -- Application
          | Y Term                             -- Fixed point recursion
          | CaseB Term Term Term               -- Boolean condition
          | T                                  -- True
          | F                                  -- False
          | CaseN Term Term Term               -- Natural condition
          | Z                                  -- Zero
          | S Term                             -- Successor
          | Channel CName Type Context         -- Channel
```

Note that channels carry a type and a context.

In order for the channels to be defined completely we need to have types and contexts.

```
data Type = Bool
          | Nat
          | Fun Type Type
        deriving ( Eq )
```

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

As it might be convenient for debuging, we define instances of Show for a term, a type and a context.

```
instance Show Term where
 show (Var v)           = v
 show (Abs v e)         = "   " ++ v ++ "." ++ show e
 show (App t1 t2)       = "(" ++ show t1 ++ ")␣(" ++ show t2 ++ ")"
 show (Y t)             = "Y␣(" ++ show t ++ ")"
 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 (Channel n t c) = "[" ++ n ++ ":" ++ show t ++ ","
                             ++ show c ++ "]"

instance Show Type where
 show Bool                    = "B"
 show Nat                     = "N"
 show (Fun s@(Fun _ _) t) = "(" ++ show s ++ ")->" ++ show t
 show (Fun s t)               = show s ++ "->" ++ show t

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

Appendix B.2: State of the LTKAM

We now define what the state of the LTKAM is.

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

As it is to be expected, the state contains a term, a stack, a variable environment and a channel environment. We also add an integer to the definition, this will be useful to get fresh variable names.

To have the definition in full we need to define what are a stack, a variable environment and a channel environment.

```
newtype Stack = Stack [Closure]

newtype Closure = Closure (Term, VEnv)

newtype VEnv = VEnv [(VName, Closure)]

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

As we did before, we define instances of Show for the state of the machine. This might be useful for debuging.

```
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
```

Appendix B.3: Transition function

We will now define the transition function, according to the modifications we did to the LTKAM for testing. We need some kind of flags to tell what kind of state the machine is in after a transtion.

```
data Kind = Error -- There was an error
          | Inst  -- Need for a channel instantiation
          | Value -- Value state
        deriving ( Show )
```

The signature of the transition funtion will be the following.

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

We use the Either datatype in the return type in order to distinguish between the continuation states and the final states (error, instantiation and value state).

We first give the rule for boolean condition. The computation continues with the condition term, and the two other terms are added to the top of the stack.

```
step _ (MState (CaseB c t e, Stack s, lv, env, i)) =
  let ct = Closure (t, lv)
      cf = Closure (e, lv)
  in Left $ MState (c, Stack (ct:cf:s), lv, env, i)
```

Then the rules for true and false can only be applied if there are at least two closures on the stack.

```
step _ (MState (T, Stack (Closure ct:_:s), _, env, i)) =
  Left $ MState (fst ct, Stack s, snd ct, env, i)
step _ (MState (F, Stack (_:Closure cf:s), _, env, i)) =
  Left $ MState (fst cf, Stack s, snd cf, env, i)
```

If there is no closure on the stack, the this is a value.

```
step _ st@(MState (T, Stack [], _, _, _)) = Right (Value,st)
step _ st@(MState (F, Stack [], _, _, _)) = Right (Value,st)
```

And if there is just one, then this is an error.

```
step _ st@(MState (T, Stack (_:[]), _, _, _)) = Right (Error,st)
step _ st@(MState (F, Stack (_:[]), _, _, _)) = Right (Error,st)
```

We now switch to the case construct over natural numbers. This is a very similar case.

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

The zero and successor case are almost the same as the true and false case. The only difference 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 cz:_:s), _, env, i)) =
  Left $ MState (fst cz, Stack s, snd cz, env, i)
step _ (MState (S t, Stack (_:Closure cs:s), _, env, i)) =
  Left $ MState (App (fst cs) t, Stack s, snd cs, env, i)
```

If the stack is empty, then we have a value state.

```
step _ st@(MState (Z,   Stack [], _, _, _)) = Right (Value,st)
step _ st@(MState (S t, Stack [], _, _, _)) = Right (Value,st)
```

And if it has only one element then this is an error.

```
step _ st@(MState (Z,   Stack (_:[]), _, _, _)) = Right (Error,st)
step _ st@(MState (S t, Stack (_:[]), _, _, _)) = Right (Error,st)
```

Now the rule for application only take the argument term to the top of the stack and computation continue 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)
```

In the case of a lambda-abstraction, if the stack is not empty, the the variable in the lambda is mapped to the top of the stack in the variable environment, and the computation keeps going with the body.

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

If the stack is empty however, then the state is a value.

```
step _ st@(MState (Abs v e, Stack [], _, _, _)) = Right (Value,st)
```

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)
```

For variables, if the variable is mapped to something in the variable environment, the 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. If the context does not contain the variable then it is not exhaustive, this leads to an error.

```
step ctx@(Context c) st@(MState (Var v, s, VEnv lv, env, i)) =
  case lookup v c of
    Nothing -> Right (Error,st)
    Just tv ->
      let ch = Channel ("ch" ++ show i) tv ctx
          cl = Closure (ch, VEnv lv)
      in Left $ MState (ch, s, VEnv ((v,cl):lv), env, i+1)
```

There is an alternative possibility, which is an optimization. In this case we restrict the environment to the variables that are in scope.

```
step ctx@(Context c) st@(MState (Var v, s, VEnv lv, env, i)) =
  case lookup v c of
    Nothing -> Right (Error,st)
    Just tv ->
      let vs = map fst lv
          ch = Channel ("ch" ++ show i) tv
                 (Context (filter (\(n,_) -> n `elem` vs) c))
          cl = Closure (ch, VEnv lv)
      in Left $ MState (ch, s, VEnv ((v,cl):lv), env, i+1)
```

The latter version seem to always terminate, and the first one seems to diverge, even if it might just be that it is taking a very long time. Sometimes it terminates as expected in about 10 seconds.

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, l, CEnv env, i)) | c `isIn` env =
  let t = fromJust $ lookup c env -- safe since c is in env
  in Left $ MState (t, s, l, 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.

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

We have now a full definition of the transition function. We can define a new function that makes a full transition. Its definition is straight-forward.

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

Appendix B.4: Atomic normal forms

All the ANFs of a certain type in a certain context will be computed by a function with the following signature.

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

Note that the third parameter (which is an integer) is used to obtain fresh variable names and channel names. And similarly, in the return type, terms are coupled with integers to give the next fresh integer if this term is selected.

We patern-match on the type in the definition. When the type is a function taking an element of type t and returning an element of type t', we take a fresh variable v and compute the ANFs of type t' in the previous context extended with v : t. Finally we enclose these ANFs in an abstraction over v to obtain the desired result.

```
atomicNFs (Fun t t') (Context c) i =
  let var = "var" ++ show i
      anfs = atomicNFs t' (Context ((var,t):c)) (i+1)
  in map (\(i,e) -> (i, Abs var e)) anfs
```

The ANFs of boolean type are the basic cases true and false together with more complex cases handled by the auxiliary function adaptVarType. For each variable in the context, this function build an ANF based on application of arguments to the variables and case expressions. This function will be given latter.

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

The ANFs of natural number type are defined in a very similar way, only the base case are different. They are zero and the successor of a channel of type natural number type in the same context.

```
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
```

We now describe the auxiliary function adaptVarType, required for the definition of the previous function.

```
adaptVarType :: Context -> Type -> Int -> (VName,Type)
             -> (Int,Term)
adaptVarType ctx tc i (v,tv) =
  let ts = components tv
      -- ts : 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
      -- ch : channels to apply to v
      t = foldl App (Var v) ch
      -- t : 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 (Fun Nat tc) ctx)
  in (i' + 2, term)
 where components :: Type -> [Type]
       components (Fun ta tb) = ta : components tb
       components _           = []
       baseType :: Type -> Type
       baseType (Fun _ t) = baseType t
       baseType t         = t
```

Now that we can compute all the ANFs of a certain type in a certain context, we need a function to pick one at random.

```
atomicNF :: StdGen                    -- Random seed
         -> Type                      -- Type of the ANF
         -> Context                   -- Context
         -> Int                       -- Next index for a fresh name
         -> (StdGen,(Int,Term))       -- New seed, 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'')
```

We also give a function that instantiate a channel in a machine state. This function can only be used if the term in the state is a channel.

```
instantiateChannel :: StdGen              -- Random seed
                   -> MState              -- State
                   -> (StdGen,MState)     -- New random and 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'))
```

Appendix B.5: Testing

We now give the core testing function. It has the following signature.

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

To test a function taking a type t and returning a type t', we test the function applied to a fresh variable v to have type t' in the context extended with v : t.

```
test g (Context c) e (Fun t t') i =
  let v  = "vart" ++ show i
      c' = (v,t):c
      e' = App e (Var v)
  in test g (Context c') e' t' (i+1)
```

If the type tested is the boolean type, then we wrap the term in an empty state, and start looping. In the loop, we run the TKAM using the steps function, and patern-match on the kind of the output state. If the output state is an error, then the test fails, we return False. If the result is a value that is T or F then the test is a success, we return True. Other kind of values are functions and natural numbers. This is a failure. In the case of a neef for instantiation, we instantiate the channel and loop.

```
test g c e Bool _ = let st = MState (e, 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)
              Value -> case term st' of
                         T -> (g,True)
                         F -> (g,True)
                         _ -> (g,False) -- Nat or lambda
              Inst  -> let (g',st'') = instantiateChannel g st'
                       in loop g' st'' c
```

Finally, the natural number case is very similar. The only different part is in the handleing of values. Z leads to success, and when we get S t as a value we must continue looping on the state with the S removed. In other cases the test is a failure.

```
test g c e Nat _ = let st = MState (e, 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)
              Value -> case term st' of
                         Z   -> (g,True)
                         S t -> loop g (remS st') c
                         _   -> (g,False) -- Bool or lambda
              Inst  -> let (g',st'') = instantiateChannel g st'
                       in loop g' st'' c
```

In the previous function we have used some auxiliary functions that helped us get the term inside a state or remove the S on the term of a state.

```
term :: MState -> Term
term (MState (t,_,_,_,_)) = t

remS :: MState -> MState
remS (MState (S t,a,b,c,d)) = MState (t,a,b,c,d)
```

Appendix B.6: Convenient functions for testing

We give a function that sequence a given number of tests. It takes as parameter the number of tests to perform, and returns the number of successful tests before failure.

```
runTests :: StdGen                       -- Random seed
         -> Int                          -- Number of tests to run
         -> Context -> Term -> Type -- Judgement
```

```
          -> Int                              -- Nb of success
runTests g 0  c e t = 0
runTests g nb c e t = let (g',r) = test g c e t 0
                      in if r then 1 + runTests g' (nb-1) c e t
                              else 0
```

We also give a function that is more convenient for testing. It is wraped in the IO monad to get access to a random seed from the environment.

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

Appendix B.7: Examples

We gave everything that is needed for testing, and now we are going to make some tests. But before we give some convenient functions that allow some shortcuts during the creation of a term.

```
toNat :: Int -> Term
toNat n | n < 0 = error "No negative integers..."
toNat 0 = Z
toNat n = S (toNat (n-1))

fromNat :: Term -> Int
fromNat Z     = 0
fromNat (S t) = 1 + fromNat t
fromNat _     = error "Not a natural number..."
```

We test a first term which correspionds to a function taking as input a natural number and returning true if it is zero and false otherwise (this is an implementation of the isZ function).

```
t1 = Abs "n" (CaseN (Var "n") T (Abs "x" F))
```

We test this term in the following context.

```
c1 = Context [ ("n", Nat), ("x", Nat) ]
```

The type we want to test is the following.

```
ty1 = Fun Nat Bool
```

We can do a test run as follow.

```
run1 = quickTest 1000 c1 t1 ty1
```

Here is an other test example. The term corresponds to a function taking a function f as input and returning true if f 0 is equal to 0 and false otherwise.

```
t2 = Abs "f" (CaseN (App (Var "f") Z) T (Abs "x" F))
c2 = Context [ ("f", Fun Nat Nat), ("x", Nat) ]
ty2 = Fun (Fun Nat Nat) Bool

run2 = quickTest 1000 c2 t2 ty2
```

The last test is the one that appears in Pierre Clairambault's notes.

```
t3 = Abs "f" (CaseB (App (Var "f") T) (App (Var "f") F) T)
c3 = Context [ ("f", Fun Bool Bool) ]
ty3 = Fun (Fun Bool Bool) Bool

run3 = quickTest 1000 c3 t3 ty3
```

Appendix C: Résumé et mots clés

Dans ce rapport nous étudions une nouvelle notion de test pour la théorie des types. Nous commencons par décrire la *Krivine Abstract Machine (KAM)* qui est une machine virtuelle permettant d'évaluer des termes du λ-calcul. Nous présentons ensuite la *Testing KAM* qui est une version modifiée de la *KAM* que Pierre Clairambault a décrite dans des notes ([1]).

Nous utilisons ensuite deux versions de la *TKAM*, la première restreinte au language *Finite System T* et la seconde dérivée de la version originale laquelle nous ajoutons des entiers naturels lazy.

Les machines virtuelles sont ensuite utilisées comme la partie centrale d'une procedure de test pour le typage de termes.

Ce travail est une implémentation de certains aspects d'un manuel de test présenté par Peter Dybjer dans son papier *Program Testing and Constructive Validity* [2].

Mots clés: instantiation / generation paresseuse, fonction de transition, test de typage, implémentation.