Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option Parallel, Distributed and Embedded Systems

A Classical Realizability Interpretation of Judgement Testing

Rodolphe Lepigre

19/06/2013

Research project performed at the LAMA, Université de Savoie

Under the direction of:
Pierre Hyvernat, LAMA, Université de Savoie
Christophe Raffalli, LAMA, Université de Savoie

Defended before a jury composed of:
Prof. Marie-Christine Fauvet, LIG, UJF
Prof. Olivier Gruber, INRIA, UJF
Prof. Martin Heusse, LIG, ENSIMAG
Prof. Arnaud Legrand, LIG, UJF
Prof. Alexandre Miquel, ENS Lyon
Prof. Noël de Palma, IMAG, UJF (president)

June 2013

Abstract

A notion of test for intuitionistic type theory has recently been introduced by Peter Dybjer and his collaborators [Peter Dybjer 2010]. It is meant to be the foundation for automatic testing tools that could be implemented in proof assistants such as Coq or Agda. Such tools would provide a way to test, at any time during the construction of a proof, if the current goal can be completed in the context. The failure of such a test would mean that the goal is impossible to prove, and its success would corroborate the partial result.

In this report, we investigate the possibility of extending the testing procedure to classical systems. We propose an interpretation of the testing procedure in terms of Krivine's classical realizability. Finally we show that the notion of test is correct, in the sense that a judgement is valid if and only if it passes every possible test.

Résumé

Peter Dybjer et ses collaborateurs ont introduit une notion de test pour la théorie des types intuitionniste [Peter Dybjer 2010]. Le but de ces récents travaux est de servir de fondation à des outils de test automatiques qui pourraient être implémentés, à terme, dans des assistants de preuve tels que Coq ou encore Agda. De tels outils permettraient aux utilisateurs de tester une preuve en cours de construction, et ainsi, soit réfuter leur résultat avec un échec, soit le corroborer avec des résultats positifs. Ici, faire un test signifie vérifier si le but courant peut être complété ou non, dans le contexte.

Dans ce rapport, nous explorons les possibilités d'extension de cet outil de test aux systèmes classiques. En outre, une interprétation du test en terme de la réalisabilité classique de Krivine est proposée. Nous montrons également que le système de test est correct, ce qui signifie qu'un jugement est valide si et seulement passe tout les tests.

Table of contents

1 Introduction

For several decades, our world has been governed by computer programs. We entrust them with many business-critical tasks, and sometimes even our lives. Reasonable code-writers have come to cherish testing, as it is particularly helpful for ensuring that a program behaves correctly. However, testing can be a tedious task, which is why it is often neglected. Automatic testing tools have been designed to overcome this problem.

1.1 Automatic testing tools

There are a number of frameworks that allow testing with automatic input generation such as *QuickCheck* [Koen Claessen, John Hughes 2000] or *SmallCheck* [Colin Runciman, Matthew Naylor, Fredrik Lindblad 2008] for the programming language *Haskell*. These tools allow the user to test predicates over the application of generated input to functions, the main difference between *QuickCheck* and *SmallCheck* being the method of generating input data. The former uses random generation while the latter generates input in a systematic way.

1.1.1 Specification-based testing

In order to illustrate the possibilities offered by such tools, let us consider the following well-known example of a *Haskell* program implementing a sorting algorithm (quick-sort).

```
qsort :: [Int] -> [Int]
qsort []     = []
qsort (x:xs) = qsort ltx ++ [x] ++ qsort mtx
  where ltx = filter (< x)  xs
        mtx = filter (> x) xs
```

There are two properties that we can expect from a sorting function:

- the output list should contain exactly the same elements as the input list,
- and the output list should be ordered.

These properties can be formulated as predicates over the input data of the sorting function. Finally, one can run arbitrary tests on the predicates by calling the function `quickCheck`.

```
propEqElem :: [Int] -> Bool
propEqElem l = bagEq l (qsort l)
  where bagEq :: [Int] -> [Int] -> Bool
        bagEq x y = null (x \\ y) && null (y \\ x)

propOrdered :: [Int] -> Bool
propOrdered l = ordered (qsort l)
  where ordered (x:y:xs) = x <= y && ordered (y:xs)
        ordered _        = True

test :: IO ()
test = do quickCheck propEqElem
          quickCheck propOrdered
```

When the `test` function is run, we get the following message, telling us that the seventh test on the first predicate failed, while every test passed for the second predicate.

```
> test
**** Failed! Falsifiable (after 7 tests and 2 shrinks): [2,2]
+++ OK, passed 100 tests.
```

We now know that our implementation of the sorting function is not correct with respect to the first property that we stated, i.e. it may happen that the output list does not contains exactly the same element as the input list.

1.1.2 Debugging by testing

In the previous section, we showed that testing was a good way to check if a program behaves according to its specification. Another use of testing, although closely related, is debugging. When a program, function or procedure does not behave as expected, a programmer can often isolate the problem by running tests with carefully-chosen input data, and by observing the external behaviour of the program, function or procedure.

Let us go back to our example with the sorting function. Upon test failure, the input data is shrunk by *QuickCheck*, and a minimal example of an input leading to a test failure is returned. In our case, the minimum failing instance is the list `[2,2]`. We can easily deduce that the problem arises when an element appears twice in the list, and isolate the problem: elements equal to the pivot are filtered out, and simply forgotten. Once the problem is corrected, every test passes.

```
qsort :: [Int] -> [Int]
qsort []     = []
qsort (x:xs) = qsort ltx ++ [x] ++ qsort mtx
  where ltx = filter (< x)  xs
        mtx = filter (>= x) xs -- forgot = here

> test
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

1.2 Proof assistants

As was wisely pointed out by Dijkstra, testing can be a very good way to find bugs, but it cannot be used to show their absence. In fact, this is not true for functions having a finite domain, since one can exhaustively try every possible input (this was the main motivation for *SmallCheck* [Colin Runciman, Matthew Naylor, Fredrik Lindblad 2008]). However, such functions are rather rare in real-life settings where integers, lists and trees are manipulated all the time. One can also think of higher-order types such as functions: $\mathbb{N} \to \mathbb{B}$ is not even countable. When considering such common examples, Dijkstra's remark is fully realised. However, we would like to be able to reason about programs or functions that have an infinite domain as well.

Proof assistants are programs designed to lead users through the construction of formal proofs. One of the main uses of such tools is program proving, i.e. proving that a program behaves according to its specification. Fortunately, this technique provides a way to reason about any program, including those having an infinite domain.

A lot of progress has been made with proof assistants such as *Coq* or *Agda*. They are able to build a formal proof with less and less help from the human user, and can be used to prove that a program meets its specification. However, even a program

that has been proved should not be trusted entirely. Indeed, as Peter Dybjer and many others before him highlighted, many things can go wrong. For instance, it could be that the specification does not capture the desired behaviour, or that there is a bug in the implementation of the proof-assistant. In any case, it is always safer to run some tests before putting programs into production.

Building a formal proof with a proof assistant can be very tricky, and it is likely that many programming mistakes will be made during the process. Many such bugs will be discovered by the proof assistant, but some others might be hard to track. Moreover, there are very few tools and techniques that can be used to debug proofs and specifications.

This led Peter Dybjer and his collaborators to introduce a notion of test for intuitionistic type theory [Peter Dybjer 2010] a few years ago. It is analogous to the notion of test discussed in the previous section, in the sense that it can be used for debugging. This new notion of test affects the testing of judgements in intuitionistic type theory, and it is meant to be the foundation for automatic testing tools that could be implemented in proof assistants. Such tools would provide a way to test, at any time during the construction of a proof, if the current goal can be completed in the context. The success of a test would corroborate the correctness of the partial proof, but the failure of a test would convey the impossibility for the proof to be completed.

1.3 A brief history of judgement testing, motivations

The notion of judgement testing was first introduced by Peter Dybjer, who related program testing to Martin-Löf's meaning explanation for intuitionistic type theory [Peter Dybjer 2010]. In his paper, the four forms of judgement present in Martin-Löf's type theory are considered to be hypothetical and can be corroborated or refuted by testing.

Pierre Clairambault then focused only on typing judgements and initiated the design of a testing procedure based on Krivine's Abstract Machine for the PCF language [Pierre Clairambault]. As part of a previous internship under the direction of Peter Dybjer, the current author refined the design and implemented the testing procedure [Rodolphe Lepigre 2012].

For their part, Peter Dybjer and Pierre Clairambault are currently working on the design of a testing procedure that would work in Martin Löf's dependent type theory [Peter Dybjer, Pierre Clairambault 2012]. Their aim is to implement in full the testing manual for Martin Löf's dependent type theory presented by Peter Dybjer in his paper *Program Testing and Constructive Validity* [Peter Dybjer 2010].

During a presentation at the LAMA (Université de Savoie) of the author's previous research [Rodolphe Lepigre 2012], it was noticed that, as the central part of the procedure relies on Krivine's Abstract Machine, it would be logical to relate the notion of test in type theory to Krivine's classical realizability, which led to the current work. The interest of this work is twofold: it will give another interpretation of the testing of typing judgements and relate it to realizability, and it will also extend the notion of test to classical systems through the *call/cc* instruction. Moreover, the soundness of the testing procedure will be expressed in terms of the adequacy of the realizability system.

1.4 Contents of this report

In chapter 2 we provide the necessary background for this work. This report is intended to be self-contained, therefore we will start with the very basics of logic, the λ-calculus, type theory and classical realizability. Readers familiar with these are advised to skim through this chapter all the same in order to familiarize themselves with our notations (even though they are generally standard).

The real content of this work and our main contributions are contained in chapters 3, 4 and 5. In chapter 3 we present a Krivine's abstract machine where environments are left explicit (i.e. substitutions are not applied directly). We prove an adequation lemma for this system.

In chapter 4 we introduce contextually typed holes (CTH) which will represent holes to be filled in a proof. The realizability model of chapter 3 is extended to contain a new typing rule related to CTH. The resulting adequation lemma gives us a correction result saying that if a judgement is valid, then the execution cannot go wrong. The contrapositive tells us that if but a single test fails, then the judgement is not valid.

Finally, in chapter 5, we define the testing procedure. In particular, we show what preprocessing is required before being able to test a judgement. We also consider some examples.

2 Prerequisites

In this chapter we provide the background for a reader not familiar with type theory and classical realizability. We also introduce the notations that we will use throughout this thesis. Readers familiar with logic, type theory and classical realizability can safely skip this part and go straight to chapter 3. However, it might be a good idea to skim through this part, if only to become familiar with our notations.

2.1 Constructive logic

Let $\mathcal{A} = \{A, B, C...\}$ be a countable set of atomic formulae, and $\bot$ be an atomic formula representing the absurdity. Let $\mathcal{F}$ be the set of all formulae. It is defined to be the smallest set such that $\mathcal{A} \subset \mathcal{F}$, $\bot \in \mathcal{F}$ and if $A, B \in \mathcal{F}$, then $A \to B \in \mathcal{F}$. The formula $A \to B$ is to be read "A implies B". It states that if A is true, then B is also true. Other logical connectives can be defined in terms of the implication connective:

- logical negation $\neg A$ can be defined as $A \to \bot$,
- disjunction $A \vee B$ can be defined as $(A \to B) \to B$,
- conjunction $A \wedge B$ can be defined as $(A \to (B \to \bot)) \to \bot$.

As formulae are often written linearly (and not as trees), we use parentheses to remove any ambiguity. However we will adopt the usual conventions in order to lighten the notations:

- The implication arrow $\to$ is associative to the right, which means that the formula $A \to B \to C$ should be understood as $A \to (B \to C)$.
- The implication symbol $\to$ binds stronger than the conjunction symbol $\wedge$ and the disjunction symbol $\vee$, which in turn bind stronger than the negation symbol $\neg$. The formula $(\neg A \vee B) \wedge \neg C \to D$ should be understood as $(((\neg A) \vee B) \wedge (\neg C)) \to D$.

Definition 2.1.1 Given a countable set of atomic formulae $\mathcal{A}$, the set of well-formed formulae $\mathcal{F}$ is generated by the following BNF grammar:

$$F ::= A \mid \bot \mid F \to F \qquad\qquad A \in \mathcal{A}$$

Proof-trees are constructed using logical rules. The basic syntactic unit used to build proofs are called sequents, and are denoted $\Gamma \vdash A$ where Γ is a set of formulae to be interpreted as a set of hypotheses, and A is a formula to be interpreted as the conclusion. We will use the Greek capital letters Γ, Δ and Σ to range over sets of formulae.

Definition 2.1.2 A sequent is a couple denoted $\Gamma \vdash A$ where $\Gamma \subset \mathcal{F}$ is a finite set of formulae (hypotheses), and $A \in \mathcal{F}$ is a formula (conclusion).

In order to lighten the notations, if $\Gamma = \{A_1, \ldots, A_n\}$ and $\Delta = \{A_1, \ldots, A_n, A\}$ are sets of formulae, and if C is a formula, then
- writing $\Gamma \vdash C$ is equivalent to writing $A_1, \ldots, A_n \vdash C$, and
- writing $\Delta \vdash C$ is equivalent to writing $\Gamma, A \vdash C$.

Deduction rules consist of a finite set (potentially empty) of sequents called premises, and one sequent called conclusion separated by a horizontal bar at the right of which is usually written the name of the rule.

Definition 2.1.3 A deduction rule has the following form, where $\{S_1, \ldots, S_n\}$ is a finite set of sequents forming premises, S is a sequent forming the conclusion, and ν is the name of the rule.

$$\frac{S_1 \quad \ldots \quad S_n}{S}\,\nu$$

We will now give the deduction rules of the system. The first rule has no premise and simply states that when a formula is among the hypotheses, then we can prove it. This rule is named *axiom*.

$$\frac{}{\Gamma, A \vdash A}\,\text{ax}$$

There are two rules related to the implication connective $\rightarrow$. The first one is an introduction rule, and the second one is an elimination rule. The introduction rule has in its conclusion an additional occurrence of the connective, while the elimination rule contain, in one of its premises, an occurrence of the connective which will disappear in the conclusion.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\,\rightarrow_i \qquad\qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\,\rightarrow_e$$

The name of these rules are, respectively, *arrow introduction* and *arrow elimination* (commonly referred to as *modus ponens*). The first rule states that when a formula B holds

under assumptions $\Gamma \cup \{A\}$, then it is true that A implies B under assumptions Γ. The second rule is quite straight-forward: if under some assumptions we know that A implies B, and under the same assumptions we know that A holds, then we can deduce that B holds as well.

As an example, we give a proof of the formula $(A \to B) \to (\neg B \to \neg A)$, which can be seen as a rule for doing a proof by contraposition. We will use the letter Γ as a shorthand for $\{A \to B, \neg B, A\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B, \neg B, A \vdash B \to \bot}\text{ax} \quad \dfrac{\dfrac{}{\Gamma \vdash A \to B}\text{ax} \quad \dfrac{}{\Gamma \vdash A}\text{ax}}{A \to B, \neg B, A \vdash B}\to_e}{A \to B, \neg B, A \vdash \bot}\to_e}{A \to B, \neg B \vdash \neg A}\to_i}{A \to B \vdash \neg B \to \neg A}\to_i}{\vdash (A \to B) \to (\neg B \to \neg A)}\to_i$$

2.2 λ-calculus and type theory

In this section, we will introduce the λ-calculus, which is a very simple language which was designed in the Thirties by Alonzo Church. We will also present a type system which will form the simply-typed λ-calculus.

2.2.1 λ-calculus

The λ-calculus consists of only three kinds of terms: variables coming from a countable set $\Lambda_x = \{x, y, z...\}$, λ-abstractions which allow the construction of functions, and function applications. The language Λ of the λ-calculus is generated by the following BNF grammar:

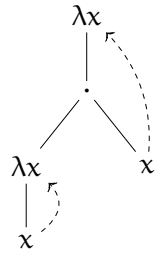
$$t ::= x \mid \lambda x.t \mid (t)\,t \qquad\qquad x \in \Lambda_x$$

Note that we use Krivine's notation for function application: we write $(f)\,a$ and not $(f\,a)$ for the application of function f to the argument a. Let us consider the following examples of λ-terms, which represent simple mathematical objects:

1. $\lambda x.x$ and $\lambda y.y$ both represent the identity function. They take one parameter as input, and return it. The only difference between these two functions is the name used for the argument variable.
2. $((\lambda x.\lambda y.x)\,a)\,b$ is the application of a function to two arguments, a and b. The function simply returns the first of its two arguments, and discards the second.

3. $\lambda g.\lambda f.\lambda x.(g)\,(f)\,x$ is a function taking two functions, f and g, as input, and returning the function $g \circ f$.

A λ-abstraction is a binding term. The variable which is abstracted on can be used in the body of the abstraction to denote the argument of the function. An occurrence of a variable x can either be free, which means that it does not appear in the body t of a λ-abstraction $\lambda x.t$, or it can be bound. A bound variable can appear in the body of several nested λ-abstractions, however, the one it is bound to is always the first encountered when going up the syntactic tree of the term. For example, in the term $\lambda x.(\lambda x.x)\,x$, the first occurrence of the variable term x is bound to the second λ-abstraction, and the second occurrence of x is bound to the first λ-abstraction.



For any term t, we can define the set of variables that appear free in t, and the set of variables that do not appear free in t (bound variables). In example 1, both terms that represent the identity function have no free variables, but only one bound variable. In example 2, the set of free variables is $\{a, b\}$, and the set of bound variables is $\{x, y\}$.

Definition 2.2.4 Let t be a λ-term. The set of bound variables and the set of free variables in t, respectively denoted $\mathrm{BV}(t)$ and $\mathrm{FV}(t)$, are defined inductively on the structure of t as follows:

$$\mathrm{BV}(x) = \emptyset \qquad \mathrm{BV}(\lambda x.t) = \{x\} \cup \mathrm{BV}(t) \qquad \mathrm{BV}(u\,v) = \mathrm{BV}(u) \cup \mathrm{BV}(v)$$

$$\mathrm{FV}(x) = \{x\} \qquad \mathrm{FV}(\lambda x.t) = \mathrm{FV}(t) \setminus \{x\} \qquad \mathrm{FV}(u\,v) = \mathrm{FV}(u) \cup \mathrm{FV}(v)$$

Definition 2.2.5 A term t such as $\mathrm{FV}(t) = \emptyset$ is said to be closed. Otherwise it is said to be open.

Example 1 displays two different representations of the identity function. In fact, it is always possible to find an infinite number of λ-abstractions representing the same

function by changing the name of the variable that is bound. However, we would like to relate these terms when reasoning about program equality, since they will have exactly the same behaviour. We can define an equivalence relation over the set of terms in order to consider such functions equal.

Definition 2.2.6 Two λ-terms u and v are said to be α-equivalent if they are equal up to the renaming of bound variables, under the constraint that no free variable becomes bound during the process. We will denote this $u \equiv_\alpha v$.

The evaluation procedure of the λ-calculus requires a way to substitute a free variable x in a term t with another term u. Although defining this operation seems trivial, it poses a new problem: capture. Since the term u may contain free variables, if we are not careful and only replace every occurrence of x by u, it may be that free variables in u will become bound. We would like to avoid this, since it would completely change the behaviour of the term. Let us consider as an example the term $t = \lambda y.x$, which represents a constant function. Now suppose that we wish to replace every occurrence of x by the term $u = y$. If we naively substitute x by y, the resulting term will be $\lambda y.y$ which is the identity function. One way to solve this problem would be to first rename the bound variable y in t to z, and do the substitution afterwards.

Definition 2.2.7 Let t and u be terms, and x be a variable. The capture-avoiding substitution of variable x by the term u in the term t, denoted $t[x := u]$ is defined inductively on the structure of t.

$$x[x := u] = u \qquad y[x := u] = y \quad \text{if} \quad y \neq x$$

$$(\lambda x.t)[x := u] = \lambda x.t \qquad ((t)\, v)[x := u] = (t[x := u])\, v[x := u]$$

$$(\lambda y.t)[x := u] = \lambda y'.\, t'[x := u] \quad \text{if} \quad y \neq x \text{ and } \lambda y.t \equiv_\alpha \lambda y'.\, t' \text{ with } \quad y' \notin \mathrm{FV}(u)$$

Definition 2.2.8 A β-redex is a term of the form $(\lambda x.t)\, u$ where t is a term in which x may appear free, and u is a term.

The evaluation of terms of the λ-calculus relies on β-reduction, which is a procedure for eliminating β-redexes. It can be intuitively thought of as function application. When a term does not contain any β-redex, it is said to be in β-normal-form.

Definition 2.2.9 Given a β-redex $(\lambda x.t)\,u$, we define its β-reduction to be $t[x := u]$. We denote this operation $\to_\beta$.

$$(\lambda x.t)\,u \to_\beta t[x := u]$$

Let us now consider again our term examples, and fully reduce them in order to obtain terms in β-normal-form. The terms in example 1 and 3 do not contain any β-redex, hence they cannot be reduced any further. However, example 2 can be reduced in the following way, which yields the expected result:

$$((\lambda x.\lambda y.x)\,a)\,b \to_\beta ((\lambda y.x)[x := a])\,b = (\lambda y.a)\,b \to_\beta a[y := b] = a$$

In the previous examples, the evaluation was finite, i.e. it terminated in a finite number of steps. However this is not always the case. We can consider the term $(\lambda x.(x)\,x)\,\lambda x.(x)\,x$ as a counter-example.

$$(\lambda x.(x)\,x)\,\lambda x.(x)\,x \to_\beta ((x)\,x)[x := \lambda x.(x)\,x] = (\lambda x.(x)\,x)\,\lambda x.(x)\,x \to_\beta \ldots$$

2.2.2 Simply-typed lambda-calculus

Since correctness properties such as termination are often not decidable, type systems have been designed in order to identify correct programs. Of course a type-system cannot be made to identify every correct program because if it were the case, the correctness property would be decidable. The price to pay is that some correct programs are rejected by the type-system. In this section we introduce a very simple type-system for the λ-calculus, which is called simply-typed λ-calculus.

We consider a fixed set of atomic types $\mathcal{B}$ that is non-empty and finite. The set of all types $\mathcal{T}$ contains atomic types, and functions. It is generated by the following BNF grammar:

$$\tau ::= \varphi \mid \tau \to \tau \qquad\qquad \varphi \in \mathcal{B}$$

In order to be able to type open terms, we need to introduce typing contexts. They will provide a type to every free variable in the considered term of the language.

Definition 2.2.10 A typing context is a partial function $\Gamma : \Lambda_x \to \mathcal{T}$ mapping variables to their type. It can be represented as a list mapping every variable in $\mathrm{dom}(\Gamma)$ to its type, according to the following BNF grammar:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \qquad\qquad x \in \Lambda_x, \tau \in \mathcal{T}$$

Definition 2.2.11 A typing judgement is a triple of a typing context Γ, a term t of the λ-calculus, and a type τ, with the condition that $\mathrm{FV}(t) \subseteq \mathrm{dom}(\Gamma)$. It is denoted $\Gamma \vdash t : \tau$ and read "t has type τ in context Γ".

The type of a term is derived using rules. These typing rules will look very similar to the deduction rules presented for intuitionistic logic, but we will discuss this in more detail later.

Definition 2.2.12 A typing rule has the following form, where $\{J_1, \ldots, J_n\}$ is a finite set of typing judgements forming premises, J is a judgement forming the conclusion, and ν is the name of the rule:

$$\frac{J_1 \quad \ldots \quad J_n}{J}\,\nu$$

The simply-typed λ-calculus has only three rules. The first one gives to variables the type they have in the typing context.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\,\mathrm{Var}$$

The second rule allows us to give a type to a functions. If the body t of the function has type B in the context Γ extended with the argument variable x of type A, then the function $\lambda x.t$ has type $A \to B$.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B}\,\mathrm{Fun}$$

Finally, the third rule concerns function application, and simply says that if we have a function u of type $A \to B$ and an element v of type A, then we can apply u to v and obtain an element of type B.

$$\frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\,v : B}\,\mathrm{App}$$

Let us derive, as an example, the type of the term that was given in example 3. Recall that this term represents function composition. In order to be concise, we will use Γ as a shorthand for the context $f : A \to B, g : B \to C, x : A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash g : B \to C}\text{Var} \quad \dfrac{\dfrac{}{\Gamma \vdash f : A \to B}\text{Var} \quad \dfrac{}{\Gamma \vdash x : A}\text{Var}}{\Gamma \vdash (f)\,x : B}\text{App}}{f : A \to B, g : B \to C, x : A \vdash (g)\,(f)\,x : C}\text{App}}{f : A \to B, g : B \to C \vdash \lambda x.\,(g)\,(f)\,x : A \to C}\text{Fun}}{f : A \to B \vdash \lambda g.\,\lambda x.\,(g)\,(f)\,x : (B \to C) \to A \to C}\text{Fun}}{\vdash \lambda f.\,\lambda g.\,\lambda x.\,(g)\,(f)\,x : (A \to B) \to (B \to C) \to A \to C}\text{Fun}$$

2.3 Curry-Howard Isomorphism

It has been made evident by Haskell Curry [Haskell B. Curry, Robert Feys 1958] and William Howard [William A. Howard 1980] that types can be considered as formulae, and that terms of the λ-calculus correspond to proofs of these formulae. If we compare the typing rules of the simply-typed λ-calculus, and the deduction rules for the logical implication together with the axiom rule, the correlation becomes obvious.

$$\dfrac{}{\Gamma, A \vdash A}\text{ax} \quad \Leftrightarrow \quad \dfrac{}{\Gamma \vdash x : \Gamma(x)}\text{Var}$$

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}{\to_i} \quad \Leftrightarrow \quad \dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.\, t : A \to B}\text{Fun}$$

$$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}{\to_e} \quad \Leftrightarrow \quad \dfrac{\Gamma \vdash u : A \to B \quad \Gamma \vdash v : A}{\Gamma \vdash u\,v : B}\text{App}$$

In order to recover the deduction rules starting from the typing rules, we only have to remove the terms of the λ-calculus. The relation between typing rules, deduction rules, terms and proofs goes even further: β-reduction can be given an interpretation in terms of cut-elimination for the implication symbol. What is called a cut in a proof is an occurrence of an introduction rule directly followed by an elimination rule. Some part of the proof can be moved in such a way that the proof is still valid, and that the symbol that is introduced and then eliminated no longer appears in the proof. We do not give the details of the cut-elimination procedure here, as it is not necessary for this work.

2.4 Extension to classical systems

In the previous section, we presented the set of rules for intuitionistic logic. Now, consider the following formula, which is usually referred to as the *law of the excluded middle*.

$$A \vee \neg A$$

Although it seems to be very simple and straight-forward, it cannot be proved using the rules of intuitionistic logic. The consequences are great, since this rule has been used in many mathematical reasonings ever since Aristotle. For instance, it forbids reasoning using the principle of *reductio ad absurdum*.

2.4.1 Classical Logic

Classical logic is an alternative system in which the *law of the excluded middle* is added as an axiom.

$$\frac{}{\Gamma \vdash A \vee \neg A}\,\text{LEM}$$

The difference between intuitionistic and classical logic is that the former enforces that we know from which branch of the proof-tree the proof comes. Let us consider the usual example of a classical proof of the following theorem.

Theorem 2.4.1 There exist $a, b \in \mathbb{R} \setminus \mathbb{Q}$ such that $a^b \in \mathbb{Q}$.

Proof. We know that $\sqrt{2}$ is irrational ($\sqrt{2} \in \mathbb{R} \setminus \mathbb{Q}$). Let us now consider the following number.

$$c = \sqrt{2}^{\sqrt{2}}$$

By the law of the excluded middle, it is either rational, or it is not rational (irrational).

- If it is rational, then we can take $a = b = \sqrt{2}$.
- If it is not rational, then let us take $a = c$ and $b = \sqrt{2}$, which will give us

$$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^{2} = 2$$

  which is indeed rational. □

Even if this proof shows that the theorem is correct, we do not know whether $\sqrt{2}^{\sqrt{2}}$ is rational or not. In fact, it is possible to show, not without difficulty, that it is irrational. There also exists a very simple intuitionistic proof for the theorem, using a cleverer counter-example ($a = \sqrt{2}$ an $b = \log_2(3)$).

There are alternative ways of recovering classical logic from intuitionistic logic that have been shown to be equivalent to adding the law of the excluded middle. One can for example introduce *Pierce's law* as an axiom,

$$\frac{}{\Gamma \vdash ((A \to B) \to A) \to A}\,\text{PL}$$

or add a rule for *Double negation elimination*.

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,\neg\neg_e$$

2.4.2 Curry-Howard Isomorphism for classical logic

Before the Nineties, it was believed among researchers that classical systems had no interesting algorithmic content. Everything changed when Timothy G. Griffin discovered in 1990 that the Curry-Howard isomorphism could be extended to classical logic through control operators [Timothy G. Griffin 1990]. Afterwards many other ways to perform this extension were proposed, like Michel Parigot's $\lambda\mu$-calculus [Michel Parigot 1992].

Let us consider the language Λ_{cc}, which is an extension of the λ-calculus with the control operator cc. It is generated by the following BNF grammar.

$$t ::= x \mid \lambda x.t \mid (t)\,t \mid cc \qquad\qquad x \in \Lambda_x$$

The type system of the simply-typed λ-calculus is extended with a rule, giving cc the type of Pierce's law.

$$\frac{}{\vdash cc : ((A \to B) \to A) \to A}\,cc$$

The semantics of terms containing control operators such as cc can be given in terms of evaluation contexts. Here we do not go into detail about this, since the operational semantics of Λ_{cc} will be given by the abstract machine that will be defined in the next section.

2.5 Krivine's classical realizability

In this section we give the interpretation of the type system presented in the previous sections in terms of classical realizability. We use the formalism introduced by Krivine [Jean-Louis Krivine 2009], and adapt it to our system. We consider the language Λ_c and the set of stacks Π, which are generated by the following mutually recursive grammars:

$$t ::= x \mid \lambda x.t \mid (t)\, t \mid cc \mid k_\pi \qquad \pi ::= \varepsilon \mid t \cdot \pi \qquad x \in \Lambda_x$$

The terms of Λ_c are those of Λ_{cc} extended with a constant k_π for every stack $\pi \in \Pi$. Stacks are simply lists of terms terminated by the empty stack constant ε.

A state of the Krivine's Abstract Machine (KAM) is called a process and is a couple of a term t and of a stack π. It is denoted $t \star \pi$, and the set of processes is denoted $\Lambda_c \star \Pi$. The execution of the machine is given by an evaluation relation $\succ$ defined over processes. It contains the following four rules.

$$\lambda x.t \star u \cdot \pi \succ t[x := u] \star \pi$$

$$(t)\, u \star \pi \succ t \star u \cdot \pi$$

$$cc \star t \cdot \pi \succ t \star k_\pi \cdot \pi$$

$$k_\pi \star t \cdot \pi' \succ t \star \pi$$

The classical realizability interpretation is parametrised by a pole $\perp\!\!\!\perp$, which is a set of processes closed under anti-evaluation (i.e. if $p \in \perp\!\!\!\perp$ and $p' \succ p$, then $p' \in \perp\!\!\!\perp$).

Definition 2.5.13 A formula (or type) τ is interpreted as two sets: a falsity value $\|\tau\| \subseteq \Pi$ and a truth value $|\tau| \subseteq \Lambda_*$. They are defined inductively on the structure of τ.

$$\|\tau_A \to \tau_B\| = |\tau_A| \cdot \|\tau_B\| = \{c \cdot \pi \mid c \in |\tau_A|, \pi \in \|\tau_B\|\} \qquad \|\varphi\| = \{\varepsilon\}$$

$$|\tau| = \|\tau\|^{\perp\!\!\!\perp} = \{c \in \Lambda_c \mid \forall \pi \in \|\tau\|\ c \star \pi \in \perp\!\!\!\perp\}$$

Definition 2.5.14 Given a pole $\perp\!\!\!\perp$, we say that a process p realizes a formula (or type) A, denoted $p \Vdash_{\perp\!\!\!\perp} A$, if $p \in |A|$.

One of the most interesting properties of classical realizability systems is that they are compatible in some sense with type systems. Roughly, what we mean is that when a proof-like term t has type T, then t should also realize T. This property is given more formally by the adequation lemma.

Theorem 2.5.2 Let $\perp\!\!\!\perp$ be a fixed pole, I be a finite family of index, and t be a proof-like term. If the judgement $(x_i : T_i)_{i \in I} \vdash t : T$ is valid in the type system, and $\forall i \in I \ t_i \Vdash_{\perp\!\!\!\perp} T_i$, then $t[x_i := t_i]_{i \in I} \Vdash T$.

3 An explicit environment Krivine machine

The Krivine's abstract machine usually used as a model for the evaluation of terms in classical realizability performs substitutions explicitly. When a β-redex is reduced, only a term remains, and the machine does not keep track of what variable was substituted, and by what term. This is not a problem in usual applications, however, when the terms that are evaluated contain holes, we need to know exactly which variables are in scope, and what values they have been assigned. Otherwise, holes cannot be filled using terms containing variables that are in scope.

In this section, we introduce a version of Krivine's abstract machine, which has the particularity of keeping the environment explicit by working on closures, and not on terms. Our variant is close to the original design of Krivine's abstract machine, but has, to our knowledge, never been used in classical realizability settings. It will hopefully offer a greater flexibility, since it will allow us to define operations modifying the environment directly (these aspects of the machine will be investigated in future works).

Explicit environments are convenient since values that are assigned to variables stay available throughout the evaluation. This feature will be used when generating values to fill holes. Another advantage of this system is that it is closer to what an actual implementation would look like (one would obviously not perform substitutions explicitly since it would be inefficient).

3.1 Environment machine

We consider the language Λ_{cc} defined in the previous chapter. It consists of the λ-calculus extended with the instruction cc. We recall that it is generated by the following grammar, where $\Lambda_x = \{x, y, z, ...\}$ is a countable set of variables:

$$t ::= x \mid \lambda x.t \mid t\,t \mid cc \qquad\qquad x \in \Lambda_x$$

Definition 3.1.1 A closure is a couple $\langle t, \sigma \rangle$ where t is a term and σ is an environment, which is a substitution which maps free variables of t to closures.

Intuitively, in order to build a closure starting from a term t with $FV(t) = \{x_1, \ldots, x_n\}$, we have to provide an environment $\sigma = \{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}$ where $c_1, \ldots, c_n$ are closures.

Definition 3.1.2 The set of free variables in a closure, and the set of free variables in a substitution are mutually inductively defined as follows:

$$FV(\langle t, \sigma \rangle) = (FV(t) \setminus dom(\sigma)) \cup FV(\sigma)$$

$$FV(\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}) = FV(c_1) \cup \ldots \cup FV(c_n)$$

Definition 3.1.3 A closure $\langle t, \sigma \rangle$ with $FV(\langle t, \sigma \rangle) = \emptyset$ is said to be closed.

Definition 3.1.4 The application of the substitution σ to the term t is denoted $t[\sigma]$. This operation is defined inductively on the structure of the term t.

$$x[\sigma] = \sigma(x) \qquad ((t)\,u)[\sigma] = t[\sigma]\,u[\sigma] \qquad cc[\sigma] = cc$$

$$(\lambda x.t)[\sigma] = \lambda x'.t'[\sigma] \text{ with } \lambda x.t \equiv_\alpha \lambda x'.t' \text{ and } x' \notin FV(\sigma)$$

We also consider the type-system of the simply-typed λ-calculus, with a rule giving the type of Pierce's law to cc. We recall that the set of all types $\mathcal{T}$ is generated by the following grammar, where $\mathcal{B}$ is a finite, but non-empty, set of atomic types.

$$\tau ::= \tau \to \tau \mid \varphi \qquad \varphi \in \mathcal{B}$$

A typing judgement will have the form $\Gamma \vdash t : \tau$, where t is a term, τ is a type, and Γ is a typing context mapping every free λ-variable in t to its type. The typing rules of the system are the following:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\,\text{Var} \qquad \frac{}{\Gamma \vdash cc : ((A \to B) \to A) \to A}\,cc$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,\text{Fun} \qquad \frac{\Gamma \vdash u : A \to B \quad \Gamma \vdash v : A}{\Gamma \vdash u\,v : B}\,\text{App}$$

We now extend the language with an additional set of constants, which will allow the machine to save the current state of its stack π, and restore it when the constant

associated to the stack k_π is encountered. The set of closed closures of the extended language will be denoted Λ_*. A stack is simply a list of closures, terminated by the empty stack constant ε. The set of stacks will be denoted Π. Note that the terms and stacks are mutually inductively defined.

$$t ::= \ldots \mid k_\pi \qquad\qquad \pi ::= \varepsilon \mid \langle t, \sigma \rangle . \pi$$

Definition 3.1.5 A process is a couple of a closed closure $\langle t, \sigma \rangle$ and a stack π, it is denoted $\langle t, \sigma \rangle \star \pi$. The set of all processes is written $\Lambda_* \star \Pi$.

The one-step evaluation relation $\succ$ over the set of processes is exhibited below.

$$\langle x, \sigma \rangle \star \pi \succ \sigma(x) \star \pi$$

$$\langle \lambda x.t, \sigma \rangle \star c . \pi \succ \langle t, \sigma + \{x \mapsto c\} \rangle \star \pi$$

$$\langle (t)\, u, \sigma \rangle \star \pi \succ \langle t, \sigma \rangle \star \langle u, \sigma \rangle . \pi$$

$$\langle \mathfrak{c}c, \sigma \rangle \star c . \pi \succ c \star \langle k_\pi, \sigma \rangle . \pi$$

$$\langle k_\pi, \sigma \rangle \star c . \pi' \succ c \star \pi$$

Note that the first three rules are those relating to the evaluation of terms of the λ-calculus, while the two last rules are those handling the classical part of computation, i.e. saving the execution context, and restoring it. The exact same remark can be made when looking at the evaluation relation of the Krivine's abstract machine presented in the last chapter. The main difference between the two abstract machines is that the one presented in this chapter has an additional rule to replace variables by the value they have in the environment. This rule is not required in setting with implicit environments since variables are substituted on the fly by their values.

3.2 Realizability

Relizability using a Krivine's abstract machine with explicit environments is very similar to the notion or realizability presented in the last chapter. This is not surprising

since the terms of the language coincide exactly. The realizability interpretation is again parametrized by a pole.

Definition 3.2.6 A pole is a set of processes $\bot\!\!\!\bot \subseteq \Lambda_* \star \Pi$ closed under anti-evaluation, i.e. for all $p, p' \in \Lambda_* \star \Pi$, if $p' \in \bot\!\!\!\bot$ and $p \succ^* p'$, then $p \in \bot\!\!\!\bot$.

Every type is interpreted as two sets: a set of stacks (falsity value) and a set of closures (truth value).

Definition 3.2.7 Every type τ is interpreted as a falsity value $\|\tau\| \subseteq \Pi$ and a truth value $|\tau| \subseteq \Lambda_*$ defined inductively on the structure of τ.

$$\|\tau_A \to \tau_B\| = |\tau_A| \cdot \|\tau_B\| = \{c \cdot \pi \mid c \in |\tau_A|, \pi \in \|\tau_B\|\} \qquad \|\varphi\| = \{\varepsilon\}$$

$$|\tau| = \|\tau\|^{\bot\!\!\!\bot} = \{c \in \Lambda_* \mid \forall \pi \in \|\tau\| \ c \star \pi \in \bot\!\!\!\bot\}$$

Definition 3.2.8 Given a pole $\bot\!\!\!\bot$, we say that a closure $\langle t, \sigma\rangle$ realizes its type τ, denoted $\langle t, \sigma\rangle \Vdash_{\bot\!\!\!\bot} \tau$, when $\langle t, \sigma\rangle \in |\tau|$. If $\langle t, \sigma\rangle \Vdash_{\bot\!\!\!\bot} \tau$ for all $\bot\!\!\!\bot$, we say that $\langle t, \sigma\rangle$ universally realizes the type τ. We denote this $\langle t, \sigma\rangle \Vdash \tau$.

We now give the adequation lemma for the new system. It states that the realizability interpretation is adequate with the type system, in the sense that if a term t has type τ in a context Γ, then a closure $\langle t, \sigma\rangle$, built by mapping variables of Γ to closures realizing their types, realizes the type τ.

Theorem 3.2.1 Let $\bot\!\!\!\bot$ be a fixed pole, I be a finite family of indices, $\Gamma = (x_i : \tau_i)_{i \in I}$ be a context and $\sigma = \{x_i \mapsto \langle t_i, \sigma_i\rangle\}_{i \in I}$ be a substitution. If the judgement $\Gamma \vdash t : \tau$ is provable using the deduction system, and if $\langle t_i, \sigma_i\rangle \Vdash_{\bot\!\!\!\bot} \tau_i$ for all $i \in I$, then $\langle t, \sigma\rangle \Vdash_{\bot\!\!\!\bot} \tau$.

Proof. We do a proof by induction on the length of the proof of $\Gamma \vdash t : \tau$. We consider the rule used in the last step of the proof.

- Var: we have $t = x_i$ for some fixed $i \in I$ and we must show that $\langle x_i, \sigma\rangle \Vdash_{\bot\!\!\!\bot} \tau_i$, which is equivalent to showing that $\langle x_i, \sigma\rangle \star \pi \in \bot\!\!\!\bot$ for every stack $\pi \in \|\tau_i\|$. By hypothesis we know that $\langle t_i, \sigma_i\rangle \Vdash_{\bot\!\!\!\bot} \tau_i$, which gives us $\langle t_i, \sigma_i\rangle \star \pi \in \bot\!\!\!\bot$ for every stack $\pi \in \|\tau_i\|$. We also know that $\langle x_i, \sigma\rangle \star \pi \succ \langle t_i, \sigma_i\rangle \star \pi$. Therefore, since $\bot\!\!\!\bot$ is closed under anti-evaluation, we have $\langle x_i, \sigma\rangle \star \pi \in \bot\!\!\!\bot$ for every stack $\pi \in \|\tau_i\|$.

- App: we have $t = u\,v, \Gamma \vdash u : \tau_A \to \tau_B$ and $\Gamma \vdash v : \tau_A$, and we must show that $\langle u\,v, \sigma\rangle \Vdash \tau_B$, which is equivalent to showing that $\langle u\,v, \sigma\rangle \star \pi \in \perp\!\!\!\perp$ for every stack $\pi \in \|\tau_B\|$. By induction hypothesis we have $\langle u, \sigma\rangle \Vdash \tau_A \to \tau_B$ and $\langle v, \sigma\rangle \Vdash \tau_A$, which are equivalent by definition to $\langle u, \sigma\rangle \in |\tau_A \to \tau_B|$ and $\langle v, \sigma\rangle \in |\tau_A|$. Moreover $\langle v, \sigma\rangle . \pi \in \|\tau_A \to \tau_B\|$ for all $\pi \in \|\tau_B\|$. This gives us $\langle u, \sigma\rangle \star \langle v, \sigma\rangle . \pi \in \perp\!\!\!\perp$ for every stack $\pi \in \|\tau_B\|$. We also have $\langle u\,v, \sigma\rangle \star \pi \succ \langle u, \sigma\rangle \star \langle v, \sigma\rangle . \pi$, hence, since $\perp\!\!\!\perp$ is closed under anti-evaluation we obtain that $\langle u\,v, \sigma\rangle \star \pi \in \perp\!\!\!\perp$ for every stack $\pi \in \|\tau_B\|$.
- Fun: we have $t = \lambda x.v$ and $\Gamma, x : \tau_A \vdash v : \tau_B$. We must show that $\langle \lambda x.v, \sigma\rangle \Vdash \tau_A \to \tau_B$, which is equivalent to showing that $\langle \lambda x.v, \sigma\rangle \star \pi \in \perp\!\!\!\perp$ for every stack $\pi \in \|\tau_A \to \tau_B\|$. By definition, a stack $\pi \in \|\tau_A \to \tau_B\|$ has the form $c . \pi'$, with $c \in |\tau_A|$ and $\pi' \in \|\tau_B\|$. By induction hypothesis we have $\langle v, \sigma + \{x \mapsto c\}\rangle \Vdash \tau_B$, which is equivalent to $\langle v, \sigma + \{x \mapsto c\}\rangle \star \pi' \in \perp\!\!\!\perp$ for every stack $\pi' \in \|\tau_B\|$. Hence, since $\langle \lambda x.v, \sigma\rangle \star c . \pi' \succ \langle v, \sigma + \{x \mapsto c\}\rangle \star \pi'$ and $\perp\!\!\!\perp$ is closed under anti-evaluation we obtain that $\langle \lambda x.v, \sigma\rangle \star c . \pi' \in \perp\!\!\!\perp$ for every stack $\pi = c . \pi' \in \|T_a \to T_b\|$.
- cc: we have $t = cc$ and we must show that $\langle cc, \sigma\rangle \Vdash ((\tau_A \to \tau_B) \to \tau_A) \to \tau_A$ which is equivalent to showing that $\langle cc, \sigma\rangle \star \delta \in \perp\!\!\!\perp$ for every stack $\delta \in \|((\tau_A \to \tau_B) \to \tau_A) \to \tau_B\|$. By definition, any such stack will have the form $c . \pi$, with $c \in |(\tau_A \to \tau_B) \to \tau_A|$ and $\pi \in \tau_A$. Since we know that $\langle cc, \sigma\rangle \star c . \pi \succ c \star \langle k_\pi, \sigma\rangle . \pi$, it is enough to show that $c \star \langle k_\pi, \sigma\rangle . \pi \in \perp\!\!\!\perp$ for every $\pi \in \tau_A$ and $c \in |(\tau_A \to \tau_B) \to \tau_A|$. To do so, it is enough to show that $\langle k_\pi, \sigma\rangle \in |\tau_A \to \tau_B|$ for all $\pi \in \tau_A$, which is equivalent to showing that $\langle k_\pi, \sigma\rangle \star \delta' \in \perp\!\!\!\perp$ for every stack $\delta' \in \|\tau_A \to \tau_B\|$. But every such stack has the form $c' . \pi'$ with $c' \in |\tau_A|$ and $\pi' \in \tau_B$, and $\langle k_\pi, \sigma\rangle \star c' . \pi' \succ c' \star \pi$, hence it is enough to show that $c' \star \pi \in \perp\!\!\!\perp$, which is true since $c' \in |\tau_A|$ and $\pi \in \|\tau_A\|$. □

As an immediate corollary, we get that if a term is typed in an empty context, then the closure built from the term with an empty environment realizes the type of the term.

Corollary 3.2.2 Let $\perp\!\!\!\perp$ be a fixed pole. If t is a term and τ is a type such that $\vdash t : \tau$, then $\langle t, \emptyset\rangle \Vdash \tau$.

Proof. Immediate from the adequation lemma. □

One of the greatest strengths of realizability is that the language of terms and the transition relation can be extended very easily, without losing any of the results obtained without the extension. In fact, the usual presentation of realizability is very parametric.

An axiomatization of the requirements gives rise to an infinite family or realizability models. Our realizability model can also be expressed in a parametric way, however, we chose to present only a specific model for the sake of simplicity and conciseness.

4 Realizability and holes

In this chapter we extend the system with a new kind of term that will be used to model holes, i.e. goals in proof assistants. These holes will have the particularity of carrying a type and a context, and will be called contextually typed holes (CTH). The type system will be extended, and we will need to adapt the adequation lemma. Finally, we will add reduction rules that will allow the holes to be filled during the computation. This process will act as the central part of the testing procedure.

4.1 Contextually typed holes and their substitution

Let $\Lambda_h = \{c, d, e...\}$ be a countable set of names with $\Lambda_x \cap \Lambda_h = \emptyset$. Elements of Λ_h will be used to identify holes.

Definition 4.1.1 A contextually typed hole (or CTH for short) is a new atomic term $[\Delta \vdash c : \tau]$, where Δ is a context, $c \in \Lambda_h$ is a name, and $\tau \in \mathcal{T}$ is a type. When the context and type associated to a CTH are not explicitly required we will omit them and write $[\Delta \vdash c]$, $[c : \tau]$ or simply $[c]$.

The language is now extended with CTH, which will, in some sense, play the role of meta-variables.

$$t ::= x \mid \lambda x.t \mid t\,t \mid \mathfrak{cc} \mid k_\pi \mid [\Delta \vdash c : \tau] \qquad x \in \Lambda_x, \pi \in \Pi, c \in \Lambda_h, \tau \in \mathcal{T}$$

Definition 4.1.2 The set of CTH appearing in a term t is defined inductively on the structure of t as follows:

$$\mathrm{CTH}(x) = \emptyset \qquad \mathrm{CTH}((t)\,u) = \mathrm{CTH}(t) \cup \mathrm{CTH}(u) \qquad \mathrm{CTH}(\lambda x.t) = \mathrm{CTH}(t)$$

$$\mathrm{CTH}(\mathfrak{cc}) = \emptyset \qquad \mathrm{CTH}([c]) = \{[c]\}$$

The CTH have an associated notion of substitution allowing the conversion of a term containing holes (i.e. CTH) into a term without holes.

Definition 4.1.3 A substitution candidate for a CTH $[\Delta \vdash c]$ is a term t such that $FV(t) \subset dom(\Delta)$. The set of substitution candidates for the CTH having the name c is denoted SC_c.

Definition 4.1.4 Given a term t, a hole substitution is a partial function $\Sigma : \Lambda_h \to \Lambda_{cc}$ mapping every CTH $[c]$ in t to a term $t_c \in SC_c$.

Intuitively, a hole substitution for a term t with $CTH(t) = \{c_i\}_{i \in I}$ (with I being a finite family of index) is a map $\{c_i \mapsto t_i\}_{i \in I}$ with $t_i \in SC_{c_i}$ for all $i \in I$.

Definition 4.1.5 We denote $t\Sigma$ the term t to which the hole substitution Σ has been applied. This operation is defined inductively on the structure of t.

$$x\Sigma = x \qquad (\lambda x.t)\Sigma = \lambda x.t\Sigma \qquad ((t)\,u)\Sigma = (t\Sigma)\,u\Sigma$$

$$cc\Sigma = cc \qquad [c]\Sigma = \Sigma(c)$$

4.2 Type of a hole and adequation lemma

We now extend the system with an additional typing rule, which will be responsible for giving CTH their respective types. There is only one constraint: the context of the CTH should be a subset of the global context. Intuitively, substitution candidates that will replace holes can contain free occurrences of the variables that are in the context of the CTH. Consequently, values for these variables should be available in the environment during computation.

$$\frac{\Delta \subset \Gamma}{\Gamma \vdash [\Delta \vdash c : A] : A}\,\text{Hole}$$

The adequation lemma for the new system can now be stated. It differs only slightly from the one for the system without holes. There are additional hypotheses, and a hole substitution is introduced and applied to the conclusion. The proof itself is mostly unchanged, only a new case appears for the typing rule of CTH.

Theorem 4.2.1 Let $\perp\!\!\!\perp$ be a fixed pole, I and K be two finite families of indices, $\Gamma = (x_i : \tau_i)_{i \in I}$ be a context, $\sigma = \{x_i \mapsto \langle t_i , \sigma_i \rangle\}_{i \in I}$ be a substitution, t be a term with $\mathrm{CTH}(t) = \{[\Delta_k \vdash c_k : \theta_k]\}_{k \in K}$ and $\Sigma = \{c_k \mapsto u_k\}_{k \in K}$ be a hole substitution. If the judgement $\Gamma \vdash t : \tau$ is provable using the deduction system, if $\langle t_i , \sigma_i \rangle \Vdash_{\perp\!\!\!\perp} \tau_i$ for all $i \in I$, and if $\Delta_k \subset \Gamma$ implies $u_k \in \mathrm{SC}_{c_k}$ for all $k \in K$, then $\langle t\Sigma , \sigma \rangle \Vdash_{\perp\!\!\!\perp} \tau$.

Proof. We do a proof by induction on the length of the proof of $\Gamma \vdash t : \tau$. The case for the rules Var, App, Fun and cc are very similar to the proof of the adequation lemma in the previous chapter, hence we will not give details. There is one new case in this rule since we have an additional typing rule:

- Hole: We consider $t = [\Delta_k \vdash c_k : \theta_k]$ for some $k \in K$. We have $\Delta_k \subset \Gamma$, and we must show that $\langle [c]\Sigma , \sigma \rangle \Vdash_{\perp\!\!\!\perp} \theta_k$, which is equivalent to showing that $\langle u_k , \sigma \rangle \Vdash_{\perp\!\!\!\perp} \theta_k$, which is true by hypothesis. □

We now have a realizability model that is sound with the type system extended with the rule for typing CTH. It remains to define the way in which holes will be reduced. Note that adding additional transition rules to the system will not change the adequation lemma. It is a very common practice in the field of classical realizability to prove an adequation lemma using a set of rules, and to work with an extended set of rules afterwards.

4.3 Reduction of CTH and plugs

A CTH is meant to be replaced by a term called a plug, during the computation process of the machine. A plug p is built using variables that are in the context, and possibly new CTH that would have to be replaced (or instantiated) later on.

Definition 4.3.6 We denote P_c the set of all plugs that are candidates for replacing the CTH $[\Delta \vdash c : \tau]$. Its definition depends on the structure of τ:

- if $\tau = \tau_A \to \tau_B$, then $P_c = \{\lambda x.[\Delta , x : \tau_a \vdash d : \tau_B]\}$ where $x \in \Lambda_x$ is a fresh variable (not appearing in Δ), and $d \in \Lambda_h$ is a fresh name.
- if $\tau = \varphi \in \mathcal{B}$, then $P_c = \{(x)[\Delta \vdash c_1 : \tau_1] \dots [\Delta \vdash c_n : \tau_n] \mid x : \tau_1 \to \dots \to \tau_n \to \varphi \in \Delta\} \cup \{cc\,[\Delta \vdash d : (\varphi \to \theta) \to \varphi] \mid \theta \in \mathcal{T}\}$.

Note that if the language contained typed constants, they could also be used for building plugs. In fact, the most important properties of plugs is that the replacing of a CTH by

a plug will always preserve the typing. Plugs should be defined in a way that respects typing.

Theorem 4.3.2 Given a CTH $[\Delta \vdash c : \tau]$, any term $p \in P_c$ satisfies the judgement $\Delta \vdash p : \tau$ (and by weakening any judgement $\Gamma \vdash p : \tau$ with $\Delta \subset \Gamma$).

Proof. Trivial (by definition of P_c). □

The reduction of CTH terms is defined in terms of a non-deterministic evaluation relation $\rightsquigarrow$ over the set of processes. The CTH that is in head position is substituted by a plug, and the same plug should also be used to substitute every occurrence of the same CTH in the environment and in the stack. Since we are in call-by-name settings, if we do not substitute every occurrence of a CTH at once, the CTH might be substituted by different plugs in different places, and this would mean that the language is no longer functional.

Definition 4.3.7 The substitution of a CTH $[c]$ by a plug $p \in P_c$ in a term, closure, environment, stack, is defined mutually inductively on the structure of the term, closure, environment, stack:

$$x[c := p] = x \qquad (\lambda x.t)[c := p] = \lambda x.t[c := p]$$

$$((t)\, u)[c := p] = (t[c := p])\, u[c := p] \qquad cc[c := p] = cc$$

$$[c][c := p] = p \qquad [d][c := p] = [d] \quad \text{with } d \neq c$$

$$[k_\pi][c := p] = k_{\pi[c := p]} \qquad \langle t, \sigma\rangle[c := p] = \langle t[c := p], \sigma[c := p]\rangle$$

$$\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}[c := p] = \{x_1 \mapsto c_1[c := p], \ldots, x_n \mapsto c_n[c := p]\}$$

$$(d \cdot \pi)[c := p] = d[c := p] \cdot \pi[c := p] \qquad \varepsilon[c := p] = \varepsilon$$

Definition 4.3.8 The evaluation relation $\rightsquigarrow$ is defined as follows:

$$\langle [c], \sigma\rangle \star \pi \rightsquigarrow \langle p, \sigma[c := p]\rangle \star \pi[c := p] \qquad p \in P_c$$

A full transition relation for the system can now be defined, which will include the transition rules of the initial transition relation, and also the rules of the relation $\rightsquigarrow$.

Definition 4.3.9 The full transition relation, denoted $\succcurlyeq$ is defined to be the union of the initial transition relation $\succ$ and $\rightsquigarrow$.

$$\succcurlyeq = \succ \cup \rightsquigarrow$$

We now want to show that the reduction of CTH terms always has a correct behaviour, i.e. that a CTH either reduces to a correct final state of the machine, or it instantiates new CTH forever (alternating finite sequences of reduction from the relation $\succ$, and single reductions using $\rightsquigarrow$).

Definition 4.3.10 We denote G the set of correct final states of the machine. For every final state $g \in G$, there should not exist any process p such that $g \succ p$.

Lemma 4.3.3 The relation $\succ$ is normalizing in typed settings (i.e. under the same assumptions as the adequation lemma).

Proof. Let I be a finite family of index. We consider a term t such that $(x_i : \tau_i)_{i \in I} \vdash t : \tau$, and an environment $\sigma = \{x_i \mapsto \langle t_i, \sigma_i \rangle\}_{i \in I}$ such that $\forall i \in I \ \langle t_i, \sigma_i \rangle \Vdash \tau_i$. Under these conditions, the adequation lemma gives us $\langle t, \sigma \rangle \Vdash \tau$. This is true in particular for the pole $\perp\!\!\!\perp = \{p \mid \exists n \in \mathbb{N} \ p \succ^n p' \in G\}$. We get that $\langle t, \sigma \rangle \Vdash_{\perp\!\!\!\perp} \tau$ which is equivalent to $\langle t, \sigma \rangle \star \pi \in \perp\!\!\!\perp$ for every $\pi \in \|\tau\|$. By definition of $\perp\!\!\!\perp$ such processes have a finite reduction to a good final state of the machine. □

Definition 4.3.11 We split the reduction of a CTH into three kinds of stages:

- computation: finite reduction from a process not having a CTH in head position to a process having a CTH in head position, using the relation $\succ$.
- generation: one-step reduction from a process having a CTH in head position using the relation $\rightsquigarrow$.
- result: finite reduction from a process not having a CTH in head position to a good final state of the machine using the relation $\succ$.

Definition 4.3.12 A testing tree is a potentially infinite tree modeling the possible input generations for a given test. Each leaf of the tree represents in fact a result stage.

Theorem 4.3.4 Any reduction of a CTH term has a correct behaviour, i.e. either it is an infinite alternation of generations and computations, or it is a finite alternation of generations and computations terminated by a result.

At the time of the writing, the proof of the last theorem is not finished. We start the proof by considering the following function F, and take as a pole its greatest fixed-point νF:

$$F(X) = \left\{p \mid p \succ^* p' \in G\right\} \cup \left\{p \mid p \succ^* \langle [c], \sigma\rangle \star \pi \wedge \forall p \in P_c, \langle a, \sigma\rangle \star \pi \in X\right\}$$

The pole $\perp\!\!\!\perp = \nu F$ is closed under anti-reduction. It remains to prove that a CTH realizes its type to complete the proof.

5 Testing procedure

In the previous two chapters we introduced a realizability model making use of a Krivine's abstract machine which had explicit environments. This system was extended with contextually typed holes which allowed us to represent holes in a term. In this chapter we are going to define the testing procedure. In particular we will show what preprocessing is required before being able to run a test, and we will provide some examples.

5.1 Notion of test

Our aim is to test typing judgements of the form $\Gamma \vdash t : \tau$, where t is a term containing two kinds of free variables:

- regular free variables to which Γ provides a type,
- and free goal variables that do not appear in Γ.

The goal type of the second set of variables should be provided by the user along with the judgement to test, in the form of an additional context Σ.

Definition 5.1.1 A judgement with goal is a quadruple of two contexts Γ and Σ with the condition that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Sigma) = \emptyset$, a term t with $\mathrm{FV}(t) \subset \mathrm{dom}(\Gamma \cup \Sigma)$, and a type τ. It is denoted $\Gamma \vdash_\Sigma t : \tau$.

Running a test on a judgement with goal $\Gamma \vdash_\Sigma t : \tau$ is done in three steps:

1. Preprocessing: the judgement is transformed into a term t' containing CTH, which is semantically equivalent to the judgement t in some sense. This process requires data from the context Σ, and some scope resolution into the term t.
2. Data generation: a substitution σ, mapping every variable in Γ to a closure that realizes its type, is generated, along with a stack $\pi \in \|\tau\|$.
3. Test run: the abstract machine is executed starting from the state $\langle t', \sigma \rangle \star \pi$.

The result of a test is to be interpreted in terms of the observable behaviour of the abstract machine, and more precisely, in terms of the state in which the machine stops (if it stops).

- If the machine stops in a correct final state, or does not stop, then the test is a success.
- If the machine stops in a bad final state, or if the instantiation of a plug fails (i.e. the set of plugs is empty) then the test fails.

Of course, if the machine does not stop by itself, we will have to interrupt it at some point. However, in that case, we will lose the information: we will never know whether the machine was going to stop in, say, ten thousands steps, or if it was going to run forever. This is one of the limits of the testing procedure.

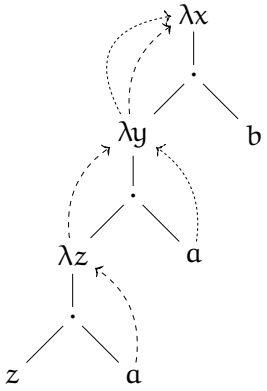
5.2 Preprocessing

Before being able to run the actual testing procedure, we have to change the data provided by the user into a term that will be executed in the abstract machine. Given a judgement with goal $\Gamma \vdash_\Sigma t : \tau$, we need to build from the term t, a term t' containing no more free goal variables, but that is, in some sense, semantically equivalent to t. The idea will be to replace goal variables with CTH.

Since a free goal variable x may appear more than once in the term t, we need to know the set of variables that are in scope at every occurrence of x in t, i.e. what variables will be available to be used when filling every hole corresponding to x.

Definition 5.2.2 The minimal scope of a free variable x in a term t is the intersection of the scopes at each occurrence of x in t. It is denoted $MS_x(t)$.

Intuitively, the minimal scope of a free variable corresponds to the set of λ-abstractions that are in the part that is common to the paths going from the root of the syntactic tree of the term to each occurrence of the variable. For example, in the term $(\lambda x.(\lambda y.(\lambda z\,.\, z\,a)\,a)\,b)\,c$, the minimal scope of variable a is $\{x, y\}$.



We can now formally define the operation consisting in getting a judgement with goal ready for testing.

Definition 5.2.3 Let $\Gamma \vdash_{(y_i : \theta_i)_{i \in I}} t : \tau$ be a typing judgement with goal where I is a finite family of index. The term that will be run into the abstract machine to test the judgement is $t' = t[\sigma]$ where $\sigma = \{\Delta_i \vdash c_i : \theta_i\}_{i \in I}$, and for all $i \in I$, Δ_i is a context mapping every variable in $MS_{y_i}(t)$ to its type.

5.3 Examples

Since the language and type-system we are considering is very simple (type-checking is decidable), it seems impossible to provide an example that is not trivial. We will however discuss a few examples if only to try to get some understanding about this new notion of test. Extensions of the system to more complex (and interesting) languages will be the subject of future work.

Let us consider, as a first example, the judgement with goal $\vdash_{h : \varphi} \lambda x.h : \varphi \to \varphi$. The preprocessing is very simple in the case of this example since there is only one free goal variable: h. The term to be run in the abstract machine is $t' = \lambda x.[x : \varphi \vdash c : \varphi]$. Since $\Gamma = \emptyset$, we have $\sigma = \emptyset$. We need to pick a stack $\pi \in \|\varphi \to \varphi\|$, which will be of the form $a \cdot \varepsilon$ with $a \in |\tau|$, by definition. The machine is then run from the state $\langle \lambda x.[x : \tau \vdash c : \tau], \emptyset \rangle \star a \cdot \varepsilon$, which results in the following computation.

$$\langle \lambda x.[x : \tau \vdash c : \tau], \emptyset \rangle \star_{\emptyset} a \cdot \varepsilon \quad \succ \quad \langle [x : \tau \vdash c : \tau], \{x \mapsto a\} \rangle \star_{\emptyset} \varepsilon$$

$$\rightsquigarrow \quad \langle x, \{x \mapsto a\} \rangle \star_{\{c \mapsto x\}} \varepsilon \quad \succ \quad a \star_{\{c \mapsto x\}} \varepsilon$$

The test is a success for every $a \in |\varphi|$.

Let us look at another example: $\vdash_{h : \varphi_C} \lambda f.\lambda x.h : (\varphi_A \to \varphi_B) \to \varphi_B \to \varphi_C$. After the preprocessing, the machine will start running in state $\langle \lambda f.\lambda x.[f : \varphi_A \to \varphi_B, x : \varphi_B \vdash c : \varphi_C], \emptyset \rangle \star a \cdot b \cdot \varepsilon$, where $a \in |\varphi_A \to \varphi_B|$ and $b \in \varphi_A$.

$$\langle \lambda f.\, \lambda x.\, [c], \emptyset \rangle \star a \cdot b \cdot \varepsilon \quad \succ \quad \langle \lambda x.\, [c], \{f \mapsto a\} \rangle \star b \cdot \varepsilon$$

$$\succ \quad \langle [f : \varphi_A \to \varphi_B, x : \varphi_B \vdash c : \varphi_C], \{f \mapsto a, x \mapsto b\} \rangle \star \varepsilon$$

This is an error case: the process cannot be reduced further since $P_c = \emptyset$. The failure was to be expected since there is no proof of $\vdash (A \to B) \to B \to C$. The corresponding testing tree is empty.

Let us now consider a judgement with goal that will generate a diverging test: $\vdash_{h:\varphi_A} \lambda f.h : (\varphi_A \to \varphi_A) \to \varphi_A$. The machine starts on the state $\langle \lambda f.[f : \varphi_A \to \varphi_A \vdash c : \varphi_A], \sigma \rangle \star a \,.\, \varepsilon$ where $\varepsilon \in |\varphi_A \to \varphi_A|$. In the case of this process, an infinite number of tests will be generated since the only variable in the context is $f : \varphi_A \to \varphi_A$, the only possible choice for generating a value of type φ_A is to apply f to a hole of type φ_A. This process will be repeated an infinite number of times. Intuitively, the term generated will have the form $(f)\,(f)\,(f)\,(f)\ldots$. Whatever the test that will be run on this term, the computation will not stop, hence we will not have any information about the term. Obviously, $\vdash (A \to A) \to A$ cannot be proved, therefore this example shows that the testing procedure is not complete. The corresponding testing tree is infinite and linear, it has only one infinite branch.

Alternatively, there are terms that can have diverging tests, but that are correct. Consider for example the judgement with goal $\vdash_{h:\varphi_A} \lambda f.\lambda x\,.h : (\varphi_A \to \varphi_A) \to \varphi_A \to \varphi_A$. Intuitively, the same infinite tests that were generated in the previous example can be obtained. However, there is an other way of generating a plug of type φ_A that will stop the computation on a success (i.e. the term generated will have the form $(f)\,(f)\,(f)\ldots x$). The testing tree has exactly one infinite branch, but at each note, there is a path of length one to a leaf.

6 Conclusion and future work

The outcomes that we can draw from this study are multiple, however we must balance them against the fact that, in the system we are using, types are decidable. Hence, Type-checking is a simpler and more powerful tool to check if a judgement is valid.

6.1 Main contribution

Giving an interpretation of the testing procedure in terms of Krivine's classical realizability allowed us to show that the notion of test is correct, in the sense that if all tests pass for a given judgement, then the judgement is true, and conversely, it a test fails, then the considered judgement is false.

In order to show this property, we used a Krivine's abstract machine which has the particularity of leaving the environment explicit. We also proved an adequation lemma for this machine, which, to our knowledge, had never been done before.

Moreover, through the use of the *call/cc* instruction, we extended the testing framework to classical systems. This extension was performed in a very natural way through the framework of classical realizability.

There remains plenty of ground for future exploration, and we have submitted an application for funding for a PhD thesis.

6.2 PhD Thesis project

The starting point of the thesis project will be to study the property of realizability when making use of a Krivine machine with explicit environments. The explorations done during this study seem to show that the machine with environments might be more general. One of our ideas is to try to use environments for modeling references in the language of terms.

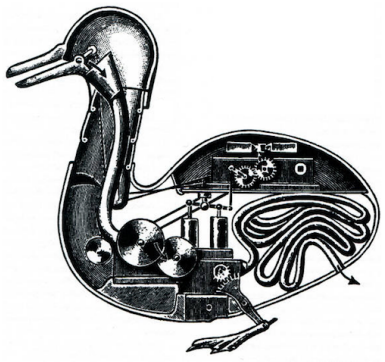
In a second part of the thesis we will extend the system to more expressive languages such as dependent type theories, for example. This will require us to define a notion of realizability for such systems.

6.3 Acknowledgements

First of all, I would like to thank my two advisors, Pierre Hyvernat and Christophe Raffalli, for their help and support throughout my internship. Thank you also to Alexandre Miquel for accepting to review my work, and to attend my official presentation as an external expert. Thank you to Karim Nour for accepting to be my advisor for my PhD thesis next year, together with Pierre Hyvernat. I would also like to address a special thank to my valued editor, and girlfriend, Branwen Chilton.

6.4 Disclaimer

This work was written using a new typesetting system called *Patoline* (patoline.com), which is an alternative to the TeX/LaTeX systems. This project was initiated at the LAMA, and it is still being developed currently. This document is one of the first to be completely written using *Patoline*, which is why we would like to apologise for any small typesetting errors that may be found in this document.



Bibliography

[Peter Dybjer 2010] Peter Dybjer, Program Testing and Constructive Validity, 2010

[Colin Runciman, Matthew Naylor, Fredrik Lindblad 2008] Colin Runciman, Matthew Naylor, Fredrik Lindblad, Smallcheck and lazy smallcheck: automatic exhaustive testing for small values, 2008

[Koen Claessen, John Hughes 2000] Koen Claessen, John Hughes, QuickCheck: a lightweight tool for random testing of Haskell programs, 2000

[Rodolphe Lepigre 2012] Rodolphe Lepigre, Testing Judgements of Type Theory, 2012

[Pierre Clairambault] Pierre Clairambault, Testing semantics for PCF

[Peter Dybjer, Pierre Clairambault 2012] Peter Dybjer, Pierre Clairambault, Testing semantics of dependent types, 2012

[William A. Howard 1980] William A. Howard, The Formulas-as-Types Notion of Construction, 1980

[Haskell B. Curry, Robert Feys 1958] Haskell B. Curry, Robert Feys, Combinatory Logic, Volume I, 1958

[Michel Parigot 1992] Michel Parigot, Lambda-mu-calculus: An Algorithmic Interpretation of Classical Natural Deduction, 1992

[Timothy G. Griffin 1990] Timothy G. Griffin, A Formulae-as-Types Notion of Control, 1990

[Jean-Louis Krivine 2009] Jean-Louis Krivine, Realizability in classical logic, 2009