Ecole normale supérieure de Lyon
Master 1 d'informatique fondamentale

Rapport de stage
Pierre Boutillier
Mai - Juillet 2009

Mathematically Structured Programming
Computer and Information Sciences
University of Strathclyde (Glasgow)

Equality for λ-terms with list primitives

Internship under the supervision of Conor McBride

Contents

Introduction

Formal proofs are really valuable for the guarantees they offer but who hasn't complained about time lost proving the obvious? More precisely, while good definitions of objects and properties are made, proving an equality often becomes finding the good variable to make an induction on and then doing some obvious rewriting. Rather than doing these annoying transformations by hand, we could try to find heuristics for finding and solving common cases but could we also define a reduction for both side of an equation that will give exactly the same normal form for the two terms if they are equal according a given equational theory?

For sure, turning an equational theory into a strongly normalizing rewriting rule system can only be made on really specific cases. Nevertheless, these cases can be really relevant for objects used in a strong normalizing way such as programs.

Work has been done to decide the *intensional equality* of programs : programs that compute the same way. Starting from here, to internalize specific laws for common operators is a tempting thing. But extending the rewrite rule system for evaluation makes it quickly unconfluent and some normalization is necessarily type-directed. Our key idea is consequently to normalize in two phases, first doing the ordinary computation then standardizing the presentation of terminated programs. Immediately, questions of how, how many and which rules can be added this way to get a extended equational theory that stay decidable are raised. More precisely, figuring out the completeness of the procedure with regards to the elementary transformation made in the standardisation phase is the ultimate goal. We'll try here to find clues to solve this new question by dealing with the example of lists.

My report has three parts. First, explanations about what I am dealing with and how we want things to go will be given. This part show the "state of the art" and the intuition my supervisor had to offer me this internship. Then I'll describe how works the implementation I've made to have a first look at if this idea works. This "concrete" step was also useful to understand deeply what appends. As a result, by experimentation, we became ready to make a formal proof of what we hope to get. This work, reported on the last part, is where new ideas really appear and are exposed.

Preliminary : Reading and understand a semantic rule

This whole report will deal with rules. So before beginning, it's necessary to explain what they tell and how they are made.

A rule has always the same form :

$$\frac{Hypothesis}{Conclusion}$$

Of course an axiom is a rule without hypothesis so here, you've got an :

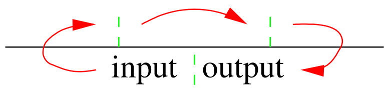
$$\frac{}{Axiom}$$

If there is more than one hypothesis they are separated by a big space

$$\frac{Like \quad that}{Something\ relevant}$$

In all rules, variables should have relevant names and follow intuition.

Moreover, when described rules are functional there are given to be naturally read in the way that computation is made. Left part of the conclusion are the input given to the first hypothesis which right part is its output that will be used by the second if its left side has input etc until conclusion of the last hypothesis of which is made the output of the conclusion (its right part).



1 Motivation, objectives and plan

1.1 A λ-calculus like Gödel's system T

In order to find results about programs, we need a mathematical model for them. We also want a system as small as will allow us to do our experiments. We'll use one of the standard one: the simply typed *λ-calculus*. In this model of the computation, programs are named *terms*. All terms are functions and calculating is applying a function to its argument. Moreover, terms behaviour is described by a *type*. Types are useful to ensure that we write programs that can be computed but moreover guarantee the termination of the computation. Here is a more precise description of how it works.

1.1.1 Types definition

A type can be a variable (Greek letters are often used to represent type variable) and an arrow between two types. $\sigma \to \tau$ represent for example functions that take an argument of type σ and return a term of type τ.

Stopping here could be enough. But, lists can't be expressed in a strict simply type system. We have to add more to get something like Gödel T system (λ-calculus with natural numbers).

Let us call:

ONE the type written **1** that has only one inhabitant

α β **PAIR** which has one term of α and one of β written $\alpha \times \beta$.

σ **LIST** written σ*List* to be either UNIT or $\sigma \times \sigma$ *List*.

1 *List* are exactly like natural numbers, Pairs are nothing more expressive too, [Miq] gives us system T equivalence.

$$\textsc{Ty} \quad := \quad \mathbf{1} \mid v \mid \textsc{Ty} \to \textsc{Ty} \mid \textsc{Ty} \times \textsc{Ty} \mid \textsc{Ty}\ \mathit{List}$$

Figure 1: Definition of types

1.1.2 Terms definition

Because we want to be able to ensure by an algorithm that terms behave as the programmer expects, we'll put some typing information in them. In fact, terms will be of two mutually defined kinds : on the one hand ***checkable terms*** that we can check against a given type and on the other hand ***inferable terms*** whose type can be infered relatively to a typing context which gives types for variables. This way of acting follows the bidirectional type checking process exposed by Pierce and Turner in [PT].

First of all:

A variable is an **inferable term** because the context gives type...

A functional abstraction noted $\lambda x.t$ type can also be checked if you give it an arrow type. This term links the occurrences of the variable x in the term t.

Moreover, giving an **inferable term** f and a **checkable term** t, we can build and ensure the behaviour of the

Elimination $f\ x$ of which we can infer type (from the eliminator one).

At last, of course an **inferable term** is a **checkable term** and if you give explicitly the type of an **checkable term** , this pair is an **inferable term** .

$$\begin{array}{rcl} \textsc{InfTm} & := & v \mid \textsc{InfTm}\ \textsc{Elim} \mid \textsc{ChkTm} : Ty \\ \textsc{ChkTm} & := & \lambda v.\textsc{ChkTm} \mid () \mid (\textsc{ChkTm}, \textsc{ChkTm}) \mid \underline{\textsc{InfTm}} \end{array}$$

Figure 2: Definition of **inferable term** and **checkable term**

In order to keep minds clear from now, syntactic terms are not all different. As you can expect $\lambda x.x$ is exactly the same than $\lambda y.y$. Declared variables (that we call *closed*) are consequently mute. We'll see in a moment how carrying this.

As for types, we could stop there but we shall add here for even more deep reasons that we'll see later primitives to build and use inhabitants of the types we've defined.

(a, b) is the **inferable term** made from the two **inferable terms** a and b. It represents the pair $(a\ ,\ b)$.

Eliminator are constructors which such as application destruct terms. This is the operation that represents calculating. We'll use at least :

app $x\ f$ to give an argument to a function.

first p to get the first part of a pair p

second p to get the second part of a pair p

$\mathbf{map}_\tau\ f\ l$ which apply the function f over each element of the list l. (For typing reason, we have to give the output type τ of f) and

$\mathbf{rec}_\tau\ m\ s\ l$ the primitive recursion over list: it gives back m if l is empty or compute **app** q (**app** h (**app** ($\mathbf{rec}_\tau\ m\ s\ t)\ s$)) if l is (h, t) (For the same typing reason, we have to give the output type τ)

$$\textsc{Elim} \ := \ \mathbf{rec}_{Ty}\ \textsc{ChkTm}\ \textsc{ChkTm} \mid \mathbf{first} \mid \mathbf{second} \mid \mathbf{map}_{Ty}\ \textsc{ChkTm} \mid \mathbf{app}\ \textsc{ChkTm}$$

Figure 3: Definition of Primitive elimination operators

Here is for the syntax. Typing rules follow exactly this so there is no interest in giving more explanation than the rules.

$$\textsc{TyCtxt} \;:=\; \varepsilon \mid v : \textsc{Ty}, \textsc{TyCtxt}$$

Figure 4: Definition of valid typing context

$$\boxed{\textsc{TyCtxt} \vdash \textsc{InfTm} \in \textsc{Ty}}$$

$$\frac{\Gamma, x:\tau, \Gamma' \vdash \mathit{tyCtxt}}{\Gamma, x:\tau, \Gamma' \vdash x \in \tau} \qquad \frac{\Gamma \vdash \tau \ni m}{\Gamma \vdash m : \tau \in \tau} \qquad \frac{\Gamma \vdash \sigma \ni t \quad \Gamma \vdash \sigma\ [e\rangle\ \tau}{\Gamma \vdash \mathit{Elim}\ t\ e \in \tau}$$

$$\boxed{\textsc{TyCtxt} \vdash \textsc{Ty}\ [\textsc{Elim}\rangle\ \textsc{Ty}}$$

$$\frac{\Gamma \vdash \sigma \ni t}{\Gamma \vdash \sigma \to \tau\ [\mathbf{app}\ t\rangle\ \tau} \qquad \frac{}{\Gamma \vdash \sigma \times \tau\ [\mathbf{first}\rangle\ \sigma} \qquad \frac{}{\Gamma \vdash \sigma \times \tau\ [\mathbf{second}\rangle\ \tau}$$

$$\frac{\Gamma \vdash \tau \ni m \quad \Gamma \vdash \tau \to \sigma \to \sigma\ \mathit{List} \to \tau \ni s}{\Gamma \vdash \sigma \mathit{List}\ [\mathbf{rec}_\tau\ m\ s\rangle\ \tau} \qquad \frac{\Gamma \vdash \sigma \to \tau \ni f}{\Gamma \vdash \sigma \mathit{List}\ [\mathbf{map}_\tau\ f\rangle\ \tau\ \mathit{List}}$$

$$\boxed{\textsc{TyCtxt} \vdash \textsc{Ty} \ni \textsc{ChkTm}}$$

$$\frac{\Gamma \vdash \mathit{tyCtxt}}{\Gamma \vdash \mathbf{1} \ni ()} \qquad \frac{\Gamma, x:\sigma \vdash \tau \ni t}{\Gamma \vdash \sigma \to \tau \ni \lambda x.t} \qquad \frac{\Gamma \vdash \sigma \ni s \quad \Gamma \vdash \tau \ni t}{\Gamma \vdash \sigma \times \tau \ni (s,t)} \qquad \frac{\Gamma \vdash t \in \tau}{\Gamma \vdash \tau \ni \underline{t}}$$

1.1.3 Semantics

It is the interesting part: one step of computation is defined by the *β-reduction*. This says that $\lambda x.t\ s$ β-reduce to $t[s/x]$ where $t[s/x]$ stands for the term t where every occurrence of x is replaced by the term s. Caution: we must be careful at this moment because, as in types, declared variables can have arbitrary names. Consequently if nothing is done after reduction, ambiguity can appear with what is called *variable capture* (take a look at $(\lambda x.\lambda y.x)\ (\lambda x.\lambda y.y) \longrightarrow \lambda y.\lambda x.\lambda y.y$ to see it). To maintain terms only readable one way, we must ask the programmer to use different names or to rename variable and deal with what is called α-conversion. In fact, the implementation part will show a standard way to avoid both of these procedures.

Of course, structural rules have to be added to the semantics to allow us to simplify subterms of terms but here again there is nothing requiring explanation to be understood. The problem is that they do not look functional at the first glance. The reason for this is that no priority is made between them but there is still a way to apply all of them which will be explained now.

1.2 Normalisation by evaluation

Our goal is to convert a semantic equality not apparently algorithmic into a syntactic decidable equality. We would like to be able to say that at least $\alpha\beta$-equivalent terms are equals. To do so, we need to normalize, which means finding a functional system of rules that convert any semantically equal terms in the same syntactic term. In this order, terms will be converted into value. Values have reduction rules over then both functional and compatible with the equational theory. So normalizing computation over values is made during this stage. This is what we call the *evaluation* step (noted $\Downarrow$). Then, from the computed value we read back a "normal" term: this phase is called *quotation* (noted $\Uparrow$). This principle to obtain decidability of equality by transforming objects into a relevant and computable representation is what is called *normalisation by evaluation.*

But what are in our case values ? Again, there are two kinds of values following the terms: values on which we hope to do more computation and *neutral terms* (corresponding to **inferable term**) under which no more reduction is possible and that can only become bigger and bigger as far as they are given to elimination terms.

To build values, we'll use the power of functional languages. λ-abstraction will be represented by higher order functions of the language and this way, computation is hidden by the language machinery. Moreover, we show a "big step" presentation of evaluation and we separate issues of termination, even through this language happens to b total.

$$\textsc{Env} \;:=\; \varepsilon \mid \textsc{Env}, v := \textsc{ChkTm}$$

Figure 5: Definition of environment

$$\boxed{\textsc{Env} \vdash \textsc{ChkTm} \Downarrow \textsc{ChkTm}}$$

$$\frac{\gamma \vdash env}{\gamma \vdash \lambda x.t \Downarrow \lambda[\gamma]x.t} \qquad \frac{\gamma \vdash env}{\gamma \vdash () \Downarrow ()} \qquad \frac{\gamma \vdash t \Downarrow v}{\gamma \vdash \underline{t} \Downarrow v} \qquad \frac{\gamma \vdash s \Downarrow s' \quad \gamma \vdash t \Downarrow t'}{\gamma \vdash (s,t) \Downarrow (s',t')}$$

$$\boxed{\textsc{Env} \vdash \textsc{Elim} \Downarrow \textsc{Elim}}$$

$$\frac{\gamma \vdash s \Downarrow \underline{s'}}{\gamma \vdash \mathbf{app}\ s \Downarrow \mathbf{app}\ s'} \qquad \frac{}{\gamma \vdash \mathbf{first} \Downarrow \mathbf{first}} \qquad \frac{}{\gamma \vdash \mathbf{second} \Downarrow \mathbf{second}}$$

$$\frac{\gamma \vdash f \Downarrow f'}{\gamma \vdash \mathbf{map}_\tau\ f \Downarrow \mathbf{map}_\tau\ f'} \qquad \frac{\gamma \vdash m \Downarrow m' \quad \gamma \vdash s \Downarrow s'}{\gamma \vdash \mathbf{rec}_\tau\ m\ s \Downarrow \mathbf{rec}_\tau\ m'\ s'}$$

$$\boxed{\textsc{Elim}\ @\ \textsc{ChkTm}\ \rightarrow\ \textsc{ChkTm}}$$

$$\frac{}{e\ @\ \underline{t}\ \rightarrow\ \underline{e\ t}} \qquad \frac{\gamma, x := s \vdash t \Downarrow v}{\mathbf{app}\ s\ @\ \lambda\ \gamma\ x.t\ \rightarrow\ v} \qquad \frac{}{\mathbf{map}_\tau\ f\ @\ []\ \rightarrow\ []}$$

$$\frac{f\ @\ h\ \rightarrow\ v \quad \mathbf{map}_\tau\ f\ @\ l\ \rightarrow\ w}{\mathbf{map}_\tau\ f\ @\ h :: l\ \rightarrow\ v :: w} \qquad \frac{}{\mathbf{rec}_\tau\ m\ s\ @\ []\ \rightarrow\ m}$$

$$\frac{\mathbf{rec}_\tau\ m\ s\ @\ l\ \rightarrow\ w \quad \mathbf{app}\ s\ @\ w\ \rightarrow\ w' \quad \mathbf{app}\ w'\ @\ h\ \rightarrow\ v' \quad \mathbf{app}\ v'\ @\ l\ \rightarrow\ v}{\mathbf{rec}_\tau\ m\ s\ @\ h :: l\ \rightarrow\ v}$$

$$\frac{}{\mathbf{first}\ @\ (s,t)\ \rightarrow\ s} \qquad \frac{}{\mathbf{second}\ @\ (s,t)\ \rightarrow\ t}$$

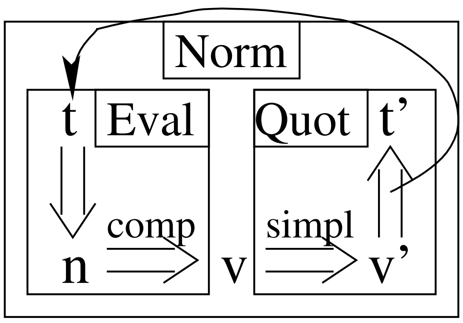
$$\boxed{\textsc{Env} \vdash \textsc{InfTm} \Downarrow \textsc{ChkTm}}$$

$$\frac{\gamma \vdash fG \Downarrow f' \quad \gamma \vdash s \Downarrow s' \quad f'\ @\ s'\ \rightarrow\ v}{\gamma \vdash f\ s \Downarrow v} \qquad \frac{\gamma, x := v, \gamma' \vdash env}{\gamma, x := v, \gamma' \vdash x \Downarrow v} \qquad \frac{\gamma, x, \gamma' \vdash env}{\gamma, x, \gamma' \vdash x \Downarrow \underline{x}}$$

$$\frac{\gamma \vdash t \Downarrow v}{\gamma \vdash t : \tau \Downarrow v}$$

As said before, structural rules such as $m\ n$ reduces to $m'\ n'$ if m reduces to m' and n to n' induce ambiguity. To compute them, priorities need to be established. But never mind the priority we've chosen because typed λ-calculus is strongly normalizing, so computation will be confluent and stop at the same value each time. An other hidden problem appears with the rule $\lambda x.m$ reduces to $\lambda x.m'$ if m reduces to m'. It amounts to a fonctional program rewriting its own code. Consequently, it is never done by the language however we need it to get a normal form. The solution is given by quotation. Quotation in its naive version is basially syntactic. But λ quotation is a bit more complex. In this case, a new "abstract" variable is created. (To create it, we'll use for example a counter or a name made from the one given to this declared variable by the uses if we've remembered it.) This created object is a concrete thing we can give as argument to the function. The β-reduction can consequently now be made under binders using it to complete evaluation.

In the end, the couple evaluation/quotation made reductions everywhere.



1.3 Type based clever "simplification" rule

From these good ideas, we would like to go further. We would like to extend not by adding more strongly normalizing process but by fully using the two phases of the process. In quotation, one more step can be made : it can be smarter by following types in order to express terms in a more regular way. We would like for example to figure out that if f is a functional term, it is equal to $\lambda x.f\ x$ (this rewriting is called *η-expansion*). These simplification are based on the property of Church presentation for the *lambda*-calculus that because types are put in the syntax and they are consequently unique. We are as a result allowed to use them as term exact description.

Lists and their primitives add even more non evaluating equality. Everyone is easily able to see that mapping the identity function, even if you have an abstract list, is useless or that because map deals with list elements separately, $map\ f\ (map\ g\ l) \equiv map(f \circ g)\ l$ for example.

To be sure that we won't miss a simplification, they will have to be done from the more elementary sub-term to the biggest one. Even if there are still critical pairs, we shall ensure that we see all the occurrences of transformable terms.

Quotation will now evaluate under λs but also doing *simplification.* Keep in mind that "simplification" here means "making more regular" and not necessary shorter. Developing has always been more syntactic than factorizing, therefore, we will orient equality rules in the easiest way for computing. For example, $\lambda x.x : (A \to A) \to (A \to A)$ is shorter but less regular than $\lambda x.\lambda y.(x : A \to A)\ y : (A \to A) \to (A \to A)$.

$$\boxed{\textsc{TyCtxt} \vdash \textsc{Ty} \ni \textsc{ChkTm} \Rightarrow \textsc{ChkTm}}$$

$$\frac{\Gamma \vdash l \Rightarrow l' \in \tau\ \mathit{List} \quad \Gamma \vdash \tau \to \tau \ni f \Rightarrow g \quad \Gamma \vdash \tau \to \tau \ni \lambda x.x \Rightarrow g}{\Gamma \vdash \mathbf{map}_\tau\ f\ l \Rightarrow l' \in \tau\ \mathit{List}}$$

$$\frac{\Gamma \vdash t \Rightarrow \mathbf{map}_u\ g\ l \in u\ \mathit{List} \quad \Gamma \vdash l \Rightarrow l' \in \sigma\ \mathit{List} \quad \Gamma \vdash \sigma \to \tau \ni \lambda x.f\ (g\ x) \Downarrow h}{\Gamma \vdash \mathbf{map}_\tau\ f\ t \Rightarrow \mathbf{map}_\tau\ h\ l' \in \tau\ \mathit{List}}$$

$$\frac{\overline{\Gamma} \vdash \mathbf{first}\ p \Downarrow s \quad \overline{\Gamma} \vdash \mathbf{second}\ p \Downarrow t}{\Gamma \vdash \sigma \times \tau \ni p \Rightarrow (s, t)} \qquad \frac{\overline{\Gamma, x : \sigma} \vdash \mathbf{app}\ x\ f \Downarrow v}{\Gamma \vdash \sigma \to \tau \ni f \Rightarrow v} \qquad \frac{\Gamma \vdash \mathit{tyCtxt}}{\Gamma \vdash \mathbf{1} \ni x \Rightarrow ()}$$

$$\boxed{\textsc{TyCtxt} \vdash \textsc{Ty} \ni \textsc{ChkTm} \Uparrow \textsc{ChkTm}}$$

$$\frac{\Gamma, x : \sigma \vdash \tau \ni v \Uparrow t}{\Gamma \vdash \sigma \to \tau \ni \lambda x.v \Uparrow \lambda x.t} \qquad \frac{\Gamma \vdash \mathit{tyCtxt}}{\Gamma \vdash \mathbf{1} \ni () \Uparrow ()}$$

$$\frac{\Gamma \vdash \sigma \ni s \Uparrow s' \quad \Gamma \vdash \tau \ni t \Uparrow t'}{\Gamma \vdash \sigma \times \tau \ni (s, t) \Uparrow (s', t')} \qquad \frac{\Gamma \vdash t \Uparrow t' \in \tau}{\Gamma \vdash \tau \mathit{List} \ni \underline{t} \Uparrow \underline{t'}}$$

$$\boxed{\textsc{TyCtxt} \vdash \textsc{InfTm} \Uparrow \textsc{InfTm} \in \textsc{Ty}}$$

$$\frac{\Gamma \vdash \tau \ni t \Uparrow t'}{\Gamma \vdash t : \tau \Uparrow t' : \tau \in \tau} \qquad \frac{\Gamma, x : \tau, \Gamma' \vdash \mathit{tyCtxt}}{\Gamma, x : \tau, \Gamma' \vdash x \Uparrow x \in \tau} \qquad \frac{\Gamma \vdash \sigma \,[f \equiv f'\rangle\ \tau \quad \Gamma \vdash s \Uparrow s' \in \sigma}{\Gamma \vdash f\ s \Uparrow f'\ s' \in \tau}$$

$$\boxed{\textsc{TyCtxt} \vdash \textsc{Ty} \,[\textsc{Elim} \equiv \textsc{Elim}\rangle\ \textsc{Ty}}$$

$$\frac{\Gamma \vdash \sigma \ni s \Uparrow s'}{\Gamma \vdash \sigma \to \tau \,[\mathbf{app}\ s \equiv \mathbf{app}\ s'\rangle\ \tau}$$

$$\frac{}{\Gamma \vdash \sigma \times \tau \,[\mathbf{first} \equiv \mathbf{first}\rangle\ \sigma} \qquad \frac{}{\Gamma \vdash \sigma \times \tau \,[\mathbf{second} \equiv \mathbf{second}\rangle\ \tau}$$

$$\frac{\Gamma \vdash \tau \ni m \Uparrow m' \quad \Gamma \vdash \tau \to \sigma \to \sigma\ \mathit{List} \to \tau \ni s \Uparrow s'}{\Gamma \vdash \sigma\ \mathit{List} \,[\mathbf{rec}_\tau\ m\ s \equiv \mathbf{rec}_\tau\ m'\ s'\rangle\ \tau}$$

$$\frac{\Gamma \vdash \sigma \to \tau \ni f \Uparrow f'}{\Gamma \vdash \sigma\ \mathit{List} \,[\mathbf{map}_\tau\ f \equiv \mathbf{map}_\tau\ f'\rangle\ \tau\ \mathit{List}}$$

Three things about simplification must be explained. For the reasons described above, it is a bottom up operation. It first tries to simplify subterms before trying to find if the term has one of the expected forms. As we can see with *map id* (*map f l*), critical pairs can appear too but evaluating the simplified terms will make result of the two strategies equal. Completeness proof has to say that this is always so. And for checking that we don't map the identity function for example, we need an equality. We'll use the syntactic equality over quoted terms to catch for example different forms of the same object. (Identity example will be more precisely described as an example of the implementation technique.)

Finding generic conditions for admissible simplification rule set question hasn't been raised yet but notice that, at least, they must take a neutral term and give a neutral term in order to be separated from the evaluation.

To finish with, naive formalisation of "the subterms can't be simplified" is not terminating each time (it is not strictly positive). But what happens is that quotation recursively does all the possible simplification. So it only represents the fact that we deal with quoted normal form for subterms. It is computable and consequently can be expressed in a strictly positive way.

2 Cooking up the normalisation algorithm

2.1 Variables representation

λ-calculus can have several representations well described in [Aug06]. The one given at the beginning is the easiest to understand for humans but, because of α-conversion, finding syntactic equal terms or β-reduce is difficult. Consequently, we may want to adopt what we call *Barendregt convention* where all declared variables have a different name. Substitution is now easier because ambiguities are avoided. But still the same term has several representations and we deal with variable name that means nothing for the machine. In a functional language, we can hide this behind the language interpreter and use language's higher order functions. By this way, computing is (for us) trivial even if we've lost equality checking in this case.

A solution commonly used is to represent variables as *deBruijn indices.* The deBruijn index of a variable is the number of λ to skip before getting the one that declares the variable. Let's be more concrete with two examples. $\lambda x.\lambda y.x\ y$ in the naive representation become $\lambda.\lambda.1\ 0$ because in $x\ y$, x is declared one λ further and y by the last λ seen. $\lambda x.x\ (\lambda y.y)$ is $\lambda.0\ (\lambda.0)$ because in the position where they respectably are , x and y both stand for the last variable declared. But because they aren't at the same place, a same number can represent different variables.

If in its representation, substitution of closed terms is straightforward and especially if terms have a canonical form, they are not really easy to read by a human being and having names depending of the position oblige to write really annoying functions to manipulate such terms (such as *shifting* free and closed variables).

Moreover, when terms have free variables (which is the case when we want to allow definitions to make the written terms simpler to read) deBruijn indexes require a closing strategy which represent that fact to add some λs before the term to represent free variables as its private parameters. This is one of the bigger source of mistake and you should better separate free and bound variables and define free variables by an explicit textual name. This is called the *locally nameless approach* from [MM04a].

Nevertheless, depending on what you want to do, the best (because in the end easiest thing to do) is to change representations, and not try to do something for which a representation is not made. Users will consequently be allowed to write naive terms. They'll read some under Barendregt convention in order to be sure that interpreters write output with no possible confusions. Computation will be done using CaML function and terms with deBruijn indexes will be compared.

By having a quick look at it, we discover that representation changes only deal with definitions, lambdas and variables. Therefore, we should make a generic function to follow the syntax in other cases and just give what to do in the three above situations. It will require an environment to keep in memory necessary stuff and can even be use to give back information about variables such as free variable list or what ever.

It declaration and type is

```
val var_care_inf f_def f_lam f_var env term :
  ('a -> name -> tType -> checkTerm -> infTerm -> infTerm) ->
  ('a -> name -> checkTerm -> checkTerm) ->
  ('a -> name -> tType -> infTerm) -> 'a -> infTerm -> infTerm
val var_care_check  f_def f_lam f_var env term :
  ('a -> name -> tType -> checkTerm -> infTerm -> infTerm) ->
  ('a -> name -> checkTerm -> checkTerm) ->
  ('a -> name -> tType -> infTerm) -> 'a -> checkTerm -> checkTerm
```

2.2 An interpreter in CaML

The software written in CaML aims for watching if every simplification is really made in non trivial cases. Interpreter is written following the clue of [MM04b]. This tool is absolutely not something built to be user friendly but you can still take a look at it. It's called `first` in byte-code mode with debugging information and `inter` in native code mode. (Mode is chosen from the first line of the Makefile). It takes an "Haskell like" text file as an input and answer in the standard output. More precisely, input is a succession of type then **checkable term** definition followed by an **inferable term** . In answers, variables are numbers but aren't deBruijn indexes at all, numbers are here nothing more than a trivial way to generate names ! They in fact represent the number binders seen before the one we are talking about and are known as *deBruijn levels*.

The core of the implementation is made as expected with two mutual definitions for each step. Here are the evaluation functions as an exemple. They carry two environments but no types because first bound variable definitions and free variable ones live in a different space and therefore can be just juxtaposed and second evaluation is an untype procedure; Types ensure terminaison of it from the outside.

```
val evalInf : Ttype.value list ->
  (Ttype.name * Ttype.value) list -> Ttype.infTerm ->
  Ttype.value
val evalCheck : Ttype.value list ->
  (Ttype.name * Ttype.value) list -> Ttype.checkTerm ->
  Ttype.value
```

Then the machinery to find equality over $\beta\eta\iota$-expansion is made in the same time as simplification and quotation because 1 needs 2 needs 3 needs 1.

```
val equiv_fun : int -> Ttype.tType -> Ttype.value ->
  Ttype.value -> bool
val simplify : int -> Ttype.neutral -> Ttype.neutral
val quoteValue : int -> Ttype.tType -> Ttype.value ->
  Ttype.checkTerm
val quoteNeu : int -> Ttype.neutral ->
  Ttype.infTerm * Ttype.tType
```

Even if the proof does not, the interpreter goes a bit further and has the `append` primitive over lists which makes a list from two lists by putting all the elements of the first one in the same order behind the element of the second. It has the properties that $append\ l\ Nil \equiv l$ and $map\ \tau\ f\ (append\ x\ y) \equiv append\ (map\ tau\ f\ x)\ (map\ \tau\ f\ y)$. Here again rewriting orientation is crucial because $map\ (+1)\ (append\ (map\ (+1)\ x)\ y) \longrightarrow append\ (map\ (+2)\ x)\ (map\ (+1)\ y)$ is really easier this way.

Now that we've got the structure, the best way to make progress is to show some of the examples we use to find how things have to work.

```
 (a * b) -> (a * b)                                  \1.(first(1),second(1))
 \x. x
 (a->a)->(a->a)
\x. x                                                       \ 1.\ 2.(1)2
```
First of all, generic function can have particular forms that we must recognize. We've work with this simpliest : the identity.
```
 (a List) List -> (a List) List
\ 1.map {a List} (\x. map {a} (\y. y) x ) (1)                        \ 1.1
```
Here, we've have been obliged to simplify the inner function to discover the identity.

```
 (c -> b) -> (b -> a) -> c List -> a List
\g f xs. (map {a} f (((\x. append                 \ 1.\ 2.\ 3.map {a}
(map {b} g x) ()) : c List -> b List) xs))        (\ 4.(2)(1)4) (3)
```
Then, succession of differrent simplification rules must be tested.
```
 let swap : (a * b) -> b * a
          = \ x. ( second x , first x )

let swap2 : (b * a) -> a * b
          = \ x. ( second x , first x )                              \ 1.1

(a * b) List -> (a * b) List
\ x. map {(a * b)} swap2 (map {(b * a)}
   swap x)
```
In the end, we've check the ability to find all simplifications from the bottom to the top of sub-terms with doing evaluation between two simplifications.

3 Formalizing the behaviour

3.1 Big step reduction without defining substitution

Normalisation seems to behave correctly as far as we have tested. The next step is to prove it. The tool to ensure proofs will be `Agda 2` ([Com]), a Swedish functional programming language with a dependent type checker based on Martin-Löf theory. To model the λ-calculus, we'll use a purely deBruijn indexed representation and omit definitions for simplicity. Nevertheless, we would like proofs to be as simple as possible and consequently the fewer operations over indexes we require, the better.

For β-reduction by substitution, terms are moved one into another and consequently operation over variables must appear. But we can compute without substitutions thanks to *environments*.

An environment explains how to interpret the free variables at the time a term is created with respect to the free variables at the time it is used. So, it is characterized by two natural numbers. The first argument tells the length of the environment and so how many different variables can have a term compute in this environment. The second tells the number of declared variables. As a result, it says how many different variables values compute in the environment will have.

With this new tool, life is easier. The value of variable i in the environment γ is the i-th element of γ if i is defined or itself if it is declared. β-reducing in an environment is also only defining the last variable with the value of the argument and computing the function in this new environment.

Only one operation over indices has to be defined: *weakening*. This means adding a new declared variable on top of an environment and consequently incrementing all indices in the corresponding terms, in order that they deal with the same old variables. We need it because computing under λ during quotation is exactly computing the term under the λ in the weakened environment. But we never go further than λ closure and consequently never insert a variable elsewhere in the environment which is "*thinning*".

Defining evaluation and primitive quotation (the one that follows terms) with this definition is obvious, you'll just have to be precise to keep in mind that evaluation does not compute under lambda but only keeps the closure of the term under the λ and the environment is ready for computation if an argument is given. The model used follows this principle.

Evaluation and naive quotation have been defined to be obviously deterministic even if the totality proof in agda are more complicated than that because as you know evaluation is sure to terminate only on typed terms and agda needs termination to be shown to stay consistent.

Moreover, the property "can not being simplified" can be expressed on a positive way and a simplification step terminates because it makes the lexical order "length of type", "length of terms" decrease so simplification and our type-directed quotation are just extra work between the two old phases which is trivially functional.

3.2 Type soundness for everyone

Types are the fuel consummation we constantly use to say that our relations are terminating functions. A primary work indeed was to check if all our transformations preserve types.

Types validity were one more time defined with regards to an typing *context.* It will be represented by a list of types where the ith element of the list gives the type of the ith deBruijn index.

Environment compatibility with context will have to be ensured first. This will just means that definition have types coherent with those given for the declared variables. In the same kind of question, thinning an environment must keep its elements well typed but everything is made in this purpose.

$$\boxed{Env :: TyCtxt \mapsto TyCtxt}$$

$$\frac{}{\varepsilon :: \Gamma \mapsto \Gamma} \qquad \frac{\gamma :: \Gamma \mapsto \Delta \quad \Gamma \vdash x \in \tau}{x, \gamma :: \Gamma \mapsto \tau, \Delta}$$

Here again, all elimination rules have a really similar behaviour over types: These elements all take a non elementary type and destruct it. For example $\sigma \times \tau$ [**first**⟩ σ. There is still no need to break the factorisation made during the term definition.

Evaluation, simplification and quotation must also preserve typing. For evaluation, typing rules are made in this goal. No problem must occur. Simplification followed the type structure, consequently no mad behaviour in type arrived. Naive part of the quotation is the only thing a bit more difficult. In fact, quotation of value takes a type to quote but quotation of neutral term (which represents **inferable term** remember) doesn't and returns back a term and a type. For this reason, we'll need a stronger recursion hypothesis to say that whatever the type given back, we will be able to take it to continue quotation and we'll find the same type that the one giving in input in the end.

Beside the induction, easy lemmas about for example variables or composition has been required but definition are made in order that everything goes directly well.

3.3 Equality and Soundness

Here is the time to formally express which terms we want to be equals. As for evaluating, we don't want to define substitution formally. We will consequently work with environments again. These environments are different of those for the evaluation one because they are environments of terms and not environments of values. Therefore, at one moment we'll have to define and use compatibility between representatives of each kind. After several failures, we've defined it this way: each term stored in the equality environment evaluates to the value stored at the same place in the evaluation environment.

With this definition, terms meaning depends on which definitions are stored in the environment they are associated with and the pertinent notion to define equality of it becomes the closure (*environment*!*term*).

Moreover, to catch η-expansion and rewriting for pairs, for example, equality have to interact with types. For this reason, the equality judgement carries a typing context of all the declared variables and a type that the two terms accept. (Of course, equal terms have the same types because we want to define computing in the same way which is stronger than computing the same thing and the second property is "having the same type".)

Now that form of judgement are exposed, there will be three kind of equality rules:

Structural rules to say that in different contexts, terms are equal if their subterms are.

Computational rules that expose one step of calculus in a given environment.

Simplification rules which describe the valid transformation of a term in an environment.

$$\boxed{TyCtxt \vdash Ty \ni (QEnv!ChkTm) \equiv (QEnv!ChkTm)}$$

$$\frac{}{\Gamma \vdash 1 \ni \underline{(\gamma!())} \equiv \underline{(\gamma'!())}} \qquad \frac{\Gamma \vdash \tau \ni (\gamma!t) \equiv (\delta!s)}{\Gamma \vdash \tau \ni (\delta!s) \equiv (\gamma!t)}$$

$$\frac{\Gamma \vdash \tau \ni (\gamma!r) \equiv (\delta!s) \quad \Gamma \vdash \tau \ni (\delta!s) \equiv (\theta!t)}{\Gamma \vdash \tau \ni (\gamma!r) \equiv (\theta!t)} \qquad \frac{\Gamma, \sigma \vdash \tau \ni (\gamma, x!t) \equiv (\gamma', x!t')}{\Gamma \vdash \sigma \to \tau \ni (\gamma!\lambda x.t) \equiv (\gamma'!\lambda x.t')}$$

$$\frac{\Gamma \vdash \sigma \ni (\gamma!f) \equiv (\gamma'!f') \quad \Gamma \vdash \tau \ni (\gamma!s) \equiv (\gamma'!s')}{\Gamma \vdash \sigma \times \tau \ni (\gamma!(f, s)) \equiv (\gamma'!(f', s'))} \qquad \frac{\Gamma \vdash \tau \ni (\gamma!t) \equiv (\gamma'!s)}{\Gamma \vdash \tau \ni \underline{(\gamma!t : \tau)} \equiv \underline{(\gamma'!s : \tau)}}$$

$$\frac{\Gamma \vdash \sigma \ni \underline{(\gamma!s)} \equiv \underline{(\gamma'!s')} \quad \Gamma \vdash \sigma \, [(\gamma!f) \equiv (\gamma'!f')\rangle \; \tau}{\Gamma \vdash \tau \ni \underline{(\gamma!f \; s)} \equiv \underline{(\gamma'!f' \; s')}}$$

$$\boxed{TyCtxt \vdash Ty\,[(QEnv!ChkTm) \equiv (QEnv!ChkTm)\rangle\ \ Ty}$$

$$\frac{}{\Gamma \vdash \sigma \times \tau\,[(\gamma!\mathbf{first}) \equiv (\gamma'!\mathbf{first})\rangle\ \ \sigma} \qquad \frac{}{\Gamma \vdash \sigma \times \tau\,[(\gamma!\mathbf{second}) \equiv (\gamma'!\mathbf{second})\rangle\ \ \tau}$$

$$\frac{\Gamma \vdash \sigma \to \tau \ni (\gamma!f) \equiv (\gamma'!f')}{\Gamma \vdash \sigma\,[(\gamma!\mathbf{app}\ f) \equiv (\gamma'!\mathbf{app}\ f')\rangle\ \ \tau}$$

$$\frac{\Gamma \vdash \sigma \to \tau \ni (\gamma!f) \equiv (\gamma'!f')}{\Gamma \vdash \sigma List\,[(\gamma!\mathbf{map}_\tau\ f) \equiv (\gamma'!\mathbf{map}_\tau\ f')\rangle\ \ \tau List}$$

$$\frac{\Gamma \vdash \tau \ni (\gamma!m) \equiv (\gamma'!m') \quad \Gamma \vdash \sigma \to \sigma List \to \tau \to \tau \ni (\gamma!s) \equiv (\gamma'!s')}{\Gamma \vdash \sigma List\,[(\gamma!\mathbf{rec}_\tau\ m\ s) \equiv (\gamma'!\mathbf{rec}_\tau\ m'\ s')\rangle\ \ \tau}$$

$$\boxed{TyCtxt \vdash Ty \ni (QEnv!ChkTm) \equiv (QEnv!ChkTm)}$$

$$\frac{\gamma :: \Gamma \mapsto \Delta \quad \Delta \vdash \tau \ni t}{\Gamma \vdash \tau \ni (\gamma!t) \equiv (\gamma!t)}$$

$$\frac{}{\Gamma \vdash \tau \ni \underline{(\gamma!(\lambda x.t)\ s)} \equiv \underline{(\gamma, x = s!t)}} \qquad \frac{}{\Gamma \vdash \sigma \ni \underline{(\gamma!\mathbf{first}\ (f,s))} \equiv \underline{(\gamma!f)}}$$

$$\frac{}{\Gamma \vdash \tau \ni \underline{(\gamma!\mathbf{second}\ (f,s))} \equiv \underline{(\gamma!s)}} \qquad \frac{}{\Gamma \vdash \tau List \ni \underline{(\gamma!\mathbf{map}_\tau\ f\ [])} \equiv \underline{(\gamma![])}}$$

$$\frac{}{\Gamma \vdash \tau List \ni \underline{(\gamma!\mathbf{map}_\tau\ f\ (h,t))} \equiv \underline{(\gamma!(\mathbf{app}\ f\ h, \mathbf{map}_\tau\ f\ t))}}$$

$$\frac{}{\Gamma \vdash \tau \ni \underline{(\gamma!\mathbf{rec}_\tau\ m\ s\ [])} \equiv \underline{(\gamma!m)}}$$

$$\frac{}{\Gamma \vdash \tau \ni \underline{(\gamma!\mathbf{rec}_\tau\ m\ s\ (h,t))} \equiv \underline{(\gamma!\mathbf{app}\ t\ (\mathbf{app}\ h\ ((\mathbf{app}\ \mathbf{rec}_\tau\ m\ s\ t)\ s)))}}$$

$$\boxed{TyCtxt \vdash Ty \ni (QEnv!ChkTm) \equiv (QEnv!ChkTm)}$$

$$\frac{\gamma :: \Gamma \mapsto \Delta \quad \Delta \vdash \sigma \to \tau \ni f}{\Gamma \vdash \sigma \to \tau \ni (\gamma!\lambda x.f\ x) \equiv (\gamma!f)} \qquad \frac{\gamma :: \Gamma \mapsto \Delta \quad \Delta \vdash \sigma \times \tau \ni p}{\Gamma \vdash \sigma \times \tau \ni (\gamma!(\mathbf{first} p, \mathbf{second} p)) \equiv (\gamma!p)}$$

$$\frac{}{\Gamma \vdash \tau\ List \ni \underline{(\gamma!\mathbf{map}_\tau\ \lambda x.x\ l)} \equiv \underline{(\gamma!l)}}$$

$$\frac{}{\Gamma \vdash \tau\ List \ni \underline{(\gamma!\mathbf{map}_\tau\ f\ \mathbf{map}_u\ g\ l)} \equiv \underline{(\gamma!\mathbf{map}_\tau\ f \circ g\ l)}}$$

Rules has been made in order to ensure that a term is equal to the quotation of its evaluation. Nevertheless, the property giving the *Soundness* of our normalisation is not that obvious. In fact, it presents two difficulties.

First, evaluation has a tree structure whereas equality proof is a list of one step rules linked by transitivity property. This means that terms of proof will be long and not obvious to read but moreover a lot of effort in this deconstruction has to be archived to find when to do thing on the right time. This mean that when you are still in the right context to have the piece of information you need and before other rules break the term structure. It also means that because equality rules are only for terms you need a way to deal with value compute and so never quote on witch you need equivalent.

The second point is that after evaluation two situations can occur for values. They can be quoted and in this case we would like them to be equal to the initial terms but they can also be eliminated by computation with other values and in this case soundness over the resulting values must be guaranteed too. This implies a proof in two parts with each time lemmas to find the authorized elimination and giving the right recursive call to say their soundness.

Moreover, the application rule is not elementary at all. It corresponds to three steps for equality: to introduce the definition, then, remplace variable by definition in all the terms which make the weakened term, and then, thin to get the evaluate term. Several new intermediate values has consequently to be explicitly defined to succeed.

Soundness of our procedure has been cut in two parts, we first deal with the naive untyped quotation, then we add simplification and raise cases where more stuff is done by quotation. The hardest part is the first one and the second one has a lot of cases.

Conclusion

My internship began with the study of the intuition of my supervisor found during the development of Epigram that transformation over neutral terms we want to do can be delayed to after the normal computation. This work rely even more over types of terms that are already fondamental in other way.

It has also been started with a totally different formallism by [BS06] but trying to make a formal proof of the properties of our semantic gives us new ideas to formalize more easily λ-calculus behaviour.

The consequence of the internship is canvas to implement equality decision procedure by normalisation by evaluation for more than β-equivalence. But more revelant, we also obtain first bricks to study formaly the λ-calculus and normalisation in two phases in Agda.

Nevertheless, we are only at the very beginning of this adventure and formal proof of completeness is not complete. Lists are simple data structure good for a first test but can we generalize data structure we are working with? Can we keep our result with more expressive types system or for dependant types? In the end, we would also like to find generic conditions over rules to say that we can apply our stategy in two phases and obtain decision for the equality.

References

[ACP$^+$08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, NY, USA, 2008. ACM.

[Aug06] L. Augustsson. λ-calculus cooked four ways. code avalaible, 2006.

[BS06] Freiric Barral and Sergei Soloviev. Inductive type schemas as functors. In *CSR*, pages 35–45, 2006.

[Com] The Agda Community. *The Agda Wiki*.

[Miq] Alexandre Miquel. Théorie de la démonstration. lecture slides.

[MM04a] Conor McBride and James McKinna. Functional pearl: i am not a number–i am a free variable. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9, New York, NY, USA, 2004. ACM.

[MM04b] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[PT] Pierce and Turner. Bidirectional type checking. to be completed.