Proof automatization using reflection (implementations in Agda)

M1 internship under T. Altenkirch's supervision

G. Allais

June 6, 2010

Abstract

This paper brings together an introduction to Agda and a presentation on how to extend Agda with solvers in order to improve automatization. The first part will describe the basic features of Agda through the implementation of an interpreter for a simply-typed lambda-calculus with base types (natural numbers and booleans), product and recursion. The second part will discuss the use of reflection to design solvers and compare it to other approaches. We will study the implementation of a solver for propositional logic as an example.

All the Agda files mentioned in this paper can be found on the darcs repository of this project[1]. To be able to compile these files, it is necessary to run a very recent version of Agda (a development version [4] compiled after the 11^{th} AIM).

1 Motivations

Agda has been defined in Ulf Norell's thesis [7] less as a theorem prover than as a programming language using dependent types whose theory is based on Luo's UTT [6]. The proofs are therefore functions (terms) constructed by the user rather than tactics combinations as in Coq [1]. But this does not imply that Agda is only a kind of Automath [2] with a layer of syntactic sugar: automatization is central in Agda (interactive development of the program, inference of implicit arguments and placeholders, verification of totality, etc.) even if it can still be improved.

Using Agda to develop certified programs forces one to use the entire power of dependent types and to prove various auxiliary lemmas: one may want to prove equivalences in order to replace terms by their counterparts, preservation of properties in order to certify the correction of an algorithm, etc. Spending time on simple but tedious proofs rather than on the actual difficulties such as your algorithm, its invariant, the formalisation of the notions needed for the specification for example can be very frustrating.

Designing tools to discharge the programmer's duty to prove theorems whose provability is known to be decidable is a necessity if we want Agda to be widely

[1] Directory `rls1/` : `https://patch-tag.com/r/gallais/agda/`

used. This policy has begun with the development of the ring solver thanks to Nils Danielsson's modules [3] based on their Coq equivalents.

Until now, Agda had no tactic language (unlike Coq and Ltac) and the use of solvers did not seem to be encouraged: there was no way to use the goal as an argument when calling a function. To use the goal in your developments, you had to quote it and modify it by hand to match the data structures you were using in your solver. Fortunately, the latest version of Agda[2] is allowing automatic goal quoting which is a strong argument in favor of the use of solvers.

Part I
An introduction to Agda

2 Definitions

Agda has been thought of more as a functional programming language using dependent types than a theorem prover. As a consequence, the way functions are defined and properties are proved is closer to the functional programming traditions than to other theorem provers such as Coq.

2.1 Datatypes

There exist two main datatype shapes: `data` and `record`. Inductive and co-inductive types use the same keyword `data` and are distinguished only by the way constructors are designed : they can take either finite derivations or infinite ones. The constructor `record` corresponds obviously to the tuples in a programming language.

Notations and definitions are melted in Agda: the character underscore is used to declare the spots where the arguments of the function (or constructor) will be inserted. You can, for example, define the set of typing inferences with the identifier "$\_\vdash\_:\_$" which will take a context Γ, a term t and a type τ as arguments ($\Gamma \vdash t : \tau$).

An example of an inductive datatype: the representation of the types of our lambda-terms and the definition of a context (a list of types).

```
data Ty : Set where
  bool : Ty
  nat : Ty
  base : ℕ → Ty
  _⇒_ : Ty → Ty → Ty
  _⊗_ : Ty → Ty → Ty

data Con : Set where
  ϵ : Con
  _∘_ : Con → Ty → Con
```

[2]The development version that you can get on the darcs repository [4].

As we are dealing with dependent types, an inductive datatype can obviously be indexed by other types. We can (for example) consider the set of terms typed with respect to a context.

We added the structural rule of weakening to the common ones in order to be able to construct terms by hand quite easily.

```
data Term : Con → Ty → Set where
  weak : ∀ {Γ σ τ} → Term Γ τ → Term (Γ ∘ σ) τ
  (...)
  app : ∀ {Γ σ τ} → Term Γ (σ ⇒ τ) → Term Γ σ → Term Γ τ
  lam : ∀ {Γ σ τ} → Term (Γ ∘ σ) τ → Term Γ (σ ⇒ τ)
  rec : ∀ {Γ σ} → Term Γ (σ ⇒ σ) → Term Γ (σ ⇒ σ) →
            Term Γ (nat ⇒ (σ ⇒ σ))
  (...)
  branch : ∀ {Γ σ} → Term Γ bool →
           Term Γ σ → Term Γ σ → Term Γ σ
```

Once we have defined the set of environments (an environment is a vector of values indexed by a context), we are ready to write our interpreter.

```
data Env : Con → Set where
  ϵ : Env ϵ
  _::_ : ∀ {Γ σ} → Val σ → Env Γ → Env (Γ ∘ σ)
```

2.2 Pattern matching

Once a set has been defined using `data` (or `record` if a constructor has been declared), it is possible to pattern match on it.

A function is defined in two steps: first we explicitly write the type of the function and then we write the body of the function. The interactive mode allows us to avoid rewriting everything (automatic destruction of an argument). Here is an example of a function declaration[3]: the function `_!!_` is able to return a value of type σ if it is provided with an environment and an index p such that the pth element of the environment has type σ.

```
_!!_ : ∀ {Γ σ} → Env Γ → Var Γ σ → Val σ
ρ !! p = {!!}
```

This example is a great way of showing that pattern matching in a language with dependent types is more complicated than in a traditional programming language. Considering that types can be terms over types, we can see that knowing the shape of an argument can influence the value (or the type) of other arguments.

When we define `_!!_`, pattern matching on Γ will also give us information on ρ because ρ's type is indexed by Γ: if Γ is empty, then ρ is empty too and if Γ has exactly n elements, then ρ has exactly n elements.

The empty pattern is denoted by `()` and is used to discard cases where the type is empty. If this emptiness is not trivial then the user may have to assume that an inhabitant exists and to use `⊥-elim` on the right hand of the equation[4].

[3]The pattern `{!!}` means that the body of the function still needs to be filled.

[4]Definitions in Agda are very similar to the ones in mathematics. The equational representation (pattern matching on the left) makes it easier to read the program and understand what it is supposed to do.

Example: definition of `_!!_`. There exists a far less verbose definition (pattern matching on p for example) but this one highlights the dependencies that appear during pattern matching.

```
_!!_ : ∀ {Γ σ} → Env Γ → Var Γ σ → Val σ
_!!_ {ϵ} {σ} ρ ()
_!!_ {Γ ∘ .σ} {σ} (y :: ρ) zero = y
_!!_ {Γ ∘ τ} {σ} (_ :: ρ) (suc n) = ρ !! n
```

First we destruct the context Γ (implicit argument). If the context is empty, then `Var Γ σ` cannot be inhabited (empty pattern). Otherwise we can destruct the index p and the environment ρ.

If p equals 0 then the context Γ *has to be* of the shape `Γ' ∘ .σ`[5] and we can output its first element. Else we do a recursive call (without precising the implicit arguments which will be inferred).

2.3 File checking

Agda's checking has two phases: if it manages to parse the file, it typechecks the definitions and, then, checks the termination of the functions defined in the file. Agda's termination check succeeds if and only if there are only structurally decreasing recursive calls.

The implementation of the interpreter for our typed lambda-calculus is pretty straightforward but for one problem: we had to treat the recursion case in a dedicated auxiliary function in order to help the termination checker to see that the evaluation was clearly ending.

We ended up with two mutually recursive functions:

```
mutual
  [|_|] : ∀ {Γ σ} → Term Γ σ → Env Γ → Val σ
  (...)

  Rec : ∀ {Γ σ} → ℕ → Term Γ (σ ⇒ σ) →
        Term Γ (σ ⇒ σ) → Env  Γ → Val σ → Val σ
  (...)
```

As the termination check does not fail we can say that all the functions that we can write using these constructors will be terminating ones.

This result has to be compared to the implementation of an untyped lambda-calculus with recursion without restrictions where the termination check fails (and that is completely normal because we can write $\delta = \lambda x.xx$).

3 Our interpreter in action

3.1 Expressivity

3.1.1 Primitive recursion

The light version of `rec` that is used in our language is still powerful enough to express the proper rec_2 operator used in computability theory [8].

[5] A doted value means that this value is forced by the context: here the fact that `Var Γ σ` equals 0 means that the first element of the context Γ is an inhabitant of σ.

$$rec_2\ n\ s\ z = \begin{cases} if & n = suc\ n' & then & s\ n\ (rec_2\ n'\ s\ z) \\ & & else & z \end{cases}$$

The term is not so nice because we have to handle pairs to propagate the value of the recursive call to be able to simulate rec_2 using only `rec` so we will not include it here. Here is a formal description of how it works[6]: we write a function outputting $(rec_2\ n\ s\ z, n)$ when given s, z and n.

$$\begin{aligned} f_{s,z}(0) &= (rec_2\ 0\ s\ z, 0) \\ &= (z, 0) \end{aligned}$$

$$\begin{aligned} f_{s,z}(Suc\ n) &= (rec_2\ (Suc\ n)\ s\ z, Suc\ n) \\ &= (s\ n\ (rec_2\ n\ s\ z), Suc\ n) \\ &= (s\ (proj_2\ f_{s,z}(n))\ (proj_1\ f_{s,z}(n)), Suc\ (proj_2\ f_{s,z}(n))) \end{aligned}$$

This function can be written using `rec`. Deducing rec_2 is trivial :

$$rec_2\ n\ s\ z = proj_1\ \{f_{s,z}(n)\}$$

3.1.2 Examples

We have proved that our recursion operator is equivalent to the common one (the reciprocal statement is trivial). As a consequence we can write all the primitive recursive functions using our operators. As an example, we implemented a (non efficient) exponentiation:

```
mul : ∀ {Γ} →  Term Γ (nat ⇒ (nat ⇒ nat))
mul = lam (lam (rec₂ (var (suc zero))
      (lam (lam (add (var (suc (suc zero))) (var zero))))
      (const 0)))

exp : ∀ {Γ} → Term Γ (nat ⇒ (nat ⇒ nat))
exp = lam (lam (rec₂ (var zero)
      (lam (lam (app (app mul (var (suc (suc (suc (zero))))))
      (var zero)))) (const 1)))

2⁵ : ℕ
2⁵ = [| app (app exp (const 2)) (const 5) |] ϵ
```

The normalization of 2^5 obviously leads to 32.

4 Conclusions

4.1 A real programing language

Agda's syntax using equations is very close to the way mathematicians declare a piecewise-defined function which makes the code easier to read. The use of

[6]The actual implementation is available on the darcs repository: `rls1/lambda-interp/untyped.agda`.

utf8 and the possibility of defining mixfix operators leads also to more easily human legible code.

Agda's specifications imply that it is possible to use it as a traditional functional programming language: Agda lets you write non-terminating programs and can even ignore the termination check failure[7]: one can write an interpreter for an untyped lambda-calculus if one want to and one will be able to define Ω and try to interpret it (knowing that `C-g` stops the computation is quite useful).

Examples[8]:

```
δ : Term
δ = Lam (App (Var 0) (Var 0))

Ω : Term
Ω = App δ δ
```

But it also provides the developper with tools (dependent types, termination check) to be able to check that a program does what it is supposed to do and to be able to prove properties of the program. The developpers can choose the amount of certifications that they want for their programs (termination? Computed values' properties? Fully specified program?) depending on the amount of time they are willing to spare for the proofs' development.

4.2 A lack of automatization

Agda's standard library provides the functional programmers with almost everything they could dream of. But the mathematicians are not as privileged as the functional programmers: they have to develop a lot of very basic proofs by hand even when the problem is known to be decidable.

There is a real lack of automatization for this kind of very simple (but numerous when you implement a non trivial theorem) proofs.

The second part of this paper will describe how to improve automatization by writing certified tactics for Agda in Agda.

Part II
Reflection

There are 3 ways to implement decision procedures in theorem provers. One can either rely on unverified code, use tactics written in a Meta-Language or use reflection. An oracle in unverified code will be easy to write but relying on it may be problematic (because it is unverified). Using a Meta-Language will guarantee that if the proof is accepted by the type checker, then it is correct. It will be a bit more tedious to write it (especially if the Meta-Language is designed in a not-so-handy way). A solver written using reflection will be much harder to write but it will be possible to prove that it really solves a whole set of problems.

[7] Agda's behaviour can be modified file by file by adding option pragmas at the top of each file.

[8] An interpreter for an untyped lambda calculus can be found on the darcs repository: `rls1/lambda-interp/untyped.agda`.

4.3 Concept, advantages & disadvantages

The main idea behind reflection is being able to manipulate terms of the language in the language itself. Writing a solver which will use reflection is pretty simple: one need only a couple of basic things.

1. *Representing the terms* To manipulate terms you need to be able to represent them. You will have to implement:

   a datatype `MyType` matching the set of problems you will be solving;

   a datatype `MyTerm` matching the set of constructors that you need to construct your proofs (`MyTerm` may depend on `MyType` and other variables in order to add constraints to the inductive construction);

   a function `quoteGoal` able to quote the current goal into `MyType`

2. *Solving the problem* You will consider the terms of `MyType` as types and construct (in `MyTerm`) inhabitants of these types (if it is possible)

3. *Proving the goal* To be able to go back to your proof, you need to define the semantics of your types ($[|.|]_\tau$) and terms ($[|.|]_T$) and to prove that these semantics are sound.

$$\frac{T : \tau}{[|T|]_T : [|\tau|]_\tau}\ \text{Soundness}$$

This approach has a couple of very interesting advantages compared to the use of unverified code or tactics written in a Meta-Language.

First of all, it allows you to develop certified solvers : instead of being unable to distinguish a randomly generated tactic from a smartly implemented one, you can prove that your solver really does what it is supposed to do. This can be done by proving that your algorithm is a decision procedure for the set of problems that you are looking at.

Secondly, the development of your tactics can benefit from the power of the dependent types: the type constraints prevent you from introducing bugs and, as a consequence, having to debug your tactic which can be very complicated and unpleasant[9].

As a consequence, the implementation of the solver can be quite tricky or long because you're not only constructing a proof of your goal: you have to prove things about the datatypes that you are using while constructing your proof.

4.4 Implementation in Agda

Reflection is already used by the ring solver which proves equivalences on a ring. The solver normalizes both the left and the right hand side of the equality (the normalization process is proved to preserve the equality) and then uses the reflexivity of equality.

The solver is a bit annoying to use because you have to quote the goal by hand and modify the constructors in order to make them match the solver's

[9]It should be pretty obvious to anyone that has ever tried to develop a non-trivial tactic using Ltac.

input datatype constructors. Here is an example of a theorem that is proved using the ring solver:

```
ex₅ : ∀ m n → m * (n + 0) ≡ n * m
ex₅ = solve 2 (λ m n → m :* (n :+ con 0)  :=  n :* m) refl
```

Since the 11^{th} Agda Implementors Meeting, the development version of Agda comes with two new constructs: `quoteGoal_in_` and `quote`. `quoteGoal G in E` allows you to use the variable `G` -whose value will be the quoted goal- in the expression `E`[10]; `quote x` gives you the internal representation of the name `x` if `x` is the identifier of a definition (function, datatype, etc.).

It is now very simple to code a function that will automatically transform a quoted term into an input term for the solver. We can imagine that the interface provided will allow the user to write only:

```
ex₅ : ∀ m n → m * (n + 0) ≡ n * m
ex₅ = quoteGoal t in solve t refl
```

5 A solver for propositional logic

5.1 Introduction

The Curry–Howard correspondence tells us that proving a proposition `P` of minimal logic intuitionistically is exactly the same problem as finding an inhabitant of the type `P` in the simply-typed lambda calculus. We decided to extend minimal logic to propositional logic by studying simply-typed lambda calculus with product and sum. The statement we will be able to solve therefore has the following shape:

$$\tau \ ::= \ \bot \mid atom\ \mathbb{N} \mid \tau \to \tau \mid \tau \wedge \tau \mid \ \tau \vee \tau$$

5.2 Theoretical basis

Our work is mainly based on Roy Dyckhoff's paper [5] describing a set of special deduction rules for intuitionistic logic. The particularity of these rules is that the context is always structurally decreasing. As Agda's termination checker is based on structural recursion, we can imagine that they would be the ideal candidates for our implementation.

Common rules The rules dealing with the goal are the common ones: axiom, false elimination, splitting or branching depending on the context and the shape of the context.

$$\frac{}{\Gamma, P \vdash P}\ \text{var} \qquad \frac{}{\Gamma, \bot \vdash P}\ \bot\text{-use}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \cap B}\ \text{and} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \text{lam}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \cup B}\ \text{inj1} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \cup B}\ \text{inj2}$$

[10] Typing rule of `quoteGoal_in_` : t [x := 'A] : A ⊢ `quoteGoal` x `in` t : A

Context simplification The context simplification rules are quite unusual in order to guarantee that the proof-tree can only be of finite height.

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \cup B \vdash C}\ \text{case} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, (A \cap B) \vdash C}$$

$$\frac{\Gamma, at\ n, A \vdash B}{\Gamma, (at\ n \supset A), at\ n \vdash B}\ \text{ModusP} \qquad \frac{\Gamma, A \supset (B \supset C) \vdash D}{\Gamma, (A \cap B) \supset C \vdash D}\ \text{curry}$$

$$\frac{\Gamma, B, C \supset A \vdash C \qquad \Gamma, A \vdash D}{\Gamma, (B \supset C) \supset A \vdash D} \qquad \frac{\Gamma, A \supset C, B \supset C \vdash D}{\Gamma, (A \cup B) \supset C \vdash D}$$

Unfortunately this is not structural in the sense of Agda. A recursive call is structurally decreasing in Agda if and only if the arguments of the recursive call are subterms of the arguments of the initial one. Here is a very simple example for which the termination checker fails to see that the computation will end.

```
foo : List ℕ → ℕ
foo [] = 0
foo (zero :: xs) = foo xs
foo (suc n :: xs) = foo (n :: n :: xs)
```

There exist at least two very simple (from a human point of view) proofs of the termination of this function: the first one treats the list as a multiset and the second one gives a simple measure function : $m(L) = 3^{hd(L)} + m(tl(L))$ where $m([]) = 0$.

Fortunately a trick to solve our problem exists: Dyckhoff's paper gives us a weight function which decreases as the proof goes on. We used a mixed approach relying on structurally decreasing calls when it was possible and this weight function when we were forced to.

5.3 Description of the solver

5.3.1 Datatypes

The datatype describing our goals is the exact translation of the goals' grammar (given in 5.1). A type depends on n the number of `Set`s that are used in the goal[11].

```
data Type : ℕ → Set where
  atom : ∀ {n} → Fin n → Type n
  ⊥ : ∀ {n} → Type n
  _∩_ _⊃_ _∪_ : ∀ {m} → Type m → Type m → Type m
```

A context is a vector of types. We obviously provide the user with a function `Get p from` Γ which gets the p^{th} element of Γ (p's type is naturally `Fin n` where n is Γ's length).

[11] If p has the type `Fin n` then: $p \in [|0; n-1|]$. See the module `Data.Fin` of the standard library.

```
data Con : (n : ℕ) → ℕ → Set where
  ϵ : ∀ {n} → Con n 0
  _#_ : ∀ {n l} → Con n l → Type n → Con n (Suc l)
```

A term is typed with respect to a context. The context is extended under a lambda abstraction and the variables' range is limited by the context's size. A term of type `Term ϵ t` has therefore no free variable.

```
data Term {n l : ℕ} (Γ : Con n l) : Type n → Set where

-- basic constructors
  var : (y : Fin l) → Term Γ (Get y from-Con Γ)
  app : ∀ {t1 t2 : Type n} → Term Γ (t1 ⊃ t2) →
        Term Γ t1 → Term Γ t2
  lam : ∀ {t1 t2 : Type n} → Term (Γ # t1) t2 →
        Term Γ (t1 ⊃ t2)
  ⊥-use : ∀ {t : Type n} → Term Γ ⊥ → Term Γ t

-- and constructors
  _and_ : ∀ {t1 t2 : Type n} → Term Γ t1 →
          Term Γ t2 → Term Γ (t1 ∩ t2)
  proj1 : ∀ {t1 t2 : Type n} → Term Γ (t1 ∩ t2) → Term Γ t1
  proj2 : ∀ {t1 t2 : Type n} → Term Γ (t1 ∩ t2) → Term Γ t2

-- or constructors
  inj1 : ∀ {t1 t2 : Type n} → Term Γ t1 → Term Γ (t1 ∪ t2)
  inj2 : ∀ {t1 t2 : Type n} → Term Γ t2 → Term Γ (t1 ∪ t2)
  case : ∀ {t1 t2 t3 : Type n} → Term Γ (t1 ∪ t2) →
         Term Γ (t1 ⊃ t3) → Term Γ (t2 ⊃ t3) → Term Γ t3
```

5.3.2 Algorithm

There are two steps in the inference procedure. These two steps corresponds to the two sets of rules described previously and they are implemented in two mutually recursive functions.

Goal simplification During the first step the algorithm deconstructs the goal: it introduces the arguments in the context (while lifting it) until there is no argument left, splits the goal into two goals if there is a product type and branch if there is a sum type. If a trivial solution exists (i.e. either an inhabitant of the goal or of the empty type ($\bot$) is in the context) it stops and outputs the proof. If not, the algorithm goes to the second step.

Context simplification During the second step, the algorithm tries to simplify the context by using the inference rules described in the previous section. If it manages to find an element in the context that can be simplified than it goes back to the first step with a lighter context. If none of the elements can be simplified, the algorithm just fails (the type has no inhabitant).

5.4 Guarantees

Unlike solvers written in a Meta-Language or in another Language, our solver comes with guarantees on the work which is done.

5.4.1 Free from free variables terms

As the solver is run with an empty context at the beginning, it guarantees that every single variable is referring to a binder encountered before. This is done using the `Fin` library[12]: a term in a context Γ of length n can only refer to a variable using a label in `Fin n`.

5.4.2 Well-typed terms

The terms of type `Term` are well-typed by construction: the inductive constructors are designed in a way that forces this well-typedness. Here are the typing rules equivalent to these constructors:

$$\frac{|\Gamma'| = p - 1}{(\Gamma \sharp \sigma) \sharp\sharp \Gamma' \vdash \texttt{var}\ p : \sigma}\ \texttt{var} \qquad \frac{\Gamma \vdash t : \bot}{\Gamma \vdash \bot\texttt{-use}\ t : \tau}\ \bot\texttt{-use}$$

$$\frac{\Gamma \sharp \sigma \vdash t : \tau}{\Gamma \vdash \texttt{lam}\ t : \sigma \supset \tau}\ \texttt{lam} \qquad \frac{\Gamma \vdash t_1 : \sigma \supset \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash \texttt{app}\ t_1 t_2 : \tau}\ \texttt{app}$$

$$\frac{\Gamma \vdash t_l : \sigma \qquad \Gamma \vdash t_r : \tau}{\Gamma \vdash t_l\ \texttt{and}\ t_r : \sigma \cap \tau}\ \_\texttt{and}\_$$

$$\frac{\Gamma \vdash t : \sigma \cap \tau}{\Gamma \vdash \texttt{proj1}\ t : \sigma}\ \texttt{proj1} \qquad \frac{\Gamma \vdash t : \sigma \cap \tau}{\Gamma \vdash \texttt{proj2}\ t : \tau}\ \texttt{proj2}$$

$$\frac{\Gamma \vdash t_l : \sigma_l \supset \tau \qquad \Gamma \vdash t_r : \sigma_r \supset \tau \qquad \Gamma \vdash b : \sigma_l \cup \sigma_r}{\Gamma \vdash \texttt{case}\ b\ t_l\ t_r : \tau}\ \texttt{case}$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \texttt{inj1}\ t : \sigma \cup \tau}\ \texttt{inj1} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \texttt{inj2}\ t : \sigma \cup \tau}\ \texttt{inj2}$$

As a counterpart, we had to add a lifting function to preserve the well-typedness when we introduce a new variable in the environment. This function corresponds to the following typing rule:

$$\frac{\Gamma \vdash T : \tau}{\Gamma, \sigma \vdash T \Uparrow : \tau}\ \Uparrow$$

The implementation is straight-forward: you only need a function lifting the free variables of a given term. As we use de Bruijn indexes, it is equivalent to implement a function protecting the k first variables where k is the number of λ-abstractions preceding the current term. It forces us to prove by induction a couple of results on appended lists but there is no major difficulty.

[12] A term of type `Fin n` is in the range $[|0; n-1|]$

5.4.3 Soundness

We gave the semantics of `Type` (`[|_|]`) and `Term` (`interp`)and we proved that it is sound i.e. that:

$$\forall t : \texttt{Term}\ \epsilon\ \tau, \forall \rho, \texttt{interp}\ t\ \rho : [|\tau|]\rho$$

5.5 Interface

Partiality The function transforming a quoted term into a `Type n` term is partial because the goal may not have the right shape. The inference procedure is also partial because it may not be possible to solve the quoted goal. Our function has to be total to be definable in Agda and to have a type that will match the current goal.

To tackle this issue, the interface takes an argument whose type will depend on the result of the inference procedure. If the procedure manages to output a proof then this argument will have the $\top$ type (which has only one inhabitant and this term can be inferred by Agda). Otherwise, the user will have to provide proof of the goal.

Description The interface has been developed using the latest features of Agda: a function quotes the goal into a `RawType` term which is then transformed in a `Type n` term where n is the number of `Set`s we refer to. Using the solver is very simple: a simple call to the `solve` function with the number of variables and a vector containing these variables is sufficient. An additional argument treats the case where the solver is not able to prove the goal (eg. the goal is not provable): in this case, the user has to provide the solver with a solution.

A modification of Agda's code which would provide an `unquote` function would even allow the user to simply call the `solve` function without the need for a vector of variables as an argument.

Examples The underscore means that Agda has to infer the corresponding placeholder. As these goals can be solved, Agda does not complain: it manages to find out that the placeholder is `tt` (the only constructor for $\top$). If it were impossible to solve one of the goals, Agda would of course complain (and ask the user to fill in the blank).

```
ModusP : ∀  {A B C : Set} → B → (B → A) → A
ModusP {A} {B} {C} = quoteGoal t in
                     solve 3 t (A :: B :: []) _

⊃⊃-simpl : ∀ {A B C D : Set} → (B → (C → A) → C) →
           (A → D) → ((B → C) → A) → D
⊃⊃-simpl {A} {B} {C} {D} = quoteGoal t in
               solve 4 t (A :: B :: C :: D :: []) _
```

A whole description of the different steps of the procedure can be found in the `rls1/prop_logic/Examples.agda` file. It uses the theorem `∀ {A B C : Set} → (A → ⊥) → B → A → C` as an example and shows the successive transformations[13].

[13] One can inspect a term value by normalizing it : `C-c C-n NAME RET` in emacs.

5.6 Why not a decision procedure?

As we explained before, Agda allows you to choose how much certification you want on your programs. Developing a decision procedure instead of implementing an inference algorithm would have been much more complicated because you would have had to formalize the whole meta-theory (the typing rules), to explicitly manipulate the contexts and to be able to generate counter examples when the goal was not provable.

This work would be much more complicated but it might be interesting to try to do it because it would guarantee that our solver is not only correct but also complete.

6 Conclusion and perspectives

6.1 First solver using reflection

This solver is the very first example of the use of reflection together with the implementation of a proper quoting function in Agda. It is both a proof of concept and a first step in a direction where automatization will be more important.

It also underlines the fact that Agda is a real programming language (the main functions would not have been much more easier to write in a traditional programming language) and that the developer can benefit from the use of dependent types (no free variable, well-typedness by construction, etc.).

6.2 A solver for Presburger arithmetic

This first step and the strong signals in favor of more automatization are encouraging us to pursue in this direction. Our next aim (which will occupy us during the next months) is the conception and the implementation of a solver for a much more complex but still decidable problem: we will work on a decision procedure for Presburger Arithmetic with quantifiers. Some theorem provers such as Coq or HOL already tackle this well-known problem but this will be the first solver to really use reflection[14].

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[2] N. de Bruijn. *Automath Archive - Website*. URL: `http://www.win.tue.nl/automath/`.

[3] Nils Anders Danielsson. *A ring solver for Agda*. Apr. 2009. URL: `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.UsingTheRingSolver`.

[14] There is an implementation of a solver using " reflection " in HOL but the computations are actually done using a code generated from HOL functions. The result of these computations is *accepted* as an equality proof. A proper solver using reflection should not need the implementation of this kind of hack in the source language.

[4] Agda development team development team. *Development version - darcs repository.* URL: `http://code.haskell.org/Agda`.

[5] Roy Dyckhoff and Sara Negri. "Admissibility of Structural Rules for Contraction-Free Systems of Intuitionistic Logic". In: *J. Symb. Log.* 65.4 (2000), pp. 1499–1518.

[6] Zhaohui Luo. "A Unifying Theory of Dependent Types: The Schematic Approach". In: *TVER '92: Proceedings of the Second International Symposium on Logical Foundations of Computer Science.* London, UK: Springer-Verlag, 1992, pp. 293–304. ISBN: 3-540-55707-5.

[7] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[8] Natacha Portier. "Computability - Lecture notes". 2009. URL: `http://perso.ens-lyon.fr/jeanmarie.madiot/polyfdi.pdf`.