A sequent-calculus presentation of type-theory

Gabriel Radanne

Under the supervision of Jean-Philippe Bernardy

Abstract. Dependent types are an active area of research as foundations of mathematics but also as a programming language where many invariants can be internalized as types. Both these objectives argue for a minimal and efficient dependently typed core language. Most implementations so far use a core language based on "natural deduction", which we argue is ill-suited for a type-checking backend. We think that "natural deduction" style is the source of efficiency issues in the presence of inference, it makes terms grows so large that it causes efficiency issues and make the output of the typechecker to large to be verified. Following some ideas from PiSigma, we propose to use a core language in sequent calculus style to solve those issues. We believe that this alternative solves theses efficiency issues.

Keywords: Dependent types, Sequent calculus, Type theory

Forewords

This report aims to present the work accomplished during my 6 month internship in Chalmers, under the supervision of Jean-Philippe Bernardy and with the collaboration of Guilhem Moulin and Andreas Abel.

This report presupposes some familiarity with a statically typed functional language like OCaml or Haskell, but no intimate knowledge of type theory or dependent types is required.

1 A crash-course in dependent types

In most programming languages, terms and types live in two different worlds: one cannot refer to terms in types and types can not be manipulated like terms. On the other hand, in a dependently typed programming language, types can depend on terms. This addition may sound modest at first, but it makes the language more powerful... and harder to typecheck. Dependent types were previously mostly used for theorem proving (in Coq, for example). However they have since gained some popularity as a base for programming languages with Agda [Norell, 2007], Idris [Brady, 2013] or ATS [Chen and Xi, 2005]. It is also related to the recent addition of some features in more mainstream programming languages, such as Generalized Algebraic Datatypes (GADTs) in OCaml or Haskell.

GADTs [Xi et al., 2003] allows to encode some properties in a non dependent type systems that would otherwise need dependent types. For example it is possible to use GADTs to encode vectors shown in Sec. 1.1. However in the presence of GADTs, terms and types still live in two different worlds: GADTs are not as powerful as dependent types, as we show Sec. 1.1.

1.1 An example in Agda

Numerous examples have been presented to motivate the use of dependent types in mainstream programming [Oury and Swierstra, 2008, Brady et al., 2008]. We give here a short and simple example to outline the specificity of dependent type languages from the user point of view but also from the typechecking point of view.

For this example, we use the Agda programming language. The syntax should be familiar enough given some knowledge of a statically-typed functional language.

Let us first define the `Nat` datatype. A natural number is either zero or a successor of a natural number:

```
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat
```

This definition is similar to a definition using GADTs in OCaml or Haskell. `Set` show that `Nat` is a base type, as `Int` or `Char` would be in OCaml or Haskell.

Let us now move on to a more interesting datatype: vectors with fixed length.

```
data Vec (A : Set) : Nat → Set where
  Nil : Vec A Zero
  Cons : {n : Nat} → A → Vec A n → Vec A (Succ n)
```

One can see in the signature of the above datatype that `Vec` is parametrized by type `A`, the type of the elements, and indexed by a natural number which is the length of the vector. The declaration of `Cons` exhibits a useful feature of Agda: the argument `{n : Nat}` is implicit. The compiler infers this argument whenever possible and the length of the vector we are consing to is not needed. Providing the length on every call of `Cons` would be cumbersome.

We can use this type information to implement a type-safe `head` function:

```
head : {A : Set} {n : Nat} → Vec A (Succ n) → A
head (Cons x xs) = x
```

Again, arguments `A` and `n` are declared implicit and it is left to the typechecker to infer them. The compiler knows that the `Nil` case cannot happen because the length of the provided vector is at least one. Any call to `head` with a empty vector argument does not typecheck.

Assuming an infix addition function `(+): Nat → Nat → Nat`, we can also implement the append function, which requires us to manipulate the natural numbers embedded in the type:

```
append : forall { n m A } → Vec A n → Vec A m → Vec A (n + m)
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

In the type, we assert that the length of the results is the sum of the lengths of the operands. We use the `forall` quantifier to declare the implicit arguments without specifying their types, as Agda is able to infer them.

For now, we have seen that dependent types can be useful to assert properties on some datatype. Those simple examples could be encoded with GADTs although it would need additional burden and be far less easy to manipulate. We could go on and declare some other functions on vectors, however to further motivate the use of dependent types, we rather present something difficult or impossible to do using GADTs.

We present an embedding of relational algebra that was first discussed by Oury and Swierstra [2008]. A typed embedded domain specific language for relational databases present interesting difficulties: relational algebra operators are hard to type, especially the join and cartesian product, and type safety usually relies on the static declaration of a schema. We use dependent types to overcome these two issues.

Let us first considerate the definition of a table schema:

```
data U : Set where
  BOOL : U
  CHAR : U
  NAT : U
  VEC : U → Nat → U

Schema : Set
Schema = List (String × U)
```

Here, $\times$ is simply the type of pairs. The `U` type is the universe type for the values inside our database. Databases are restricted to what type of value they can handle so this restriction is perfectly valid. A schema here is simply a list of columns with a name and a type.

We need to decode the constructors of `U` to their representation as Agda types:

```
el : U → Set
el BOOL = Bool
el CHAR = Char
el NAT = Nat
el (VEC x n) = Vec (el x) n
```

A table is composed of rows which follow some schema.

```
data Row : Schema → Set where
  EmptyRow : Row []
  ConsRow : forall {name u s} → el u → Row s → Row ((name , u) :: s)

Table : Schema → Set
Table s = List (Row s)
```

A Row is a list with added type information about the schema. Notice how the table is parametrized by the schema it instantiates.

We can now define a datatype for relational algebra operators:

```
data RA : Schema → Set where
  Read : forall { s } → Table s → RA s
  Union : forall { s } → RA s → RA s → RA s
  Product : forall { s s' } → {So (disjoint s s')} → RA s → RA s' → RA (s ++ s')
  Select : forall { s } → Expr s BOOL → RA s → RA s
  ...
```

The first two constructors are straightforward, Read read a given table and Union merge two tables following the same schema. The Product constructor, however, is much more interesting. To be able to compute the cartesian product of two tables, their columns must be disjoint. We can easily provide a function checking that two schema are disjoint, of type:

```
disjoint : Schema → Schema → Bool
```

We now want to ensure that this property is checked at the type level. In order to do so, we define a bridge from Bool to types:

```
So : Bool → Set
So false = Empty
So true = Unit
```

So takes a Bool and returns a type. The Empty type is, as its names indicates, a type with no elements. Unit being the type with only one element. The type So x has an element if and only if it is Unit, in other word if x is true. If the type has no element, it is impossible to find an expression that have this type and hence the program cannot typecheck.

The Product constructor takes So (disjoint s s') as argument: this is a proof that s and s' are indeed disjoint.

Finally we define expressions used by Select:

```
data Expr : Schema → U → Set where
  _!_ : (s : Schema) → (column : String) → {p : So (occurs column s)} → Expr s (
      lookup column s p)
  equal : forall { u s } → Expr s u → Expr s u → Expr s BOOL
  literal : forall { u s } → el u → Expr s u
  ...
```

Constructors equal and literal are simple and could be encoded easily with GADTs. However, the _!_ constructor, which allows to get the value of a column, takes as argument So (occurs column s) for some schema s. This is a proof that the column appears in the schema. The occurs function would have the type:

```
occurs : String → Schema → Bool
```

We want the `_!_` constructor to return an expression of the type of the return column, hence we could be tempted to use a mere lookup operator:

```
lookup : (col : String) → (s : Schema) → U
```

However, Agda only accepts terminating functions. The `lookup` function, as defined here, is not guaranteed to terminate. Fortunately, we know that, in the context of selects, this lookup always terminate thanks to the proof object `{p : So (occurs column s)}`. Hence we define the lookup function with this type instead:

```
lookup : (col : String) → (s : Schema) → So (occurs col s) → U
```

We can see in this example multiple characteristics of dependently typed programming languages. First, types and terms evolve in the same word and there is little to no distinction between them. Second, terms inhabiting a type are proofs of the proposition expressed by this type. This is a literal translation of the Curry-Howard isomorphism. In other theorem provers, like Coq, users can make use of a separate language to construct proofs, which are λ-terms.

Finally, the typechecker must evaluate terms in order to typecheck. This make the typechecking more complicated and is the source of some limitation in current typecheckers. It is also part of the focus of this work.

2 Limitations of current implementations

The Agda typechecker contains some well known issues that the dependent type theory community has been trying to solve:

- The "case decomposition" issue, which is presented later on, Fig. 10. This issue comes from the fact that natural deduction style makes propagating typing constraints to subterms difficult.
- Agda's type checker is using a natural deduction style and we believe this is why it suffers efficiency issues. Inference duplicates parts of terms which are not shared in the Agda core representation of terms. Therefore typechecking must be done multiple time, causing performance penalties.
- Agda currently does not have a core language that can be reasoned about and formally verified.

Various new languages have been proposed to solve these issues, including Mini-TT [Coquand et al., 2009] and PiSigma [Altenkirch et al., 2010].

PiSigma is especially interesting because it tackles those problems by putting some constructions of the language in sequent calculus style, as explained in the next section. Unfortunately, although this solve some issues, others are introduced. In particular, the language lacks subject reduction (that is, evaluation does not preserve typing). We believe that these issues are present mostly because part of the language is still in natural deduction style.

2.1 Sequent calculus presentation

There are various definitions of sequent calculus. In this report, we mean that every intermediate results or sub-terms are bound to a variable. Sequent calculus is a well known presentation for classical logic but so far has not been evaluated as a presentation of a type theory. The translation from natural deduction to sequent calculus can be mechanized for non-dependent type systems [Puech, 2013]. It is interesting to investigate a sequent calculus presentation of dependent types, because it presents interesting properties:

- It is low-level, which makes it suitable as a back-end for dependently-typed languages.
- Sharing is expressible [Launchbury, 1993]. In natural deduction style, we cannot express the fact that two subterms are identical. This is however desired as it would help solving some efficiency issues encountered in Agda, for example.

2.2 Goals

We aim to construct a type-theory which can be used as a back-end for dependently-typed languages such as Agda or Coq. We call this language nano-Agda. Concretely, our goals are to have a language that is:

- A type-theory: correctness should be expressible via types.
- Low-level: one should be able to translate high-level languages into this language while retaining properties such as run-time behaviour, complexity, etc.
- Minimal: It should well defined and be possible to formally verify the type-checking algorithm.

3 Description of the language

Before describing the language itself, we define some common notions in type theory that we manipulate in the rest of this report.

3.1 Preliminary vocabulary

Constructor and destructors
A language is often separated into destructors (also called eliminations) and constructors (also called introductions). For example in the lambda calculus, lambda abstractions are constructors and applications are destructors. A destruction of construction (called redex), can be reduced through β-reduction. In a more complicated language (including nano-Agda) there are also pairs (constructors) and projections (destructors). The projection of a pair can be similarly reduced. In sequent calculus, redexes are implement as cuts.

Universes
In regular programming languages, one has types in the one's hand and the set of types in the other hand. The latter cannot be manipulated directly but because terms cannot be used as types (or vice-versa), this is not an issue. However, in a dependently typed programming language, terms and types live together, therefore one can theoretically manipulate the set of types. One may wonder: is this set of types a type itself? For technical reasons [Benke et al., 2003] and in order to preserve the consistency of the type system, the answer has to be negative. The problem is essentially the same as Russel's paradox in mathematics: the set of sets cannot be set.

Therefore, types are stratified in universes (also called "sorts" or "kinds") indexed by natural numbers. We note these universes $\star_i$ with $i \in \mathbb{N}$. Base types, like `Int`, are in $\star_0$ while $\star_i$ is in $\star_{i+1}$. Types composed of other types live in the highest universe of their components[1]. For example `(Char, Int)` live in $\star_0$ while (`Int`, $\star_0$) is in $\star_1$. Finally, for ease of manipulation, any element in $\star_i$ is in $\star_j$ whenever $i \leq j$. In the case of nano-Agda, typing rules for universes are given Fig. 7.

3.2 nano-Agda

As explained in Sec. 2.1, every variable is bound. We can separate elements of the languages, presented Fig. 1, into the following categories:

Variables are separated in two categories: conclusions and hypotheses.
Hypotheses are available in the beginning of the program or are the result of an abstraction. It is not possible to construct an hypothesis.
Conclusions are either an hypothesis or the result of a construction of conclusions. Variables with a bar on the top $\overline{x}$ are meta-syntactic variables for Conclusions.
Destructions, ranged over by the letter d in Fig. 1, can be either a `case` (a pattern match) or of the form d as shown in Fig. 1: an application, a projection or a cut. We do not bind the result of a `case`, as opposed to other destructions.

[1] Only true in predicative theories, which are the ones we focus on in this report.

Dependent functions and products follow the same pattern: $(\mathrm{x} : \overline{\mathrm{y}}) \to \mathrm{t}$ and $(\mathrm{x} : \overline{\mathrm{y}}) \times \mathrm{t}$. The type on the left hand side can be a conclusion, because it does not depend on the element x, hence it is possible to bind it before. However, the right hand side must be a term, because it depends on x. x is an hypothesis because it is abstract here.

Enumerations are a set of scopeless and non-unique labels. Labels are plain strings starting with an apostrophe. We use the meta-syntactic variables ‘l and ‘m.

Universes are arranged in a tower, starting at 0, as explained above. We additionally use the shorthand $\star$ for $\star_0$.

Constructions, ranged over by the letter c and detailed Fig. 1, are either a conclusion, a universe, a type or a construction of pair, enumeration or function. The result must be bound to a conclusion.

$$
\begin{aligned}
\mathrm{t} ::=\ & \overline{\mathrm{x}} \\
\mid\ & \texttt{let}\ \mathrm{x} = \mathrm{d}\ \texttt{in}\ \mathrm{t} \\
\mid\ & \texttt{case}\ \mathrm{x}\ \texttt{of}\ \{(\text{‘l} \to \mathrm{t})\ {*}\} \\
\mid\ & \texttt{let}\ \overline{\mathrm{x}} = \mathrm{c}\ \texttt{in}\ \mathrm{t}
\end{aligned}
$$

(a) Terms

$$
\begin{aligned}
\mathrm{d} ::=\ & \mathrm{x}\ \overline{\mathrm{y}} \\
\mid\ & \mathrm{x.1} \mid \mathrm{x.2} \\
\mid\ & \overline{\mathrm{x}} : \overline{\mathrm{y}}
\end{aligned}
$$

(b) Destructions

$$
\begin{aligned}
\mathrm{c} ::=\ & \mathrm{x} \\
\mid\ & \lambda \mathrm{x.t} \mid (\mathrm{x} : \overline{\mathrm{y}}) \to \mathrm{t} \\
\mid\ & (\overline{\mathrm{x}}, \overline{\mathrm{y}}) \mid (\mathrm{x} : \overline{\mathrm{y}}) \times \mathrm{t} \\
\mid\ & \text{‘l} \mid \{\text{‘l}\} \\
\mid\ & \star_{\mathrm{i}}
\end{aligned}
$$

(c) Constructions

Fig. 1: Abstract syntax for nano-Agda

Conclusions are the result of constructions of conclusions and hypothesis is the base case of constructions. An hypothesis is the result of destructions of hypotheses. This means that we can only produce constructions of destructions, hence there is no reduction possible and the program is in normal form.

Obviously we do not only want to write programs already in normal form, so we need a way to construct hypotheses from conclusions. That is the purpose of the cut construction, shown in red in Fig. 1. It allows to declare a new hypothesis, given a conclusion and its type. The type is needed for type checking purposes.

3.3 A bit of sugar

Of course, it is tedious to write reasonable programs with this syntax as it is far too verbose for humans. We introduce another simpler syntax that can be seen below. This new syntax can be translated to the one defined in the previous section. The translation can be done even on ill-typed terms and hence do not need preliminary typechecking. It is similar to the transformation in continuation passing style defined by Plotkin [1975]. The translation binds every intermediate terms to a fresh variable and replaces the subterm by this variable.

Every program is composed of two parts: a term and its type. The typechecker checks that the type lives in a universe and then checks the term against its type. Fig. 2 is an example of a program in high-level syntax and its translation to the low-level syntax. The low-level version is verbose, which argues for the need of a high-level one.

```
TERM
  λa → λx → x

TYPE
  (a : ⋆₁) → (x : a) → a
```

```
TERM
  f = λa → ( f‘ = λx → (r=x; r);
             f‘ ) ;
  f
TYPE
  s1 = ⋆₁ ;
  f_ty = (a : s1) → ( a‘ = a ;
                      a2a = (x : a‘) → a‘;
                      a2a ) ;
  f_ty
```

Fig. 2: The polymorphic identity, in both high-level and low-level syntax.

3.4 An encoding for sum types

Before giving the details of our type system and evaluation strategy, let us consider a small example: we want to create a non-dependent sum type, as used in Agda, Haskell or OCaml, in nano-Agda. We only have enumerations, dependent products and dependent functions but this is enough to encode sum types. Fig. 3 shows a simple Agda sum type and the equivalent code in nano-Agda.

The trick in this encoding is to separate the tag part (`Foo` and `Bar`) from the type part. The tag part can be easily encoded as an enumeration. As for the type part, we take advantage of the dependent product to pattern match the tag and return the appropriate type. In this case, we have a sum type with a parameter, which is translated into a function.

```
data MySumtype (s : Set) : Set
   where
  Foo : s → MySumtype s
  Bar : MySumtype s
```

```
TERM
  Unit = { ‘unit } : ⋆0 ;
  λs → ( c : { ‘Foo , ‘Bar } ) ×
       ( case c of {
           ‘Foo → s.
           ‘Bar → Unit.
       } )
TYPE
  ⋆0 → ⋆0
```

Fig. 3: A sum type in Agda and nano-Agda.

This example shows the fact that, in a dependently typed programming language, enumerations are enough to simulate sum types, which is not possible in a non dependently typed programming language. Here, a more powerful type system allows us to use an arguably simpler core language.

4 Evaluation and type system

The typing rules for nano-Agda are usual, most of the cleverness is contained in the way the environment is updated. Hence we start by presenting environment and environment extensions.

We use the same notation as in Sec. 4.1: x for hypotheses, $\overline{\mathrm{x}}$ for conclusions, c for constructions and d for destructions. For the sake of clarity, elements used as types are capitalized.

4.1 The heap

Because the language is based on variables and bindings, we need a notion of environment. This notion is captured in a heap composed of four mappings:

$\Gamma : \mathrm{x} \mapsto \overline{\mathrm{y}}$	The context heap, containing the type of hypotheses.
$\gamma_c : \overline{\mathrm{x}} \mapsto \mathrm{c}$	The heap from conclusion to constructions.
$\gamma_a : \mathrm{x} \mapsto \mathrm{y}$	The heap for aliases on hypotheses.
$\gamma_d : \mathrm{x} \mapsto \mathrm{d}$	The heap from hypotheses to cuts and destructions.

We also note $\mathrm{h} = (\gamma_c, \gamma_a, \gamma_d)$ for the heap alone and $\gamma = (\Gamma, \gamma_c, \gamma_a, \gamma_d)$ for the heap with type information in the context.

4.2 Environment extensions

Here are details of how to update the heap when registering new information. We use the $\leftarrow$ operator to denote an update and the $+$ operator to denote environment extensions.

When typechecking abstractions, like lambda or dependent functions and products, we need to introduce new hypotheses in the context without any value.

$$\gamma + (\mathrm{x} : \overline{\mathrm{Y}}) = \gamma \text{ with } \mathit{\Gamma} \leftarrow (\mathrm{x} : \overline{\mathrm{Y}})$$

When adding a destruction definition, we check if a similar destruction definition exist using γ_d. This allows automatic recovery of sharing for multiple application of a function to the same argument. Searching for a specific destruction can be implemented efficiently by using a reversed map of γ_d, from destructions to hypotheses.

$$\begin{aligned}\gamma + (\mathrm{x} = \mathrm{d}) &= \gamma \text{ with } \gamma_a \leftarrow (\mathrm{x} = \mathrm{y}) && \mathsf{if}\ (\mathrm{y} = \mathrm{d}) \in \gamma_d \\ &= \gamma \text{ with } \gamma_d \leftarrow (\mathrm{x} = \mathrm{d}) && \text{otherwise}\end{aligned}$$

The rule for conclusions is straightforward, because we do not handle automatic sharing for conclusions as we do for destructions. Automatic sharing rediscovering for constructions is more costly than for destructions, because there are at most two components in a destruction whereas there can be far more in constructions. This additional cost should be evaluated but this is left for future work.

$$\gamma + (\overline{\mathrm{x}} = \mathrm{c}) = \gamma \text{ with } \gamma_c \leftarrow (\overline{\mathrm{x}} = \mathrm{c})$$

When checking or evaluating a case, we keep track of constraints on the variable decomposed by the case, allowing us to know inside the body of a case which branch we took. Of course, if two incompatible branches are taken, we abort the typechecking, because that means the context is inconsistent.

$$\begin{aligned}\gamma + (\text{‘l} = \mathrm{x}) &= \gamma && \mathsf{if}\ (\text{‘l} = \mathrm{x}) \in \gamma_c \\ &= \bot && \mathsf{if}\ (\text{‘m} = \mathrm{x}) \in \gamma_c \text{ for ‘l} \neq \text{‘m} \\ &= \gamma \text{ with } \gamma_c \leftarrow (\text{‘l} = \mathrm{x}) && \text{otherwise}\end{aligned}$$

4.3 Evaluation strategy

We use the $\rightsquigarrow$ operator to denote the reduction relation. Reduction rules operate both on a term and a heap. For clarity, we use shortcuts for lookup operations, for example we note $\gamma(\overline{\mathrm{x}}) = \mathrm{z}\,\overline{\mathrm{y}}$ instead of $\gamma_c(\overline{\mathrm{x}}) = \mathrm{x}'$ and $\gamma_d(\overline{\mathrm{x}'}) = \mathrm{z}\,\overline{\mathrm{y}}$.

The evaluation relation is presented as a big step operational semantic from a heap and a term to a multiset of heap and terms. As a short hand, we drop the multiset notation when we return only one value. We use a multiset as return value to handle case decomposition on an abstract variable.

As for every relation involving terms, we traverse the term and add every binding to the heap. When we encounter a case on a tag, we reduce it by taking the matching branch. If the variable is abstract, we return the multiset of the evaluation of each branches.

$$\textsc{EvalCase}\ \frac{\mathrm{h(x)} = (\text{‘l}_\mathrm{i} : \_) \qquad \mathrm{h} + (\text{‘l}_\mathrm{i} = \mathrm{x}) \vdash \mathrm{t_i} \rightsquigarrow \mathrm{h}' \vdash \overline{\mathrm{x}}}{\mathrm{h} \vdash \texttt{case}\ \mathrm{x}\ \texttt{of}\ \{\text{‘l}_\mathrm{i} \mapsto \mathrm{t_i}\} \rightsquigarrow \mathrm{h}' \vdash \overline{\mathrm{x}}} \qquad \textsc{AbstractCase}\ \frac{\mathrm{h(x)} \neq (\text{‘l}_\mathrm{i} : \_) \qquad \forall \mathrm{i} \quad \mathrm{h} + (\text{‘l}_\mathrm{i} = \mathrm{x}) \vdash \mathrm{t_i} \rightsquigarrow \mathrm{h}'_\mathrm{i} \vdash \overline{\mathrm{x}}_\mathrm{i}}{\mathrm{h} \vdash \texttt{case}\ \mathrm{x}\ \texttt{of}\ \{\text{‘l}_\mathrm{i} \mapsto \mathrm{t_i}\} \rightsquigarrow \{\mathrm{h_i} \vdash \overline{\mathrm{x}}_\mathrm{i}\}}$$

$$\textsc{EvalDestr}\ \frac{\mathrm{h} \vdash \mathrm{d} \rightsquigarrow \mathrm{h}' \vdash \mathrm{t}' \qquad \mathrm{h}' + (\mathrm{x} = \mathrm{t}') \vdash \mathrm{t} \rightsquigarrow \mathrm{h}'' \vdash \overline{\mathrm{x}}}{\mathrm{h} \vdash \texttt{let}\ \mathrm{x} = \mathrm{d}\ \texttt{in}\ \mathrm{t} \rightsquigarrow \mathrm{h}'' \vdash \overline{\mathrm{x}}} \qquad \textsc{AddDestr}\ \frac{\mathrm{h} \vdash \mathrm{d} \not\rightsquigarrow \mathrm{h}' \vdash \mathrm{t}' \qquad \mathrm{h} + (\mathrm{x} = \mathrm{d}) \vdash \mathrm{t} \rightsquigarrow \mathrm{h}' \vdash \overline{\mathrm{x}}}{\mathrm{h} \vdash \texttt{let}\ \mathrm{x} = \mathrm{d}\ \texttt{in}\ \mathrm{t} \rightsquigarrow \mathrm{h}' \vdash \overline{\mathrm{x}}}$$

$$\textsc{AddConstr}\ \frac{\mathrm{h} + (\overline{\mathrm{x}} = \mathrm{c}) \vdash \mathrm{t} \rightsquigarrow \mathrm{h}' \vdash \overline{\mathrm{x}}}{\mathrm{h} \vdash \texttt{let}\ \overline{\mathrm{x}} = \mathrm{c}\ \texttt{in}\ \mathrm{t} \rightsquigarrow \mathrm{h}' \vdash \overline{\mathrm{x}}} \qquad \textsc{Concl}\ \frac{}{\mathrm{h} \vdash \overline{\mathrm{x}} \rightsquigarrow \mathrm{h} \vdash \overline{\mathrm{x}}}$$

Destruction of construction are evaluated eagerly, hence we add special rules for each destructions and check if the destructed hypothesis is a cut with the relevant construction. We need to evaluate only one cut for projections, because all previous cuts have already been evaluated. However, a reduction on a lambda can reveal multiple cuts inside the lambda, which are then evaluated.

$$\textsc{EvalProj}_1\ \frac{h(y) = ((\overline{z}, \overline{w}) : \_)}{h \vdash y.1 \rightsquigarrow h \vdash \overline{z}} \qquad \textsc{EvalProj}_2\ \frac{h(y) = ((\overline{z}, \overline{w}) : \_)}{h \vdash y.2 \rightsquigarrow h \vdash \overline{w}} \qquad \textsc{EvalApp}\ \frac{h(y) = (\lambda w.t : \_) \qquad h \vdash t[\overline{z}/w] \rightsquigarrow h' \vdash \overline{x}}{h \vdash (y\ \overline{z}) \rightsquigarrow h' \vdash \overline{x}}$$

When we have only a conclusion left, the evaluation is finished: every redexes has been evaluated while updating the heap.

In the following section, we write $\gamma \vdash t \rightsquigarrow \gamma' \vdash t'$ to mean $\gamma = (\Gamma, h)$, $h \vdash t \rightsquigarrow h' \vdash t'$ and $\gamma' = (\Gamma, h')$.

4.4 Equality rules

Equality rules can only be applied to normalized terms (without cuts). The equality relation, noted $\gamma \vdash t = t'$ is commutative for t and t', hence the rules are given only in one way. Equality rules use the following two operators:

- $x \equiv y$ is the equality between variables. It means x and y are the same variable.
- $x \cong y$ is the variable equality modulo aliases. It is defined as $x \equiv y \vee \gamma_a(x) \cong y \vee x \cong \gamma_a(y)$. In other words, it tests whether two hypotheses are in the same class of aliases. The alias environment is only for hypotheses so this operator is not usable for conclusions.

These two operators are used to test equality between conclusions and hypotheses respectively.

If the context is inconsistent, everything is true. this rule discards non-matching branches of a `case`. It fulfills the same purpose as the rule for environment extensions on labels presented Sec. 4.2.

$$\bot \vdash \_ \longrightarrow \mathit{true}$$

To verify equality on terms, we traverse both terms until we reach the conclusions, then we compare the definition of the conclusions. If the conclusions are equal according to $\equiv$, we return directly.

$$\begin{aligned}
\gamma \vdash \texttt{let}\ x = d\ \texttt{in}\ t = t' &\longrightarrow \gamma' + (x = t'') \vdash t = t' \\
\gamma \vdash \texttt{let}\ \overline{x} = c\ \texttt{in}\ t = t' &\longrightarrow \gamma + (\overline{x} = c) \vdash t = t' \\
\gamma \vdash \texttt{case}\ x\ \texttt{of}\ \{`l_i \mapsto t_i\} = t &\longrightarrow \forall i \quad \gamma + (x = `l_i) \vdash t_i = t \\
\gamma \vdash \overline{x} = \overline{y} &\longrightarrow \overline{x} \equiv \overline{y} \wedge \gamma \vdash \gamma_c(\overline{x}) = \gamma_c(\overline{y})
\end{aligned}$$

To verify that two constructions are equal, we proceed by induction on the structure of constructions.

$$\begin{aligned}
\gamma \vdash `l = `l &\longrightarrow \mathit{true} \\
\gamma \vdash \star_i = \star_j &\longrightarrow i = j \\
\gamma \vdash x = y &\longrightarrow x \cong y \\
\gamma \vdash \lambda x.t = \lambda y.t' &\longrightarrow \gamma + (x = y) \vdash t = t' \\
\gamma \vdash (\overline{x}, \overline{y}) = (\overline{x'}, \overline{y'}) &\longrightarrow \gamma \vdash \overline{x} = \overline{x'} \wedge \gamma \vdash \overline{y} = \overline{y'} \\
\gamma \vdash (x : \overline{y}) \to t = (x' : \overline{y'}) \to t' &\longrightarrow \gamma \vdash \overline{y} = \overline{y'} \wedge \gamma + (x = x') \vdash t = t' \\
\gamma \vdash (x : \overline{y}) \times t = (x' : \overline{y'}) \times t' &\longrightarrow \gamma \vdash \overline{y} = \overline{y'} \wedge \gamma + (x = x') \vdash t = t' \\
\gamma \vdash \{`l_i\} = \{`m_i\} &\longrightarrow \forall i \quad `l_i = `m_i
\end{aligned}$$

The last two rules are interesting in that they are asymmetric: a construction on the left and a variable on the right. To test the equality in this case, we need to introduce new variables and apply destructions on the left-hand side of the equality. This allows to have η-equality in the type theory, therefore we can prove that $\lambda x.(f\,x) = f$, even if f is abstract.

$$\begin{aligned}
\gamma \vdash \lambda x.t = y &\longrightarrow \gamma + (\overline{x} = x) + (z = y\ \overline{x}) \vdash t = z \\
\gamma \vdash (\overline{x}, \overline{x'}) = y &\longrightarrow \gamma + (z = y.1) \vdash \overline{x} = z \wedge \gamma + (z = y.2) \vdash \overline{x'} = z
\end{aligned}$$

4.5 Typing rules

The typing rules can be divided in three mutually defined relations. The two first relations, noted $\leftleftarrows$, are typechecking relations for respectively terms and constructions. The last relation, for destructions, is an inference, noted $\rightrightarrows$.

We note typechecking for terms $\gamma \vdash t \leftleftarrows T$, the rules are presented Fig. 4. The type here is always a complete term and must have been checked beforehand. In the Constr rules, we do not need to typecheck the construction in detail because any construction added this way is typechecked by either the Concl rule or the Cut rule. In the Destr rule, we use the inference relation on destructions to ensure that every hypothesis has a type in the context. We also evaluate the destruction eagerly.

$$\textsc{Case}\ \frac{\forall\, i \quad \gamma + (\text{`}l_i = x) \vdash t_i \leftleftarrows T \qquad \Gamma(x) = \{\text{`}l_i\}}{\gamma \vdash \texttt{case}\ x\ \texttt{of}\ \{\text{`}l_i \mapsto t_i\} \leftleftarrows T} \qquad \textsc{Constr}\ \frac{\gamma + (\overline{x} = c) \vdash t \leftleftarrows T}{\gamma \vdash \texttt{let}\ \overline{x} = c\ \texttt{in}\ t \leftleftarrows T} \qquad \textsc{Concl}\ \frac{\gamma_c(\overline{x}) = c \qquad \gamma \vdash c \leftleftarrows T}{\gamma \vdash \overline{x} \leftleftarrows T}$$

$$\textsc{Destr}\ \frac{\gamma \vdash d \rightrightarrows T' \qquad \gamma \vdash d \rightsquigarrow \gamma' \vdash t' \qquad \gamma' + (x = t') + (x : T') \vdash t \leftleftarrows T}{\gamma \vdash \texttt{let}\ x = d\ \texttt{in}\ t \leftleftarrows T} \qquad \textsc{Eval}\ \frac{\gamma \vdash T \rightsquigarrow \{\gamma'_i \vdash \overline{X}_i\} \qquad \forall i \quad \gamma'_i \vdash \overline{X}_i \leftleftarrows \overline{X}}{\gamma \vdash t \leftleftarrows T}$$

Fig. 4: Typechecking a term: $\gamma \vdash t \leftleftarrows T$

The inference relation for destructions, presented Fig. 5, is noted $\gamma \vdash d \rightrightarrows T$. Most rules rely on the fact that every hypothesis has its type in the context. Once we know the type of the hypothesis part of the destruction, we check that the destruction is consistent and reconstruct the complete type. The Cut destructions, on the other hand, are verified by typechecking the conclusion of the cut.

$$\textsc{App}\ \frac{\Gamma(y) = (z : \overline{X}) \to T \qquad \gamma \vdash \overline{z} \leftleftarrows \overline{X}}{\gamma \vdash y\ \overline{z} \rightrightarrows T} \qquad \textsc{Proj}_1\ \frac{\Gamma(y) = (z : \overline{X}) \times T}{\gamma \vdash y.1 \rightrightarrows \overline{X}} \qquad \textsc{Proj}_2\ \frac{\Gamma(y) = (z : \overline{X}) \times T}{\gamma \vdash y.2 \rightrightarrows T} \qquad \textsc{Cut}\ \frac{\gamma \vdash \overline{x} \leftleftarrows \overline{X}}{\gamma \vdash (\overline{x} : \overline{X}) \rightrightarrows \overline{X}}$$

Fig. 5: Inferring the type of a destruction: $\gamma \vdash d \rightrightarrows T$.

A construction is checked against a term or a construction; it is noted respectively $\gamma \vdash c \leftleftarrows T$ and $\gamma \vdash c \leftleftarrows C$. Type checking a construction against a term is merely a matter of traversing the type to access the final conclusion, as shown in rules Fig. 6. When we reach the conclusion of the term, we can look up its definition, which has to be a construction, and continue typechecking. The Infer rule is a bit different in that it uses the context for hypotheses and typechecks by unifying the two types.

$$\textsc{TyDestr}\ \frac{\gamma + (x = d) \vdash c \leftleftarrows T}{\gamma \vdash c \leftleftarrows \texttt{let}\ x = d\ \texttt{in}\ T} \qquad \textsc{TyConstr}\ \frac{\gamma + (\overline{x} = c) \vdash c \leftleftarrows T}{\gamma \vdash c \leftleftarrows \texttt{let}\ \overline{x} = c\ \texttt{in}\ T} \qquad \textsc{TyCase}\ \frac{\forall\, i \quad \gamma + (\text{`}l_i = x) \vdash c \leftleftarrows T_i \qquad \gamma \vdash x \rightrightarrows \{\text{`}l_i\}}{\gamma \vdash c \leftleftarrows \texttt{case}\ x\ \texttt{of}\ \{\text{`}l_i \mapsto T_i\}}$$

$$\textsc{TyConcl}\ \frac{\gamma_c(\overline{x}) = C \qquad \gamma \vdash c \leftleftarrows C}{\gamma \vdash c \leftleftarrows \overline{x}} \qquad \textsc{Infer}\ \frac{\Gamma(x) = \overline{X} \qquad \gamma \vdash \overline{X} = T}{\gamma \vdash x \leftleftarrows T}$$

Fig. 6: Typechecking a construction against a term: $\gamma \vdash c \leftleftarrows T$.

The typechecking rules for constructions, shown in Fig. 7, are similar to the typechecking rules for a language in natural deduction style, except that instead of subterms, we have conclusions. The definition of those conclusions play the role of subterms. Lazys rules can only be applied if the language is lazily evaluated. On the other hand, if the evaluation is strict, the redex has already been reduced to a normal form.

$$\textsc{Pair}\ \frac{\gamma \vdash \overline{\mathrm{y}} \leftleftarrows \overline{\mathrm{X}} \qquad \gamma + (\mathrm{x} = (\overline{\mathrm{y}} : \overline{\mathrm{X}})) \vdash \overline{\mathrm{z}} \leftleftarrows \mathrm{T}}{\gamma \vdash (\overline{\mathrm{y}}, \overline{\mathrm{z}}) \leftleftarrows (\mathrm{x} : \overline{\mathrm{X}}) \times \mathrm{T}} \qquad \textsc{Lambda}\ \frac{\gamma + (\mathrm{y} : \overline{\mathrm{X}}) + (\mathrm{x} = \mathrm{y}) \vdash \mathrm{t} \leftleftarrows \mathrm{T}}{\gamma \vdash \lambda \mathrm{y}.\mathrm{t} \leftleftarrows (\mathrm{x} : \overline{\mathrm{X}}) \to \mathrm{T}} \qquad \textsc{Label}\ \frac{`\mathrm{l} \in \{`\mathrm{l_i}\}}{\gamma \vdash `\mathrm{l} \leftleftarrows \{`\mathrm{l_i}\}}$$

$$\textsc{Sigma}\ \frac{\gamma \vdash \overline{\mathrm{y}} \leftleftarrows \star_{\mathrm{i}} \qquad \gamma + (\mathrm{x} : \overline{\mathrm{y}}) \vdash \mathrm{t} \leftleftarrows \star_{\mathrm{i}}}{\gamma \vdash (\mathrm{x} : \overline{\mathrm{y}}) \times \mathrm{t} \leftleftarrows \star_{\mathrm{i}}} \qquad \textsc{Pi}\ \frac{\gamma \vdash \overline{\mathrm{y}} \leftleftarrows \star_{\mathrm{i}} \qquad \gamma + (\mathrm{x} : \overline{\mathrm{y}}) \vdash \mathrm{t} \leftleftarrows \star_{\mathrm{i}}}{\gamma \vdash (\mathrm{x} : \overline{\mathrm{y}}) \to \mathrm{t} \leftleftarrows \star_{\mathrm{i}}} \qquad \textsc{Fin}\ \frac{}{\gamma \vdash \{`\mathrm{l_i}\} \leftleftarrows \star_{\mathrm{i}}} \qquad \textsc{Universe}\ \frac{\mathrm{i} < \mathrm{j}}{\gamma \vdash \star_{\mathrm{i}} \leftleftarrows \star_{\mathrm{j}}}$$

Fig. 7: Typechecking a construction against a construction: $\gamma \vdash \mathrm{c} \leftleftarrows \mathrm{C}$.

5 Properties on typing and reduction

In order for nano-Agda to be interesting as a core language for a dependently typed framework, we need to provide some guarantee about the behaviour of the execution of well typed terms. The proof for these properties are still being worked on and are left to be published in a future work. Because the language is not in natural deduction style, we present these classic properties in a slightly different way.

A desirable property is that well-typeness is preserved by reduction rules.

Proposition 1. ***Subject reduction*** *Let h be a heap, Γ a context, T and t be two terms. If there exists a heap h' and a term t' such that $h \vdash t \rightsquigarrow h' \vdash t'$. Then we have,*

$$(\Gamma, h) \vdash t \leftleftarrows T \Rightarrow (\Gamma, h') \vdash t' \leftleftarrows T$$

Finally, we want to ensure that any successfully typechecked term evaluate to a normal form. This guarantees that the evaluation of typechecked terms always terminate.

Proposition 2. ***Strong normalization*** *Let h be a heap, Γ a context and t, T terms such that $(\Gamma, h') \vdash t' \leftleftarrows T$. Then there exists a multiset of heaps and conclusions such that cuts are never referenced from destructors in the heaps h'_i and*

$$h \vdash t \rightsquigarrow \{h'_i \vdash \overline{x}_i\}$$

6 Results and Examples

In Sec. 4.2, we argue that sharing can be recovered by checking if a variable is already present in a destruction and recording the alias in this case. We show in Fig. 8 an example where this feature is useful. The function in this example takes as argument a pair `p` and a binary predicate `P`. We then force the typechecker to unify two versions of the same destructions, once at the term level and the other at the type level. To compare them, Agda's typechecker unfolds both terms which can be inefficient if the normal forms are large. In nano-Agda, we rediscover the sharing between the two versions of `u1` and `u2`, hence the structures to compare are

```
sharing :
  (A : Set) → (B : Set) →
  (P : A → B → Set) → (p : A × B) →
  let (u1 , u2) = p
      v = P u1 u2
  in v → v
sharing A B P (u1' , u2') =
  let v' = P u1' u2'
  in \(x : v') → x
```

```
TERM
  A = ⋆₀ : ⋆₁; B = ⋆₀ : ⋆₁;
  Sharing =
    (λP → λp → (u1,u2) = split p;
               v = P u1 u2;
               λx → (y = x : v ; y) )
    : (P : A → B → ⋆₀) →
      (p : (a : A) × B) →
               (u1,u2) = split p;
               v = P u1 u2 ;
               v → v ;

  ⋆₀
TYPE
  ⋆₁
```

Fig. 8: Recovering sharing in Agda and nano-Agda.

smaller. In particular, if `p` in this piece of code was a big term instead of being abstract, the performance penalty for Agda would have been important.

For the next examples, we need a notion of equality that we can use in type signatures. Leibniz' equality is defined by `Eq` and `refl` (for reflexivity) in the example Fig. 9. The idea is to make the unification engine compute the equality. For example, provided `Bool` and `not`, `refl Bool 'true : Eq Bool 'true (not 'false)` make the unification engine compute the fact that `'true` $=$ `not 'false`. The only element of the `Eq` type is a proof by reflexivity. If the two arguments of `Eq` do not unify, the program does not typecheck.

```
data _≡_
    {A:Set} (x:A) : A → Set where
  refl : x ≡ x
```

```
Eq =
  (λA → λx → λy → (P: A → ⋆₀) → P x → P y)
  : (A : ⋆₀) → A → A → ⋆₁;
refl = (λA → λx → λP → λp → p)
     : (A : ⋆₀) → (x:A) → Eq A x x;
```

Fig. 9: Encoding equality at the type level

We assume in the following examples that `Eq` and `refl` are in the scope. We also assume that we have `Bool = { 'true, 'false } :` $\star_0$.

One of the long standing issue in Agda is that the typechecker has no knowledge of which branch was taken while it typechecks the body of a branch. In the example Fig. 10, the typechecker must verify that `h (f x0) x0` $=$ `f x` in the branch were `f x0 = 'true` (the equality is true only in this branch). In nano-Agda, `f x0` is bound to an intermediate variable `y` and the typechecker can express constraints on it (for instance the fact that `'true = y`). On the contrary, the Agda typechecker unfolds each term but does not reconstruct the constraint on `f x`. Fig. 10 does not typecheck in Agda. The fact that this example typecheck in nano-Agda and not in Agda is a direct consequence of the sequent calculus presentation. Indeed the fact that each subterm is bound to a variable allows to express constraints on a much more precise level.

A property of boolean functions is that if `f` is of type `Bool →Bool`, then `f x` $=$ `f (f (f x))`. This was introduced by Altenkirch and Uustalu [2004] in the context of type theory. Fig. 11 encode this property in Agda and nano-Agda using `Eq` and `refl`. Neither Agda nor nano-Agda manage to typecheck this example, however we think that it is possible in nano-Agda with a better handling of nested cases.

```
SmartCase :
  (A : Set) → (A → Bool) →
    (A → Bool) → A → Bool
SmartCase A f g x = h' y
  where h : Bool → A → Bool
        h true = f
        h false = g

        x0 = x
        y = f x0

        h' : Bool → Bool
        h' true =
            let z : (h y x0) ≡ y
                z = refl
            in true
        h' false = false
```

```
Bool = { 'true, 'false } : ⋆0;
A   = ⋆0 : ⋆1;
SmartCase =
  (λf → λg → λx →
    (h = λb → case b of {
                'true  → f.
                'false → g. }
       : (b : Bool) → A → Bool;
    x0 := x;
    y = f x0;
    case y of {
      'true  →
        z = (refl Bool y)
          : Eq Bool (h y x0) y;
        'true.
      'false → 'false.}
  ))
  : (f : A → Bool) → (g : A → Bool) →
     A → Bool
```

Fig. 10: Smart case

```
tripleF :
  (f : Bool → Bool) → (x : Bool) →
     (f x) ≡ (f (f (f x)))
tripleF f x with x | f x
... | true  | true  = refl
... | true  | false = refl
... | false | true  = refl
... | false | false = refl
```

```
tripleF =
  (λf → λx → (
    case x of {
       'true  → case f x of {
            'true  → refl Bool 'true.
            'false → refl Bool 'false.
       }.
       'false → case f x of {
            'true  → refl Bool 'true.
            'false → refl Bool 'false.
       }.
    }))
  : (f: Bool → Bool) → (x : Bool) →
       Eq Bool (f x) (f (f (f x)))
```

Fig. 11: Triple application of a boolean function

7 Conclusion

The language presented in this report aims to solve some issues present in other dependently type languages, including the lack of fine constraints during typechecking and efficiency issues. To tackle this problems, we proposed a language in sequent calculus style. This work is still ongoing, but the current implementation is promising in that it achieve the goals stated in Sec. 2.2. We provided some example demonstrating the ability to encode complicated constructions in the language. Moreover, nano-Agda allows to typecheck some examples that could not be typechecked by previous systems. In the future we would like to use nano-Agda as a platform for experimental features of type theory, such as colors [Bernardy and Moulin, 2013]. However, there is still a lot to do, including:

– Improve the typechecker to typecheck more examples, like the one presented in Fig. 11.
– As it stands, the language does not contain recursion. Of course, it makes the language not yet suitable as a backend. We would like to use size types [Abel, 2006] to implement well-founded recursion.
– Evaluate the performance of the typechecker on some large programs and compare with Agda.
– Prove subject-reduction and normalization.

This internship was also the occasion for me to get a more detailed knowledge on type systems, especially dependently typed ones. On this aspect, I think this internship is successful: I have now a far better understanding of dependent types, both from a usage and a typechecking point of view, than 5 month ago and I think this knowledge will be helpful in the future.

Bibliography

A. Abel. Semi-continuous sized types and termination. In *CSL*, pages 72–88, 2006.

T. Altenkirch and T. Uustalu. Normalization by evaluation for lambda^{-2}. In *FLOPS*, pages 260–275, 2004.

T. Altenkirch, N. A. Danielsson, A. Löh, and N. Oury. Pisigma: Dependent types without the sugar. In M. Blume, N. Kobayashi, and G. Vidal, editors, *FLOPS*, volume 6009 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2010. ISBN 978-3-642-12250-7.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003. URL http://portal.acm.org/citation.cfm?id=985801.

J.-P. Bernardy and G. Moulin. Type-theory in color. In *Proceeding of the 18th ACM SIGPLAN international conference on Functional Programming*, pages 61–72, 2013.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

E. Brady, C. A. Herrmann, and K. Hammond. Lightweight invariants with full dependent types. In *Trends in Functional Programming*, pages 161–177, 2008.

C. Chen and H. Xi. Combining programming with theorem proving. In *ICFP*, pages 66–77, 2005.

T. Coquand, Y. Kinoshita, B. Nordström, and M. Takeyama. A simple type-theoretic language: Mini-tt. 2009.

J. Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers Tekniska Högskola, 2007. URL http://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=46311.

N. Oury and W. Swierstra. The power of Pi. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 39–50, Victoria, BC, Canada, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411213. URL http://portal.acm.org/citation.cfm?id=1411204.1411213.

G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–129, 1975.

M. Puech. Proofs, upside down - a functional correspondence between natural deduction and the sequent calculus. In *APLAS*, pages 365–380, 2013.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235, 2003. doi: 10.1145/640128.604150. URL http://portal.acm.org/citation.cfm?id=604150&dl=GUIDE&coll=GUIDE&CFID=38830212&CFTOKEN=12075942.