Formal Systems and their Applications
Assignment: Implementing a dependently-typed calculus

Dominique Devriese
Frank Piessens

November 16, 2011

1 Introduction

In this assignment, you will develop a type-checker and evaluator for an extension of the lambda calculus with dependent types. Additionally, the assignment involves writing a few small programs in it.

In this text, we provide you with a detailed description of the dependently-typed calculus which we expect you to implement. **Please read it *fully* and *carefully***. Some parts of the assignment are clearly marked as optional. This means that you are not required to execute them to pass this assignment. Implementing the optional assignments correctly will lead to higher grades though.

For this assignment, you are encouraged to work in teams of two students. Teams should spend approximately 30 hours *per team-member* in completing the assignment. For more practical information, see section 13.

2 Before you start

For this assignment, we expect you to have attended and understood the lectures of the course FST, and the corresponding parts of the text book [1]. Additionally, before you start, you should **study section 6 of the textbook**, which has not been covered in the lectures, to understand the nameless representation of variables (using de Bruijn indices) that we use and expect you to use in the assignment.

3 Dependent types

In his well-known textbook on programming languages and types [1], Pierce discusses a variety of lambda calculi and type systems. A common point among all of them is that care is taken to strictly separate the world of terms and types.

This separation is also present in most general-purpose programming languages like Java, C++, Haskell, ML etc.

However, a promising class of programming languages drop the distinction between types and terms, leading to a fairly different kind of programming language. These languages are called *dependently typed.* In fact, this choice makes the type system so powerful that types can reflect arbitrary properties of user programs, which has many applications; dependently typed languages can be used as mathematical proof assistants, automatically checking the correctness of mathematical proofs, but they are also increasingly being used as programming languages.

As an example, probably the most elaborate program developed in a dependently typed language is CompCert[1]: an elaborate, optimizing research C compiler which has been proven *correct*: for a given C program, it produces an assembly program which is proven to behave "the same" as the C program[2]. It is implemented in Coq: a proof assistant based on a dependent type system.

In the course "Formal Systems and Applications", we study the simply typed lambda calculus (STLC) and variations because they are a model for many programming languages. Similarly, we can define a lambda calculus with dependent types, as a model for the essential workings of a dependently typed system. In fact, this calculus can be defined elegantly as a fairly simple lambda calculus with different typing and evaluation rules and it shows some important problems and compromises in implementing a dependently typed programming language. In this project, you will implement such a calculus in Scala.

4 Overview

In section 5, we describe the code package we provide as part of this assignment. It contains examples of lambda calculus implementations as well as a template for what we expect you to hand in. For the calculus which we expect you to implement, sections 6, 7 and 8 describes the syntactic constructs, the evaluation rules and the typing rules respectively. Section 10 describes a mandatory extension with a natural induction primitive and section 11 and section 12 describe optional extensions with singleton types and a form of dependent if primitive.

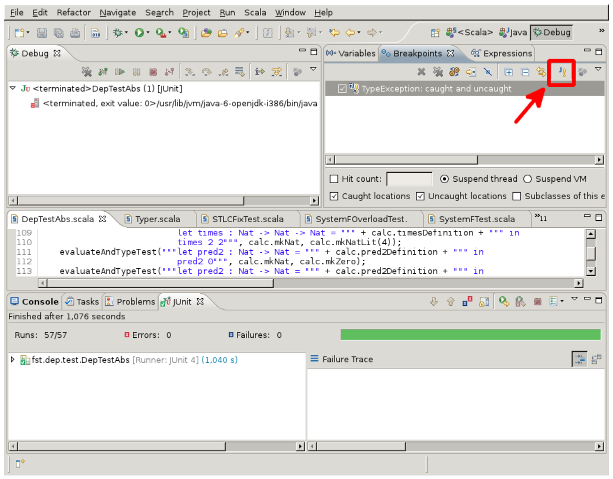
5 The code we provide

5.1 Example Calculi

Together with this assignment, you have received a package containing implementations of two lambda calculi:

[1]`http://compcert.inria.fr`. CompCert is the result of French research led by Xavier Leroy.

[2]for a certain definition of "the same", for example, nothing is specified about what happens in case of null-dereferencing or other undefined behaviour etc.


File Edit Refactor Navigate Search Project Run Scala Window Help
<Scala> Java Debug
Debug
Variables Breakpoints Expressions
DepTestAbs.scala Typer.scala STLCFixTest.scala SystemFOverloadTest. SystemFTest.scala
Console Tasks Problems JUnit
Failure Trace

<terminated>DepTestAbs (1) [JUnit]
<terminated, exit value: 0>/usr/lib/jvm/java-6-openjdk-i386/bin/java

TypeException: caught and uncaught

☐ Hit count: [●] Suspend thread ○ Suspend VM
☑ Caught locations ☑ Uncaught locations ☐ Subclasses of this e

```
                           let times : Nat -> Nat -> Nat = """ + calc.timesDefinition + """ in
                           times 2 2""", calc.mkNat, calc.mkNatLit(4));
    evaluateAndTypeTest("""let pred2 : Nat -> Nat = """ + calc.pred2Definition + """ in
                           pred2 0""", calc.mkNat, calc.mkZero);
    evaluateAndTypeTest("""let pred2 : Nat -> Nat = """ + calc.pred2Definition + """ in
```

Finished after 1,076 seconds

Runs: 57/57 Errors: 0 Failures: 0

fst.dep.test.DepTestAbs [Runner: JUnit 4] (1,040 s)

Figure 1: Exception breakpoints in Eclipse

fst.stlc A Simply Typed Lambda calculus with naturals and booleans.

fst.stlcfix A Simply Typed Lambda calculus with naturals, booleans and general recursion.

The package is intended to be used with Eclipse and the Scala plugin.

Study the implementations of these calculi. We expect you to model your solution after these calculi. Use a similar structure of the code, and deviate from it only when needed.

During debugging, it may be useful to make good use of Eclipse's advanced debugger. One feature we recommend is "exception breakpoints", which you can use to make the debugger break once an exception of a certain type is thrown. See Figure 1 for where to find the button for this.

5.2 Solution Template

The package with the examples also contains a directory source folder "solution" with a couple of mostly empty classes which we expect you to implement. There is also a folder "test-dep" which contains a set of unit tests which we think can help you test your solution to this assignment. They also show at times interesting examples. You may not modify the unit tests, and you have to make sure all of the unit tests succeed, except for the ones which consider parts of this assignment which are optional. These are clearly marked as such in the unit test class.

Pay close attention to the class DepCalculus. Some of the exercises in this assignment require you to implement some empty methods in that class.

Note that we expect you to use the parser which we provide in the class fst.common.DepParser. You may not modify it. It will call the methods in your class fst.dep.DepCalculus to build terms.

6 Syntax

6.1 Basics

Figure 2 shows the basic syntax of our dependently typed language. Note first that there is only one syntactic category: *terms*, since we do not distinguish between types and terms. The definition starts with $s, t ::=$ to indicate that we will often use the letters s and t when we implicitly mean a term. The types in our calculus will be a subset of the terms, but more on that later. Apart from the removal of the distinction between types and terms, there are only two novelties here with respect to the standard lambda calculus:

- Set: This is the *type of types*. In our calculus, we no longer maintain a strict separation between types and terms, but that does not mean all terms t can be used as the type of another term s. For that, t has to be of type Set:

  $$t : Set$$

  For example, further in this text, we will see that the typing rule for a lambda abstraction $(\lambda x : t_1.t_2)$, will require that $t_1 : \text{Set}$. We will also see that for example the type of naturals is of type Set:

  $$\text{Nat} : \text{Set}$$

- *Dependent function types ($(x : s) \to t$)*: This is a generalization of the standard lambda calculus type of functions $T \to T$. The difference here is that we give a name (x) to the value that the function can be applied to, and the result type of the function is allowed to depend on this value. For example, we could define a function which takes a boolean and returns either a natural number or another boolean $\lambda b : \text{Bool} \,.\, \text{if}\, b \,\text{then}\, 3 \,\text{else}\, \text{true}$. Its type could be $(b : \text{Bool}) \to \text{if}\, b \,\text{then}\, \text{Nat} \,\text{else}\, \text{Bool}$. More on this in section 12[3]

  Note that in the dependent function type $(x : s) \to t$, the type t can refer to the variable x. This means that during the substitution and shifting of de Bruijn variable indices, you need to properly take this extra bound variable into account. To do this, you can take inspiration from how substitution and shifting is done for lambda terms $\lambda x : t.b$, where b can refer to variable x. But just so we are completely clear: these two concepts (lambda terms and dependent function types) are *completely different* with respect to type checking or evaluation (see the respective rules in this assignment).

[3]Note that this example $\lambda b : \text{Bool} \,.\, \text{if}\, b \,\text{then}\, 3 \,\text{else}\, \text{true}$ does not mean that our calculus is "dynamically typed", where types are only checked at run-time. All types in our calculus will be checked at compile-time. For this example, whenever the function is applied to a value, it will be checked that for that value of b, the result of the function is used as the correct type.

Basic Syntax

$s, t ::=$	**terms**
x	variables
$\lambda x : t.t$	abstraction
$t\ t$	application
$(x : t) \to t$	dependent function type
$\text{let}\, x : t = t \,\text{in}\, t$	let-binding
Set	type of types

Figure 2: Dependently-typed calculus: basic syntax

> Only the substitution and shifting of de Bruijn indices in respectively the result type and the body are similar.

A function type of the form $(x : s) \to t$ is called a *dependent function type* or Π-*type* (Π is the greek capital letter "pi"). In fact, the notation of $((x : s) \to t)$ is not standard, but we borrow it from the dependently typed programming language Agda[4].

Note that we do not define ordinary function types $t_1 \to t_2$. This is because these can be seen as a restricted form of dependent function types $(x : s) \to t$, where t does not depend on x. We expect your solution to support them using the following translation:

$t_1 \to t_2$ is translated to $(x : t_1) \to t_2$ where t_2 does not refer to x

Watch out: treat the de Bruijn indices of variables properly during this translation!

6.2 Syntactic equality

In your implementation of the type-checking rules, you will occasionally need to check the equality of two terms. You have to make sure that you define this in an α-equivalent way. This means that you treat two lambda abstractions $\lambda x : t_1.b_1$ and $\lambda y : t_2.b_2$ as equal if and only if t_1 is syntactically equal to t_2 and b_1 is syntactically equal to b_2, but without requiring that the variable names x and y are the same, and similarly for dependent function types $(x : s) \to t$.

6.3 Extensions

On top of these basic syntactic constructs, we define some extensions for working with booleans and natural numbers. Figures 3 and 4 show the additional syntactic constructs. We have the types Bool and Nat, as well as standard primitive constructs as defined by Pierce [1, §3].

[4]http://wiki.portal.chalmers.se/agda/pmwiki.php

booleans

$s, t ::=$	**terms**
...	
Bool	type of booleans
true	true value
false	false value
if t then t else t	if-then-else

Figure 3: Dependently-typed calculus: booleans

natural numbers

$s, t ::=$	**terms**
...	
Nat	type of natural numbers
zero	zero
succ t	successor
iszero t	test if number is zero
pred t	predecessor

Figure 4: Dependently-typed calculus: natural numbers

7 Evaluating our calculus

One of the most important issues in a dependently-typed calculus is that type-checking now requires evaluating terms. The example $\lambda x : \text{Bool} . \text{if } x \text{ then } 3 \text{ else true}$ we saw before could be typed $(x : \text{Bool}) \to \text{if } x \text{ then Nat else Bool}$. Now, when we apply this lambda to the value true, the type-checker should see that its type $\text{if true then Nat else Bool}$ (with $x = \text{true}$ filled in) is the same as Nat. Therefore, type-checking terms with dependent types requires evaluating arbitrary terms during type-checking.

Now, this fact has an important consequence. If we evaluate terms during type-checking, we have to be sure that the type-checker will not end up in an infinite loop. Therefore, all terms in a dependently typed lambda calculus are required to normalize. Remember how all terms in the simply typed lambda calculus [1, §12] normalise, a dependently-typed calculus has the same property, and it is even more important there in order to not make the type-checker go into an infinite loop. The fact that all terms in our calculus normalized is often stated by saying that the calculus is *total* (as opposed to *partial*). Therefore, a dependently typed language cannot provide a general recursion construct like fix. This does not mean however that dependently typed languages are not Turing-complete[5].

The fact that type-checking our calculus requires evaluating its terms has another consequence for our evaluation strategy. In fact, the type-checker requires support for "evaluation under assumptions". What this means is that the type-checker needs to be able to see for example that $(x : \text{Bool}) \to \text{if true then Nat else Bool}$ is the same type as $(x : \text{Bool}) \to \text{Nat}$. This means that it must evaluate "inside" or "under" lambda's, something which a typical lambda calculus doesn't do (see e.g. [1, Figure 5-3]).

Therefore, the evaluation strategy we use is a form of *(non-deterministic) full beta-reduction* as defined by Pierce [1, §5.1 (Operational Semantics)]. The evaluation rules are shown in Figure 5 and 6. Note that this reduction relation is non-deterministic: it is not defined in which order the rules must be applied. However, it turns out that because our calculus is total, this order is less important. Because all our terms normalise anyway, the evaluation order no longer has an effect on termination of the terms, only on the efficiency and memory use of the evaluation. Therefore, in this assignment, we expect you to ignore these concerns, and in your implementation, you are free to try the different rules in any order you like. However, you must make sure that you try all the rules; the result of evaluating a term must always be in normal form, i.e. none of the rules in Figure 5 apply.

In what follows, we will write $s \longrightarrow^* t$ to mean that s normalises to t, i.e. if we apply the evaluation rules (as discussed, order does not matter) until no rule applies anymore, we get t. Clearly, this means that t must be normal: no evaluation rule applies to it anymore.

[5]The reason for this is that potentially non-terminating computations can be encoded in a dependently-typed calculus in a similar way that calculations performing I/O can be encoded in a monad in the pure programming language Haskell.

$$\frac{t_2 \longrightarrow t_2'}{\lambda x : t_1.t_2 \longrightarrow \lambda x : t_1.t_2'} \quad \text{(E-ABS1)}$$

$$\frac{t_1 \longrightarrow t_1'}{\lambda x : t_1.t_2 \longrightarrow \lambda x : t_1'.t_2} \quad \text{(E-ABS2)}$$

$$\frac{t_2 \longrightarrow t_2'}{(x : t_1) \to t_2 \longrightarrow (x : t_1) \to t_2'} \quad \text{(E-PI1)}$$

$$\frac{t_1 \longrightarrow t_1'}{(x : t_1) \to t_2 \longrightarrow (x : t_1') \to t_2} \quad \text{(E-PI2)}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1 t_2 \longrightarrow t_1 t_2'} \quad \text{(E-APP1)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2} \quad \text{(E-APP2)}$$

$$\frac{}{\text{let}\, x : t_1 = t_2 \,\text{in}\, t_3 \longrightarrow [x \mapsto t_2]t_3} \quad \text{(E-LET)}$$

$$\frac{}{(\lambda x : t_1.t_2)t_3 \longrightarrow [x \mapsto t_3]t_2} \quad \text{(E-APPABS)}$$

Figure 5: Evaluation rules for the basic parts of our calculus

8 Type checking

Figure 7 shows the typing rules for the basic parts of our calculus. There are a few important things to note in the rules. Like in other systems, we use a context containing a list of the types of the variables. Rule (T-VAR) is standard. Rule (T-ABS) checks that the argument type of a lambda abstraction is a valid type by checking it is of type Set. It then normalises this type (this is an instance of the use of evaluation inside type-checking we have discussed before). Then, the normalised type is added to the context to check the type of the body t_2. Typing this body may produce a type which depends on the value x of the variable we've added to the context (remember: dependent types), so the type of our lambda becomes a Π-type. Rule (T-PI) type-checks Π-types. The rule requires checks that type t_1 is a type, normalises it, checks that t_2 is a type with $x : t_1$ added to the context and if so, the Π-type is a type. Note the difference between $\lambda x : t_1.t_2$ and $(x : t_1) \to t_2$.

The type checking rule (T-APP) for applications $t_1 t_2$ is different from the one in a standard lambda calculus, because the function t_1's type may be a Π-type $(x : t_3) \to t_4$. In that case, t_2 must be of type t_3 as usual, but the application $t_1 t_2$ will have as type the normalisation of $[x \mapsto t_2]t_4$: the value of the Π-type for the argument that the function was applied to. Finally, instead of translating a let-expression $\text{let}\, x : t_1 = t_2 \,\text{in}\, t_3$ to an application $(\lambda x : t_1.t_3)t_2$, we treat it specially according to rule (T-LET). We check that t_1 is a set, normalise it to t_1', check that t_2 is of type t_1', but then we do something special. Instead of

$$\frac{}{\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2} \quad \text{(E-IfTrue)}$$

$$\frac{}{\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3} \quad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad \text{(E-If1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1 \text{ then } t_2' \text{ else } t_3} \quad \text{(E-If2)}$$

$$\frac{t_3 \longrightarrow t_3'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1 \text{ then } t_2 \text{ else } t_3'} \quad \text{(E-If3)}$$

$$\frac{t \longrightarrow t'}{\text{succ } t \longrightarrow \text{succ } t'} \quad \text{(E-Succ)}$$

$$\frac{}{\text{pred } 0 \longrightarrow 0} \quad \text{(E-PredZero)}$$

$$\frac{}{\text{pred}(\text{succ } t) \longrightarrow t} \quad \text{(E-PredSucc)}$$

$$\frac{t \longrightarrow t'}{\text{pred } t \longrightarrow \text{pred } t'} \quad \text{(E-Pred)}$$

$$\frac{}{\text{iszero } 0 \longrightarrow \text{true}} \quad \text{(E-IszeroZero)}$$

$$\frac{}{\text{iszero}(\text{succ } t) \longrightarrow \text{false}} \quad \text{(E-IszeroSucc)}$$

$$\frac{t \longrightarrow t'}{\text{iszero } t \longrightarrow \text{iszero } t'} \quad \text{(E-Iszero)}$$

Figure 6: Evaluation rules related to booleans and natural numbers

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \quad \text{(T-VAR)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathrm{Set} \qquad t_1 \longrightarrow^* t_1' \\ x : t_1', \Gamma \vdash t_2 : t_3\end{array}}{\Gamma \vdash \lambda x : t_1 . t_2 : (x : t_1') \to t_3} \quad \text{(T-ABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathrm{Set} \qquad t_1 \longrightarrow^* t_1' \\ x : t_1', \Gamma \vdash t_2 : \mathrm{Set}\end{array}}{\Gamma \vdash (x : t_1) \to t_2 : \mathrm{Set}} \quad \text{(T-PI)}$$

$$\frac{\Gamma \vdash t_1 : (x : t_3) \to t_4 \qquad \Gamma \vdash t_2 : t_3 \qquad [x \mapsto t_2]\ t_4 \longrightarrow^* t_4'}{\Gamma \vdash t_1\ t_2 : t_4'} \quad \text{(T-APP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathrm{Set} \qquad t_1 \longrightarrow^* t_1' \qquad \Gamma \vdash t_2 : t_1' \\ \Gamma \vdash [x \mapsto t_2] t_3 : t_4\end{array}}{\Gamma \vdash \mathrm{let}\, x : t_1 = t_2 \,\mathrm{in}\, t_3 : t_4} \quad \text{(T-LET)}$$

Figure 7: Typing rules for the basic parts of our calculus

adding $x : t_1'$ to the context to type-check t_3, we substitute t_2 for x in the body of t_3. The reason for this is that for type-checking t_3, in a dependently-typed calculus, it is not always sufficient to only know the type of x. Sometimes, we also need to know its value (the fact that $x = t_2$). We will see an example of such a case in one of the final (optional) parts of the assignment.

Figure 8 shows the typing rules for the rest of our calculus. Most rules are straightforward, except for (T-SETINSET). What the rule says is that the type of all sets is a set itself. In fact, the question of what type the set of all sets is is a difficult one. What we show here is the simplest solution, which works for most cases, but is actually unsound: in more complex systems, it is possible to construct examples using this rule which break most of the guarantees we try to offer (like the fact that all terms terminate and the validity of the proofs written in the calculus). However, these examples are not obvious, and like some real dependently-typed programming languages (e.g. Idris[6]), we choose to ignore the issue here, instead of implementing the (relatively complex) solution for this[7].

Something to note in Figure 7 is the fact that all types t_2 which appear to the right of the : sign in a typing judgement $\Gamma \vdash t_1 : t_2$ are normalised. You can

[6] `http://www.cs.st-andrews.ac.uk/~eb/Idris/`

[7] A more correct solution is called "stratification" or a "stratified hierarchy of universes". The idea is to not work with a single Set, but instead work with an infinite number of Sets, such that Nat : Set_0, Set_0 : Set_1, Set_1 : Set_2 etc.

$$\frac{}{\Gamma \vdash \mathrm{Set} : \mathrm{Set}} \quad (\text{T-SetInSet})$$

$$\frac{}{\Gamma \vdash \mathrm{Nat} : \mathrm{Set}} \quad (\text{T-Nat})$$

$$\frac{}{\Gamma \vdash \mathrm{zero} : \mathrm{Nat}} \quad (\text{T-Zero})$$

$$\frac{\Gamma \vdash t : \mathrm{Nat}}{\Gamma \vdash \mathrm{succ}\, t : \mathrm{Nat}} \quad (\text{T-Succ})$$

$$\frac{\Gamma \vdash t : \mathrm{Nat}}{\Gamma \vdash \mathrm{pred}\, t : \mathrm{Nat}} \quad (\text{T-Pred})$$

$$\frac{\Gamma \vdash t : \mathrm{Nat}}{\Gamma \vdash \mathrm{iszero}\, t : \mathrm{Bool}} \quad (\text{T-Iszero})$$

$$\frac{}{\Gamma \vdash \mathrm{Bool} : \mathrm{Set}} \quad (\text{T-Bool})$$

$$\frac{}{\Gamma \vdash \mathrm{true} : \mathrm{Bool}} \quad (\text{T-True})$$

$$\frac{}{\Gamma \vdash \mathrm{false} : \mathrm{Bool}} \quad (\text{T-False})$$

$$\frac{\Gamma \vdash t_1 : \mathrm{Bool} \qquad \Gamma \vdash t_2 : t \qquad \Gamma \vdash t_3 : t}{\Gamma \vdash \mathrm{if}\, c\, \mathrm{then}\, t_2\, \mathrm{else}\, t_3 : t} \quad (\text{T-If})$$

Figure 8: Typing rules related to booleans and natural numbers

check that in all the typing rules above and in those that follow, this will always be the case. This is useful because it facilitates the implementation of the type-checker, mostly because it makes it easier to check terms for equality (which is needed often during type-checking, e.g. in rule (T-App) where the type of t_2 and the type of the argument of t_1 need to be equal. Remember: we always need to consider a term like if true then Nat else Bool equal to Nat. The two terms don't look the same because the left one is not in normal form. Comparing normal terms for equality is much easier: they need to be syntactically equal (see section 6.2).

9 Polymorphism

With the dependent types machinery, one thing we get for free is polymorphism (similar to *generics* in Java). This means that we can define functions which can be used on different types. One of the examples in the provided unit tests is the polymorphic identity function:

$$\mathrm{let}\, id : (A : Set) \to A \to A = \lambda A : Set.\lambda x.x\, \mathrm{in} \ldots$$

This function is polymorphic: it can be used on booleans:

$$id\, \mathrm{Bool}\, \mathrm{true}$$

but also for example on naturals:

$$id \, \mathrm{Nat} \, 3$$

In fact, it is similar to the following generic Java method (ignore this if you don't know Java generics):

$$\text{public } \langle T \rangle \; T \; id(\; T \; t \;) \; \{ \text{ return } t; \}$$

This example is just an explanation, you are not expected to do something with this function for the assignment. Note that many of the unit tests show interesting examples like this. Try to understand how they all work.

10 Natural Induction

Without a general recursion construct (fix), there is actually little we can do with our natural numbers. For example, there is no way to define what it means to take the sum of two naturals. We've already discussed that we cannot add fix to our calculus. So are we stuck? Luckily, we are not. We can allow a form of recursion over naturals which is guaranteed to terminate. In mathematics, this is known as natural induction. We will do this by adding a primitive construct natInd.

Figure 9 shows the typing rule for natInd. The primitive first accepts a predicate P over naturals. P defines a set for any natural number n. natInd will give us a function that yields a value of $P \; n$ for any n, if we give it a base case and an induction step (remember: this is how you've always used natural induction in high school mathematics). The base case is just a value of $P \; 0$, which natInd expects as a second argument. The third argument of natInd is the induction step. This should be of type $(n : \mathrm{Nat}) \to P \; n \to P(\mathrm{succ} \, n)$. This means that the induction step gets a number n and a proof of the predicate P for n and should deliver a value of the predicate P for $\mathrm{succ} \, n$ $(= n + 1)$. Given these arguments, natInd gives a function that returns a value of $P \; n$ for any given $n : \mathrm{Nat}$.

natInd needs proper evaluation rules to work. However, we will not give them in this text, but it is part of this assignment for you to determine them! We expect you to write them down in a similar form as the evaluation rules in the rest of this text, and hand them in as part of this assignment. You are allowed to use a system like LaTeX[8] for this, but we recommend you write them by hand (readably) on paper. Additionally, your implementation needs to correctly execute the rules. As a hint, we suggest you think about what a correct value would be of natInd P *baseCase indStep* zero, given correctly typed values of P, *baseCase* and *indStep* and similarly (but harder ;)), what would be a correct value of natInd P *baseCase indStep* $(\mathrm{succ} \, n)$?

[8]We recommend the LaTeX package *semantic* and its support for inference rules, should you want to use LaTeX.

$$\mathrm{natInd} : (P : \mathrm{Nat} \to \mathrm{Set}) \to P\ 0 \to ((n : \mathrm{Nat}) \to P\ n \to P(\mathrm{succ}\, n)) \to (n : \mathrm{Nat}) \to P\ n$$

Figure 9: Typing rule for natInd

Using natInd, we can define summation of natural numbers:

$$\begin{aligned}
&\mathrm{let}\, plus : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\\
&\qquad = \mathrm{natInd}\ (\lambda n : \mathrm{Nat}\,.\,\mathrm{Nat} \to \mathrm{Nat})(\lambda x : \mathrm{Nat}\,.x)\\
&\qquad\quad (\lambda n : \mathrm{Nat}\,.\lambda h : \mathrm{Nat} \to \mathrm{Nat}\,.\lambda v : \mathrm{Nat}\,.\,\mathrm{succ}\ (h\ v))\\
&\mathrm{in}\ldots
\end{aligned}$$

As an optional exercise use *plus* and natInd to implement *times*: multiplication of natural numbers.

As another optional exercise, show that if we have natInd, we don't actually need pred as a builtin primitive by implementing a value (called *pred*2) in the calculus which behaves the same as pred and has the same type.

11 Singleton types (optional)

For actually writing proofs in our calculus, we need to add another ingredient in the mix: singleton types also known as equality proofs. Figure 10 shows the required new primitives, evaluation rules and typing rules. The type I $A\ x\ y$ is the type of proofs of equality of two values x and y of type A. The primitive refl is the only way to directly construct such an equality proof. From the type of refl, we see that refl $A\ x$ is a proof that value x of type A is equal to itself. As a final ingredient, we also need to be able to exploit the equality between two values, and this is done using the subst primitive. From its type, we see that it takes a type A, values x and y of type A, a type constructor P (mapping values of type A to types) and a proof of equality of x and y and will then map a value of P for value x to a value of P for y.

Add singleton types to your calculus.

Singleton types are important for using our dependently-typed calculus for machine-checked theorem proving. An example is the following proof of the associativity of the *plus* function we defined earlier. We will explain how it works step by step below.

```
1 let $plus : \text{Nat} \to \text{Nat} \to \text{Nat} = \ldots$ (see before)
2 let $prf : (n : \text{Nat}) \to (m : \text{Nat}) \to (k : \text{Nat}) \to \text{I Nat}\,(plus\,n\,(plus\,m\,k))(plus\,(plus\,n\,m)\,k) =$
3       natInd $(\lambda n : \text{Nat}\,.(m : \text{Nat}) \to (k : \text{Nat}) \to \text{I Nat}\,(plus\,n\,(plus\,m\,k))(plus\,(plus\,n\,m)\,k))$
4          $(\lambda m : \text{Nat}\,.\lambda k : \text{Nat}\,.\,\text{refl Nat}\,(plus\ m\ k))$
5          $(\lambda n : \text{Nat}\,.$
6           $\lambda hyp : (m : \text{Nat}) \to (k : \text{Nat}) \to \text{I Nat}\,(plus\,n\,(plus\,m\,k))(plus\,(plus\,n\,m)k).$
7           $\lambda m : \text{Nat}\,.\lambda k : \text{Nat}\,.\,\text{subst Nat}\,(plus\,n\,(plus\,m\,k))\ (plus\,(plus\,n\,m)\,k)$
8            $(\lambda t : \text{Nat}\,.\,\text{I Nat (succ}\ (plus\ n\ (plus\ m\ k)))\ (\text{succ}\ t))$
9            $(hyp\ m\ k)\ (\text{refl Nat (succ}\ (plus\ n\ (plus\ m\ k)))))$
```

The code uses the definition of *plus* that we saw earlier (we don't repeat it here for brevity). It then defines a value *prf* of type $(n : \text{Nat}) \to (m : \text{Nat}) \to (k : \text{Nat}) \to \text{I Nat}\ (plus\ n\ (plus\ m\ k))(plus\ (plus\ n\ m)\ k)$. This type represents the associativity property of our *plus* function, and the value we will construct is a proof of it. This value is constructed using the natural indicution primitive natInd. The first argument of natInd on line 3 is the predicate that we will prove by induction: for any n, we will prove that for all m and k, $n + (m + k)$ is equal to $(n + m) + k$. The second argument in the call to natInd is the base case of the induction on line 4: the proof that for $n = 0$, associativity holds. For $n = 0$, the property to prove is the fact that $0 + (m + k)$ is equal to $(0 + m) + k$, but by the definition of our *plus* function, both of these are equal to $m + k$. We therefore construct a proof of the base case using the refl primitive.

The induction step of the call to natInd is on lines 5–9. It is a function which for a given n, and using the induction hypothesis *hyp* which says that the property holds for n, returns a proof that the property holds for succ n. This means the function should return a proof that $plus\ (\text{succ}\ n)\ (plus\ m\ k)$ is equal to $plus\ (plus\ (\text{succ}\ n)\ m)\ k$. But by the definition of our *plus* function, these are respectively equal to $\text{succ}\ (plus\ n\ (plus\ m\ k))$ en $\text{succ}\ (plus\ (plus\ m\ n)\ k)$. Since our induction hypothesis says that the respective arguments to succ are equal, all we need to do is conclude from this that their successors are equal. To do this, we exploit the *subst* primitive. What we do is provide a value of $\text{I Nat (succ}\ (plus\ n\ (plus\ m\ k)))(\text{succ}\ (plus\ n\ (plus\ m\ k)))$ using the refl primitive (this is the final argument on line 9) and then use subst with our induction hypothesis to "cast" this to a proof of $\text{I Nat (succ}\ (plus\ n\ (plus\ m\ k)))(\text{succ}\ (plus\ (plus\ n\ m)\ k))$. This means we have to use subst with the predicate defined on line 8, because by replacing t with both values of the equality proof $hyp\ m\ k$ we get the modified types we want.

All of the above is pretty hard to work with. In many real dependently-typed programming languages, there exist facilities to make this easier to work with (e.g. pattern matching in Agda). However, it is still useful to know what is actually happening under the hood: the things we show in this section.

Syntax:

$s, t ::=$	**terms**
...	
refl	reflexivity constructor
I	singleton type constructor
subst	substitutivity

Evaluation rules:

$$\frac{}{\text{subst}\ \ A\ x\ y\ p\ (\text{refl}\ \ A\ z)\ px \longrightarrow px} \qquad \text{(E-Subst)}$$

Typing rules:

$$\frac{}{\Gamma \vdash \text{I} : (A : \text{Set}) \to A \to A \to \text{Set}} \qquad \text{(T-I)}$$

$$\frac{}{\Gamma \vdash \text{refl} : (A : \text{Set}) \to (x : A) \to \text{I}\ \ A\ x\ x} \qquad \text{(T-Refl)}$$

$$\frac{}{\Gamma \vdash \text{subst} : (A : Set) \to (x : A) \to (y : A) \to (P : A \to Set) \to \text{I}\ \ A\ x\ y \to P\ x \to P\ y} \qquad \text{(T-Subst)}$$

Figure 10: Singleton types: new constructs, evaluation rules, typing rules

As an optional exercise, construct a proof of the following mathematical theorem. Note: this is probably be the hardest exercise in this assignment. The proof is easier than the associativity proof above though.

Theorem 1 (right zero for +) *For all $n \in \mathbb{N}$, $n + 0$ is equal to n.*

Such a proof in our system is a value of type $(n : \text{Nat}) \to \text{I}\ \ \text{Nat}\ (\mathit{plus}\ n\ 0)\ n$. Use the *plus* that we have defined using natInd in section 10. As a hint (if it isn't obvious): you need to prove this theorem using natural induction, so you need to use the natInd primitive again. In the induction step, you will need to make creative use of the subst primitive like we did above.

12 Dependent if (Optional)

Before, we've mentioned that the example $\lambda x : \text{Bool} . \text{if}\, x\, \text{then}\, 3\, \text{else}\, \text{true}$ could be given the type $(x : \text{Bool}) \to \text{if}\, x\, \text{then}\, \text{Nat}\, \text{else}\, \text{Bool}$ in a dependently-typed language. However, if you have paid close attention to the typing rules in Figure 8, you will have noticed that this is not actually allowed by our system. Typing rule T-If requires both branches of the if to be of the same type.

In order to actually build values of type $(x : \text{Bool}) \to \text{if}\, x\, \text{then}\, \text{Nat}\, \text{else}\, \text{Bool}$, we will not modify the typing rules of $\text{if}\, t_1\, \text{then}\, t_2\, \text{else}\, t_3$. Instead, we will introduce a construct similar to natInd, but for booleans. As an optional exercise: add the construct boolElim of the following type to the calculus, with reasonable evaluation rules and typing rules:

$$\text{boolElim} : (P : \text{Bool} \to \text{Set}) \to P\ \text{true} \to P\ \text{false} \to (b : \text{Bool}) \to P\ b$$

Note that you can see this boolElim construct as a generalised if construct. Indeed, $\text{if}\, t_1\, \text{then}\, t_2\, \text{else}\, t_3$ roughly corresponds to boolElim $(\_{:}Bool.A)\ t_2\ t_3\ t_1$

As another optional exercise: use this new construct to construct a value of type $(x : \text{Bool}) \to \text{if}\, x\, \text{then}\, \text{Nat}\, \text{else}\, \text{Bool}$.

13 Practical info

13.1 Online Support

There will be an online support forum for the project which you can access through Toledo. This is the only place where you can ask questions about the project (except for practical issues), so that any extra guidance we may provide is available to anyone. We expect you to follow the forum while working on the project and take into account any additional feedback provided there.

13.2 Deliverables

In summary, if you choose this assignment, we expect you to hand in the following deliverables.

- Your solution to the extra question at the end of the Agda exercise session.
- Your implementation of this assignment following the template we provided and such that all the unit tests succeed (except for those pertaining to optional parts of the assignment which you choose to not execute).
- Your version of the elimination rules for natInd as discussed in section 10. If you make this on paper, you can hand this in physically at the CS secretariat, addressed to prof. Frank Piessens. Don't forget to write your name on it.

13.3 Deadline

You will find more information about the deadline on Toledo.

14 Further Reading

Should you want to know more about dependent types, we recommend the following texts:

- Dependently typed programming in Agda, Ulf Norell, `http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`
- Why dependent types matter, Thorsten Altenkirch, Conor McBride, James McKinna, `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`

References

[1] B.C. Pierce. *Types and programming languages.* MIT Press, 2002.