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

Jesper Cockx
Dominique Devriese
Frank Piessens

November 3, 2014

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 5.

1.1 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 [2].

1.2 Dependent types

In his well-known textbook on programming languages and types [2], 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 such as Coq[1], automatically checking the correctness of mathematical proofs, but they are also increasingly being used as programming languages, see for example Agda[2] and Idris[3].

As an example, probably the most elaborate program developed in a dependently typed language is CompCert[4]: an elaborate, optimizing research C compiler implemented in Coq 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[5].

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.

1.3 Overview

In section 2, we describe the code package we provide as part of this assignment. It contains an incomplete implementation of a dependently typed calculus that we expect you to use as a starting point. Sections 3.1, 3.2 and 3.3 describe the syntactic constructs, the evaluation rules and the typing rules of the calculus respectively. Section 4 describes extensions to the base calculus with booleans and natural numbers (4.1), sigma types (4.2) and identity types (4.3).

2 The code we provide

The package contains a couple of empty or partially implemented classes which we expect you to implement. There is also a set of unit tests which we think can help you test your solution to this assignment. They also show at times interesting examples. In order to run the tests in Eclipse, you may have to install the ScalaTest plugin from the repository at `http://download.scala-ide.org/sdk/helium/e38/scala211/stable/site`. You may not modify the existing 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

[1]`http://coq.inria.fr/`

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

[3]`http://www.idris-lang.org/`

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

[5]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.

clearly marked as such in the unit test class. You may of course also add your own unit tests.

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.

3 The Core Calculus

3.1 Syntax

Figure 1 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:

- *Dependent function types*: 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 $\lambda b : \mathrm{Bool}\,.\,\mathrm{if}\, b\, \mathrm{then}\, 3\, \mathrm{else}\, \mathrm{true}$. Its type could be $(b : \mathrm{Bool}) \to \mathrm{if}\, b\, \mathrm{then}\, \mathrm{Nat}\, \mathrm{else}\, \mathrm{Bool}$. More on this in section 4.1.2[6].

  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, this extra bound variable has to be taken 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). Only the substitution and shifting of de Bruijn indices in respectively the result type and the body are similar.

- 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

[6]Note that this example $\lambda b : \mathrm{Bool}\,.\,\mathrm{if}\, b\, \mathrm{then}\, 3\, \mathrm{else}\, \mathrm{true}$ 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
	Set	type of types

Figure 1: Dependently-typed calculus: basic syntax

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 $t_1 : \mathrm{Set}$. We will also see that for example the type of naturals is of type Set:

$$\mathrm{Nat} : \mathrm{Set}$$

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[7].

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:

$$t_1 \to t_2 \qquad \text{is translated to} \qquad (x : t_1) \to t_2 \qquad \text{where } t_2 \text{ does not refer to } x$$

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

3.2 Evaluating our calculus

Any typechecker certainly needs to be able to check whether two types are equal. In a dependently typed language, these types can contain terms, so the typechecker also needs to check whether two *terms* are equal. For example, $\lambda x : \mathrm{Bool}\,.\,\mathrm{if}\ x\ \mathrm{then}\ 3\ \mathrm{else}\ \mathrm{true}$ could be typed $(x : \mathrm{Bool}) \to \mathrm{if}\ x\ \mathrm{then}\ \mathrm{Nat}\ \mathrm{else}\ \mathrm{Bool}$. Now, when we apply this lambda to the value true, the type-checker should see that its type $\mathrm{if}\ \mathrm{true}\ \mathrm{then}\ \mathrm{Nat}\ \mathrm{else}\ \mathrm{Bool}$ (with $x = \mathrm{true}$ filled in) is the same as Nat. The implementation we give you to start with doesn't evaluate terms during typechecking, so this is something you should add.

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

$$\frac{t_2 \longrightarrow t_2'}{\lambda x : t_1.t_2 \longrightarrow \lambda x : t_1.t_2'} \quad \text{(E-Abs1)}$$

$$\frac{t_1 \longrightarrow t_1'}{\lambda x : t_1.t_2 \longrightarrow \lambda x : t_1'.t_2} \quad \text{(E-Abs2)}$$

$$\frac{t_2 \longrightarrow t_2'}{(x : t_1) \to t_2 \longrightarrow (x : t_1) \to t_2'} \quad \text{(E-Pi1)}$$

$$\frac{t_1 \longrightarrow t_1'}{(x : t_1) \to t_2 \longrightarrow (x : t_1') \to t_2} \quad \text{(E-Pi2)}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1 t_2 \longrightarrow t_1 t_2'} \quad \text{(E-App1)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2} \quad \text{(E-App2)}$$

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

Figure 2: Evaluation rules for the basic parts of our calculus

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 [2, §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[8].

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" abstractions, something which a typical lambda calculus doesn't do (see e.g. [2, Figure 5-3]).

The evaluation rules for the core calculus are shown in Figure 2 and 7. 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

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

on the efficiency and memory use of the evaluation. Therefore, you are free to experiment with different evaluation strategies (call-by-value, call-by-name, call-by-need, ...).

In what follows, we will write $s \longrightarrow^* t$ to mean that s evaluates to t, in zero or more steps, i.e. if we apply a number of evaluation rules to s, we get t.

3.3 Type checking

Figure 3 shows the typing rules for the core 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. However, since we have dependent types, the types of the variables in the context are allowed to depend on the *values* of the variables before it.

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. Then, the 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 t_1 is a type, 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 the rules for $\lambda x : t_1.t_2$ and $(x : t_1) \rightarrow t_2$.

The type checking rule (T-App) for applications $t_1 t_2$ is different from the one in a standard lambda calculus, because the type t_3 of the function t_1 should be equal to (i.e. *evaluate to*) a Π-type $(x : t_4) \rightarrow t_5$. In that case, the argument t_2 must be of type t_4 as usual, but the application $t_1 t_2$ will have as type $[x \mapsto t_2]t_4$: the value of the Π-type for the argument that the function was applied to.

Finally, the rule (T-SetInSet) 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, so we choose to ignore the issue here, instead of implementing the (relatively complex) solution for this[9].

3.3.1 A bidirectional typechecker

It can often become very tedious to write out the type of each variable that is bound by a lambda. For this reason, our calculus also allows unannotated lambdas of the form $\lambda x.t$ in places where the type of x can be inferred. Like other dependently typed languages, we will use a *bidirectional* typechecker [1]. A bidirectional typechecker has two modes: *checking mode*, where a term is checked

[9]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, $\text{Set}_0 : \text{Set}_1$, $\text{Set}_1 : \text{Set}_2$ etc.

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

$$\frac{\Gamma \vdash t_1 : \text{Set} \qquad \Gamma, x : t_1 \vdash t_2 : t_3}{\Gamma \vdash \lambda x : t_1 . t_2 : (x : t_1) \to t_3} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \text{Set} \qquad \Gamma, x : t_1 \vdash t_2 : \text{Set}}{\Gamma \vdash (x : t_1) \to t_2 : \text{Set}} \qquad \text{(T-Pi)}$$

$$\frac{\Gamma \vdash t_1 : t_3 \qquad t_3 \longrightarrow^* ((x : t_4) \to t_5) \qquad \Gamma \vdash t_2 : t_4}{\Gamma \vdash t_1\ t_2 : t_5[x \mapsto t_2]} \qquad \text{(T-App)}$$

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

Figure 3: Typing rules for the basic parts of our calculus

against a given type, and *inference mode*, where the type is extracted from a term. The typechecker can switch between these two modes when necessary. For example, when inferring the type of an application $t_1\ t_2$, the typechecker will first infer the type of t_1, check that it is of the form $(x : t_3) \to t_4$, and then check t_2 against the type t_3. If this succeeds, the inferred type of the entire expression is then $t_4[x \mapsto t_2]$.

It is possible to annotate the typing rules for our language with information about which rules can be applied in inference mode and which rules must be applied in checking mode. If the type t can be inferred from an expression e in a (given) context Γ, this is written as $\Gamma \vdash e \Uparrow t$, otherwise we write $\Gamma \vdash e \Downarrow t$. For example, the bidirectional typing rule for the application becomes

$$\frac{\Gamma \vdash t_1 \Uparrow t_3 \qquad t_3 \longrightarrow^* ((x : t_4) \to t_5) \qquad \Gamma \vdash t_2 \Downarrow t_4}{\Gamma \vdash t_1\ t_2 \Uparrow t_5[x \mapsto t_2]}$$

You should give a directed version of each of the rules in Figure 3. There should also be a rule to switch from checking mode to inference mode when necessary! Write these functions down on paper or in LaTeX and hand them in as described in Section 5.

The bidirectional typechecking algorithm is implemented by the function $tcTerm(t : Term, a : Option[Term], ctx : Context) : (Term, Term)$. It takes a term t, an optional type a and a context ctx containing the types of the free variables. If a is $None$, the typechecker is in inference mode, while if it is $Some(a')$, the term t is checked agains the type a'. The function returns a pair of an elaborated term (which is equal to the input term t, except that all

$$\frac{\Gamma \vdash t_1 : \mathrm{Set} \qquad \Gamma \vdash t_2 : t_1 \qquad \Gamma \vdash [x \mapsto t_2]t_3 : t_4}{\Gamma \vdash \mathrm{let}\, x : t_1 = t_2 \,\mathrm{in}\, t_3 : t_4} \qquad \text{(T-LET)}$$

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

Figure 4: Evaluation and typing rule for let-expressions

lambdas in it have been annotated with a type) and the resulting type.

The *tcTerm* makes multiple uses of the function *equalTerms*, which checks that two given terms are equal. However, the implementation of *equalTerms* in the code you are given is incomplete: it just checks whether the two terms have the same *syntax*, instead of checking whether they evaluate to the same form. You should add an evaluator to the language, and use it in the implementation of *equalTerms*.

3.4 Let-expressions

Now that we have a core calculus, we can start adding new language features to it. The first new syntax you have to add to the core language are let-expressions. The syntax of a let-expression is $\mathrm{let}\, x : t = t \,\mathrm{in}\, t$. Instead of translating a let-expression $\mathrm{let}\, x : t_1 = t_2 \,\mathrm{in}\, t_3$ 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, check that t_2 is of type t_1, but then we do something special. Instead of 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 optional parts of the assignment.

3.5 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):

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

Note that many of the unit tests show interesting examples like this. Try to understand how they all work.

3.6 Typed Church encodings

Even though the calculus we have now is very minimalistic, it is still possible to write many interesting programs in it. In particular, we can encode standard algebraic data types (such as booleans and natural numbers) by using *(typed) Church encodings.* For example, the type of Church booleans is defined as $bool = (A : Set) \to A \to A \to A$, with terms $tru = \lambda A.\lambda x.\lambda y.x$ and $fls = \lambda A.\lambda x.\lambda y.y$. Similarly, the Church encoding of natural numbers is defined as $nat = (A : Set) \to A \to (A \to A) \to A$ where $ze = \lambda A.\lambda x.\lambda y.x$ and $su = \lambda n.\lambda A.\lambda x.\lambda y.y(n\ A\ x\ y)$. As an exercise, we ask you to implement the following functions on the Church encodings

- $not : bool \to bool$
- $and : bool \to bool \to bool$
- $or : bool \to bool \to bool$
- $boolEq : bool \to bool \to bool$
- $isZero : nat \to bool$
- $plus : nat \to nat \to nat$

4 Extensions

4.1 Booleans and natural numbers

Although we have seen that the basic calculus can be used to encode many types, it is often more convenient and efficient to have native representations of standard data types. So we define some extensions for working with (native) booleans and natural numbers. Figures 5 and 6 show the additional syntactic constructs. We have the types Bool and Nat, as well as standard primitive constructs as defined by Pierce [2, §3].

The parser you are given already knows how to parse these terms, and calls the corresponding functions in the DepCalculus file. It is your job to add the syntax for them to the calculus, together with their evaluation and typing rules.

booleans

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

Figure 5: 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 6: Dependently-typed calculus: natural numbers

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

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

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

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

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

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

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

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

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

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

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

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

Figure 7: Evaluation rules related to booleans and natural numbers

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

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

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

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

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

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

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

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

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

Figure 8: Typing rules related to booleans and natural numbers

4.1.1 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 and the evaluation rules 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 $(n : \mathrm{Nat}) \rightarrow P\ n \rightarrow P(\mathrm{succ}\, n)$. 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 $n : \mathrm{Nat}$.

$$\frac{}{\text{natInd } P \; \mathit{base} \; \mathit{step} \; \text{zero} \longrightarrow \mathit{base}} \quad \text{(E-natIndZero)}$$

$$\frac{}{\text{natInd } P \; \mathit{base} \; \mathit{step} \; (\text{succ } n) \longrightarrow \mathit{step} \; n \; (\text{natInd } P \; \mathit{base} \; \mathit{step} \; n)} \quad \text{(E-natIndZero)}$$

$$\frac{}{\text{natInd} : (P : \text{Nat} \to \text{Set}) \to P \; 0 \to ((n : \text{Nat}) \to P \; n \to P(\text{succ}\, n)) \to (n : \text{Nat}) \to P \; n} \quad \text{(T-natInd)} \tag{1}$$

Figure 9: Typing and evaluation rules for natInd

Using natInd, we can define summation of natural numbers:

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

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.

4.1.2 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 $(\lambda b : \mathit{Bool}.A) \; t_2 \; t_3 \; t_1$

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

4.2 Sigma types

Product types (also called *pair types*) are a way to build more complex types from simple ones. For example, $\mathrm{Bool} \times \mathrm{Nat}$ is the types of pairs (x, y) where $x : \mathrm{Bool}$ and $y : \mathrm{Nat}$. In order to extract the components of a pair, there are two primitive functions $\mathrm{fst} : A \times B \to A$ and $tsnd : A \times B \to B$ called the *projections*.

Like for function types, there is also a dependently typed variant of product types called *sigma types*. A sigma type is a product type where the type of the second component can be dependent on the value of the first component. For example, $\Sigma[b : \mathrm{Bool}]$ $(\mathrm{if}\, b\, \mathrm{then}\, \mathrm{Bool}\, \mathrm{else}\, \mathrm{Nat})$ is the type of pairs (x, y) where $x : \mathrm{Bool}$ and $y : \mathrm{Bool}$ if $b = \mathrm{true}$ or $y : \mathrm{Nat}$ if $b = \mathrm{false}$. Also like for function types, the non-dependent product type can be defined in terms of sigma types by dropping the dependency: $A \times B = \Sigma[x : A]\ B$ where x is not free in B.

We ask you to add sigma types to the calculus. Again, the parser can already parse sigma types, so you still have to add the syntax, the evaluation rules, and the typing rules. When writing the typing rules, pay special attention to which rules can be used in inference mode and which ones can only be used in checking mode.

4.3 Identity types (optional)

For actually writing proofs in our calculus, we need to add another ingredient in the mix: identity types, also known as equality proofs. Figure 11 shows the required new primitives, evaluation rules and typing rules. The type $\mathrm{I}\ \ A\ x\ y$ 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 $\mathrm{refl}\ \ A\ x$ 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. The goal of this optional part is to add identity types to your calculus.

Identity 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

Syntax:

$s, t ::=$		**terms**
	...	
	(s, t)	pair constructor
	$\Sigma[t_1 : t_2]\ t_3$	sigma type constructor
	$\text{fst}\ \ t$	first projection
	$\text{snd}\ \ t$	second projection

Evaluation rules:

$$\frac{}{\text{fst}\ (s, t) \longrightarrow s} \qquad \text{(E-FST)}$$

$$\frac{}{\text{snd}\ (s, t) \longrightarrow t} \qquad \text{(E-SND)}$$

Typing rules:

$$\frac{\Gamma \vdash A : Set \qquad \Gamma(x : A) \vdash B : Set}{\Gamma \vdash \Sigma[x : A]\ B : Set} \qquad \text{(T-SIGMA)}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : [x \mapsto s]B}{\Gamma \vdash (s, t) : \Sigma[x : A]\ B} \qquad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash t : \Sigma[x : A]\ B}{\Gamma \vdash \text{fst}\ \ t : A} \qquad \text{(T-FST)}$$

$$\frac{\Gamma \vdash t : \Sigma[x : A]\ B}{\Gamma \vdash \text{snd}\ \ t : [x \mapsto \text{fst}\ \ t]B} \qquad \text{(T-SND)}$$

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

works step by step below.

```
1 let plus : Nat → Nat → Nat = ...  (see before)
2 let prf : (n : Nat) → (m : Nat) → (k : Nat) → I Nat (plus n (plus m k))(plus (plus n m) k) =
3         natInd (λn : Nat .(m : Nat) → (k : Nat) → I Nat (plus n (plus m k))(plus (plus n m) k))
4             (λm : Nat .λk : Nat . refl  Nat (plus m k))
5             (λn : Nat .
6               λhyp : (m : Nat) → (k : Nat) → I Nat (plus n (plus m k))(plus (plus n m)k).
7               λm : Nat .λk : Nat . subst Nat (plus n (plus m k)) (plus (plus n m) k)
8                 (λt : Nat . I  Nat (succ (plus n (plus m k))) (succ  t))
9                 (hyp m k) (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 $(n : \mathrm{Nat}) \to (m : \mathrm{Nat}) \to (k : \mathrm{Nat}) \to \mathrm{I}\ \mathrm{Nat}\ (plus\ n\ (plus\ m\ k))(plus\ (plus\ n\ m)\ k)$. 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 $plus\ (\mathrm{succ}\ \ n)\ (plus\ m\ k)$ is equal to $plus\ (plus\ (\mathrm{succ}\ \ n)\ m)\ k$. But by the definition of our *plus* function, these are respectively equal to $\mathrm{succ}\ (plus\ n\ (plus\ m\ k))$ en $\mathrm{succ}\ (plus\ (plus\ n\ m)\ k)$. 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 $\mathrm{I}\ \mathrm{Nat}\ (\mathrm{succ}\ (plus\ n\ (plus\ m\ k)))(\mathrm{succ}\ (plus\ n\ (plus\ m\ k)))$ 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 $\mathrm{I}\ \mathrm{Nat}\ (\mathrm{succ}\ (plus\ n\ (plus\ m\ k)))(\mathrm{succ}\ (plus\ (plus\ n\ m)\ k))$. 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	identity type constructor
	subst	substitutivity

Evaluation rules:

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

Typing rules:

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

$$\frac{}{\Gamma \vdash \text{refl} : (A : \text{Set}) \to (x : A) \to \mathrm{I}\;\; A\; x\; x} \qquad \text{(T-R{\scriptsize EFL})}$$

$$\frac{}{\Gamma \vdash \text{subst} : (A : Set) \to (x : A) \to (y : A) \to (P : A \to Set) \to \mathrm{I}\;\; A\; x\; y \to P\; x \to P\; y} \qquad \text{(T-S{\scriptsize UBST})}$$

Figure 11: Identity 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 $(n : \text{Nat}) \to \mathrm{I}\;\; \text{Nat}\; (\mathit{plus}\; n\; 0)\; n$. Use the *plus* that we have defined using natInd in section 4.1.1. 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.

5 Practical info

5.1 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.

5.2 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 4.1.1. If you make this on paper, you can hand this in physically at the CS secretariat. Don't forget to write your name on it.

5.3 Deadline

You will find more information about the deadline on Toledo.

6 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] Frank Pfenning. Lecture Notes on Bidirectional Type Checking. 2004.

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