Checking proofs with a computer

Hartevelt, D.

Award date:
2009

Checking Proofs with a Computer

Dennis Hartevelt

11-08-2009

Master's Thesis

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Supervisor: prof. dr. J.C.M. Baeten
Tutor: dr. R.P. Nederpelt Lazarom
Advisors: prof. dr. J.H. Geuvers
dr. F. Wiedijk

Acknowledgements

This research has been performed in fulfillment of the requirements for the degree of Master of Science (Ir.) at the Eindhoven University of Technology. I wish to thank my supervisor prof. dr. J.C.M. Baeten for the opportunity to fulfill my master project within the department of Formal Methods. I also want to thank my tutor dr. R.P. Nederpelt Lazarom and my advisors prof. dr. J.H. Geuvers and dr. F. Wiedijk for all the time and help during the master project.

Contents

Chapter 1

Introduction

This report is the master thesis of the master project called 'Checking proofs with a computer'. The master project is fulfilled within the department of Formal Methods. At the beginning of the master project, the following project description is formulated.

Project description 1.0.1 *Main subject in the TUE master course Proving with Computer Assistance was the derivation system λC (due to H.P. Barendregt) for giving proofs in typed λ-calculus. This derivation system has been extended by R.P. Nederpelt with rules for axioms and definitions, leading to the system λD^+. These new rules have currently been added to λC in a draft textbook by Nederpelt and Geuvers ([Ned]), together with a number of examples explaining how mathematical texts - including axioms and definitions - may be formalized in λD^+.*

In this assignment, λD^+ should be implemented in order to get a prototype computer tool for checking texts in λD^+. In particular, the tool should be able to check the mentioned example texts.

A publication of the tool at the web may be considered, but is not part of this master assignment; in particular, there are no requirements on the degree of sophistication of the interface.

The main concern in the design of the tool must be its faithfulness as regards the handling of the derivation rules: the tool should transparently reflect these rules and their actions. For this purpose, the tool should have a well-demarcated kernel in which these rules are easily recognizable.

A computer program that can verify the correctness of mathematical proofs is called a proof checker. In this master project we design and implement a proof checker that can verify mathematical proofs in λD^+. The examples of the new book of Nederpelt and Geuvers have eventually been checked on their correctness with the λD^+ proof checker.

In the master thesis we did not immediately start with the λD^+ proof checker. We start by designing a proof checker for the 'less complex' λ-calculus λC. As given in the project description λD^+ is an extension of λC. The λC proof checker is used as a basis for the λD^+ proof checker. For this we designed the λC proof checker in such a way that we can easily extend the proof checker to λD^+.

A mathematical proof in λC and λD^+ is formulated in a judgement. With both proof checkers we are able to verify the correctness of a judgement, by constructing a so called

derivation tree. In both proof checkers we use a deterministic and recursive algorithm to compute the derivation tree of the judgement. To make the algorithm deterministic we have strengthened the *weak* derivation rule of λC and the *env-weak* and *env-weak*$^{\perp\!\!\!\perp}$ derivation rules of λD^+. In order to make the algorithm recursive, we added a typing algorithm, that finds given a context Γ and a type M, the type of M.

If we are able to compute a complete derivation tree for the input judgement, it follows that the judgement contains no errors and we call the input judgement derivable (with the derivation rules). The algorithm that verifies the correctness of a mathematical proof is therefore called the derivability check. If we are not able to compute a derivation tree for the input judgement, the input judgement contains errors. After the derivability check, the derivation tree in reported in list notation and if the input judgement contains an error a suitable error message is given.

Before we start the derivability check of a judgement, one must enter a judgement in the proof checker. For both proof checkers we designed an input language. The input language of the λC proof checker is not user friendly but enables us to verify the syntax with an LL(1) parser. For the λD^+ we first extended the input language of the λC proof checker, but finally we implemented a user friendly input language that is comparable with the appearance of the λ-calculus λD^+.

Next to implementing the λ-calculi, the λD^+ proof checker contains some user friendly features. In λD^+ we are able to formulate axioms and definitions. For this we can maintain a library of derivable axioms and definitions for the λD^+ proof checker. The user can use the axioms and definition from the library when formulating a formal proof in λD^+. In this way the performance of the derivability check increases and the user does not have to formulate all needed axioms and definitions for each proof by hand.

This master thesis will describe both proof checkers. The Chapters 2 and 3 are used as an introduction for this master thesis. In Chapter 2 we describe the need of computer programs that can verify formal proofs on their correctness. In the chapter we will indicate, where we can use proof checkers in finding and recording a proof. In Chapter 3 we will give some background concerning λ-calculus. The chapter gives a brief introduction in λC and λD^+. For both λ-calculi, we describe the most important properties that are necessary to implement the proof checkers.

Next, we give a description of the two proof checkers. Each proof checker is described in two parts. First we give the functional description, followed by a description of the implementation. In Chapter 4 the functional description of the λC proof checker is given. Whereas the description of the implementation is included in Chapter 5.

After the description of the λC proof checker, we continue with the λD^+ proof checker. Chapter 6 gives the functional description of the λD^+ proof checker and Chapter 7 describes the implementation.

We conclude the master thesis with some conclusions and future work, in Chapter 8.

Chapter 2

Automated proofs

If one science is exact then it is mathematics. If a mathematical theorem is proven then this theorem holds forever. In other scientific fields this does not always hold, because their theorems depend on assumptions that are not absolute. A mathematical proof is identified to be correct, if a group of mathematicians have checked the proof on its correctness.

In 1998 the mathematician Thomas Hales proved the Kepler Conjecture. The Kepler Conjecture says that the pattern grocers use to stack oranges, packs the most oranges into the smallest place. The Kepler Conjecture states a simple problem, but is very hard to prove. The proof of Hales consists of 300 pages of text and 40.000 lines of computer code. Four years a group of twelve mathematicians tried to check the proof of Thomas Hales. Finally they conclude that they were certain that the proof was correct for ninety-nine procent. But since mathematics is exact, ninety-nine procent is not enough.

And mathematical proofs will only become bigger and more complex. Therefore it is likely that the group of mathematicians cannot guarantee the correctness for all future mathematical proofs. A possible solution to retain the one-hundred procent certainty, is the use of computer programs that are able to verify mathematical proofs.

2.1 Proof assistants and proof checkers

A computer program that is able to verify mathematical proofs is called a *proof checker*. The program verifies formal proofs in their corresponding proof language. The proof language is program dependent and consists of a syntax and one or more axioms and derivation rules.

In order to make the formulation of a proof more feasible an interactive "proof-development system" is needed. This is a proof environment that stands next to the proof checker and helps the human to develop proofs. The combination of a "proof-development system" and a proof checker is called a *proof assistant*. The proof assistant differs from a *theorem prover*. A theorem prover consists of a set of well chosen decision procedures that allow formulas of a specific restricted format to be proved automatically. Automated theorem provers are powerful, but have limited expressivity. This means that there is no way to set-up a generic mathematical theory in such a system.

For both programs the main goal is to verify the validity of mathematical proofs. However proof assistants do not generate the proofs themselves, but take care that the user is able to construct a proof interactively with the system. During the construction of the proof, the proof assistant maintains the progress of the proof and checks with the proof checker if the

proof does not have any errors.

In this master thesis we restrict our attention to proof assistants and proof checkers. We use [Geu1] to describe the work domain of proof checkers and a proof assistants.

2.2 Proofs

In mathematics, a proof is absolute. A mathematical proof can be reduced to a series of very small steps each of which can be verified simply and irrefutably. These steps are so small that no mathematician would ever do this. Traditional mathematical proofs are written in a way to make them easily understood by mathematicians. Logical routine steps are omitted and arguments can rely on intuitive arguments. Mathematicians know the routine steps and are able to check the intuitive arguments. A formal proof is a proof in which each step is checked all the way back to the begin axioms of mathematics. All intermediate steps must be included and arguments cannot rely on intuition. A formal proof is therefore on average four times bigger (see [Mol]) then the original proof.

The translation of an informal mathematical proof in natural language to a formal language is not so easy. One could think that in mathematics, this is not a big problem, because mathematicians already write their results in formulas and use a quite restrictive technical jargon in their proof. However there is still a considerable gap between the proofs that mathematicians write in books and articles and the proofs that can be understood by a computer.

2.2.1 Roles of proof

To describe the domain of proof checkers and proof assistants a but more, we look into the role of a proof in mathematics itself. A proof has two roles:

- A proof *convinces* the reader that the statement is correct.
- A proof *explains* why the statement is correct.

The first point consists of the administrative activities of verifying the correctness of the small reasoning steps to see if the proof is correct. One does not have to look at the broad picture, but one just has to verify step by step whether every step is correct. The second role deals with giving the intuition of the theorem.

In a proof that we find in an article or book, both roles are usually interwoven. A proof is usually constructed in three stages, namely:

- *Proof finding.* In this phase one tries to create the proof by experimenting, guessing, etc. This phase is usually not recorded but for students to learn mathematics it is indispensable to practice.
- *Proof recording.* In this phase the proof is written down. The report contains an explanation of why the statement holds and the proof steps needed to verify the statement step-by-step.
- *Proof presentation.* After a proof has been found, it goes into a phase of being communicated to others. This happens both before and after it has been written up. In the communication, one mainly tries to explain why the statement holds.

Now we have defined the roles of a proof and the stages it is constructed, we will look at the possible roles a proof assistant and/or a proof checker can play.

It is obvious that the proof checker is very helpful to verify if all the small reasoning steps that eventually result in the final proof are correct. When using the proof checker we must only translate the proof to the formal language. If we look at the explaining role of the proof, we see that the proof assistants do not have much to offer. At best they force a user to consider every small detail, which sometimes helps to see the implicit assumptions or dependencies that were lost in the high level mathematical proof. But most of the time the required amount of detail is considered to be a hindrance to a proper high level explanation of the proof.

If we look at the three stages of constructing the proof, we can see that proof checkers and proof assistants have little to offer in proof finding. For the average user it is impossible to formalize a proof that is not spelled out on paper yet. However a proof checker and a proof assistant can be very helpful in defining the scope of the proof. In this way one can find a possible proving method easier. When recording a proof, proof assistances are very useful. They are verifiable pieces of computer code that can be inspected at every level of detail. But the key idea of the proof is not recorded separately, since it is hidden in the details. Therefore it is hard to present a formalized proof to others. There are systems that have developed tools to represent proofs, but they are already too detailed for an average user.

2.3 Correctness proof checkers

A proof assistant is build by two components, a *user interface* and a *kernel* component. The user interface implements the communication with the user. It interactively creates the proofs without verifying the correctness of the proof. For this the kernel component is added. The kernel is the most important component of the proof assistant. It is implemented by a small amount of computer code. For example the kernel of the HOL light program is implemented in 500 lines of code. The kernel may be small but verifies all the proofs that are created in the proof assistant. Errors in the kernel may lead to an inconsistent system. This means that one can prove certain proofs that are actually incorrect. In this way the user is able to create a library full of errors, because also the correctness for the theorems that use the incorrect theorems cannot be guaranteed.

The first role of a proof was to convince the reader that the statement was correct. We can use a proof checker to verify if all the small reasoning steps in the proof are correct. But computer software is in principle not reliable, so why would we believe that such a program can verify proofs without making errors?

1. The first way is based on intuition. A programmer creates 1.5 bugs per line code while typing. Most bugs are typing errors that are spotted at once. About one bug per hundred lines of computer code is not found, before a program is released. The most reliable software, contains less than one bug per 10.000 lines of computer code. The various proof assistants differ in their reliability. But since the kernel of the leading proof assistants is very small and various mathematicians have checked the computer code, we believe that the kernel of these proof assistants are very reliable. This decreases the probability that the kernel contains one or more errors to almost zero.

2. If we have a system independent description of the logic and its mathematical features

(like the mechanisms for defing functions and data types), we can establish whether we believe in those, whether our definitions faithfully represents what we want to express and whether the proof steps make sense.

3. The kernel component itself is just a computer program, so its correctness can be verified. To do this, one first has to specify the properties of the program, which means that one has to formalize the rules of the logic. Then one has to prove that the proof assistant can prove a theorem α if and only if α is derivable in the logic.

4. In the past years programs are created that translate a proof from one proof language into another proof language (see [Har]). Now it is easy to check a proof by multiple proof assistants. The chance that one proof assistant has an error is small, but the chance that two or more proof assistants have the same error is even smaller, since each proof assistant is designed and implemented independently.

   But of course, one could ask if the translation programs translate the proof precise and correct. Since these are also computer programs, we can verify them as described in point 3.

5. Some proof assistants create an independently checkable proof object, while the user is interactively proving a theorem. These proof objects can be checked by hand or by a simple computer program.

2.4 Existing proof assistants

The history of computer based proof assistants starts in 1954. M. Davis proves that the sum of two even numbers is even, by using the Presburger algorithm. Only this program was not able to verify general mathematical proofs. In 1968, N.G. de Bruijn designs the first computer program that is able to check the validity of general mathematical proofs. This program was called Automath ([Bru]). Automath can be seen as the founder of the current used proof assistants. The leading programs these days are: Coq, Isabelle, HOL Light, Mizar and ProofPower.

The different proof assistants build in their own proof language a library of proven mathematical theorems. The size of the library indicates the expressiveness of the computer program. Some very impressive formalizations of mathematical theorems have been done and the volume of formalized mathematics is constantly increasing.

In [Wie] an overview of the 100 well-known theorems in mathematics is given. The overview shows which theorems are already formalized in which systems.

Chapter 3

λ-calculus λC and $\lambda D+$

As mentioned in Chapter 2.1 each proof checker has its own proof language. In this master thesis two proof checkers are built based on λ-calculus. There are various types of λ-calculi, each with a somewhat different notation and expressive power. The first proof checker is based on λC, whereas the second proof checker uses the λ-calculus λD^+. This λ-calculus is an extension of λC.

But before we start to describe the syntax and semantics of λC and λD^+, we first give a short description of the simplest λ-calculus, called 'untyped λ-calculus'. This calculus is the basis of λC and λD^+. Because of its simplicity, the language is very useful to describe some important notions. For a complete description of the untyped λ-calculus we refer to [Bar1].

In section 3.2 a description of λC is given and in section 3.3 λD^+ is described.

3.1 Untyped λ calculus

Functions are often described as expressions, e.g. x^2+1. These expressions tell us how, given an input value for x, one may calculate an output value. In this example one must take the square of x and than add one to the result. In the example x is used as an arbitrary value. In a concrete case, for example 3, one must replace x in the expression by 3. This will lead to 3^2+1, which results in 10.

To underline the function aspect of such an expression, the letter λ is introduced. One puts λx in front of the expression, followed by a dot as separation mark. Now the above expression x^2+1 is explicitly distinguished from the function $\lambda x\,.\,x^2+1$. The function maps the abstract variable x to the expression x^2+1. In ordinary mathematical notation one sometimes writes $x \to x^2+1$ for the function $\lambda x.x^2+1$. If we look at the concrete value, one can give this value as an argument to the function. The argument is written behind the expression. The example from above would be denoted as: $(\lambda x\,.\,x^2+1)3$, which is again an expression. To evaluate the expression, one may replace all the occurrences of x in the expression x^2+1 by 3.

From this example we can formulate two construction principles, namely:

- From an expression M and a variable x, one may construct a new expression $\lambda x.M$.
- From expressions M and N, one may construct a new expression MN.

In the next paragraphs we will not consider how to calculate a function in the real world. But we look at the behavior of functions in the simplest and most abstract way.

3.1.1 Untyped λ-terms

Expressions in λ-calculus are called λ-terms. In the previous subsection we gave an informal introduction of the construction principles concerning the set of λ-terms of the untyped λ-calculus. The formal definition is given below. In the definition we assume that there is an infinite set V of variables.

Definition 3.1.1 *The set Λ of all λ-terms.*
(1) (Variable:) If $x \in V$, then $x \in \Lambda$.
(2) (Application:) If M and $N \in \Lambda$, then $(MN) \in \Lambda$.
(3) (Abstraction:) If $x \in V$ and $M \in \Lambda$, then $(\lambda x\,.\,M) \in \Lambda$.

Notation 3.1.2 *(1) We use small letters and variants with subscripts and primes to denote variables in V.*
(2) To denote elements of Λ we use capital letters.

Note that, as the name of this λ-calculus states that the terms do not have a type. It is not specified for example for the λ-term $\lambda x.x^2$, what the type of the variable x is and what the type of the function $\lambda x.x^2$ is.

Variable occurrences in a λ-term can be divided into three categories: *free* occurences, *bound* occurrences and *binding* occurrences. If we have the term $\lambda x.M$ all occurrences of x in M are bound by the binding variable x between the λ symbol and the separation dot. A free variable is a variable that is not bound by a λ-declaration. We can define the free variables formally.

Definition 3.1.3 *FV, the set of free variables of a λ-term*
(1) (Variable:) FV(x) = $\{x\}$.
(2) (Application:) FV(MN) = FV(M) ∪ FV(N).
(3) (Abstraction:) FV(λx.M) = FV(M) \ $\{x\}$

In λ-calculus one is able to substitute a variable x by λ-term N. Substitution is defined by:

Definition 3.1.4 *Substitution*
(1a) $x[x := N] \equiv N$.
(1b) $y[y := N] \equiv y$ if $x \not\equiv y$.
(2) $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$.
(3) $(\lambda y.P)[x := N] \equiv \lambda y.(P[x := N])$ if $x \not\equiv y$

3.1.2 α-conversion

Functions in the λ-notation have the property that the name of the binding variable is not essential. The 'square function' for example can be expressed as well by $\lambda x.x^2$ as by $\lambda u.u^2$. In both cases the formula means: 'the function which calculates the square of an input value and gives the obtained number as its output value'. Therefore the two functions are equivalent. If we say in λ-calculus that two terms are equal we actually mean that the two terms are α-equivalent.

Definition 3.1.5 *α-conversion or α-equivalence*
(1) (Renaming:) $\lambda x.M =_\alpha \lambda y.M[x := y]$ *if* $y \notin FV(M)$.
(2) (Compatibility:) If $M =_\alpha N$, *then* $ML =_\alpha NL$, $LM =_\alpha LN$ *and* $\lambda x.M =_\alpha \lambda x.N$.
(3a) (Reflexivity:) $M =_\alpha M$.
(3b) (Symmetry:) If $M =_\alpha N$ *then* $N =_\alpha M$.
(3c) (Transitivity:) If ($L =_\alpha M$ *and* $M =_\alpha N$), *then* $L =_\alpha N$

Convention 3.1.6 *From now on, we identity α-convertible λ-terms.*
This is to say, we consider λ-terms modulo α-equivalence.

Since Convention 3.1.6 permits us to choose the names of binding and bound variables at will, it is convenient to agree on the following, which is called the Barendregt-convention.

Convention 3.1.7 *Barendregt-convention*
We choose the names of the binding (and corresponding bound) variables in a λ-term in such a manner, that all binding variable (behind the λ's) are mutually different, and such that each of them differs from all free variables occurring in the term.

Example 3.1.8 *By using the Barendregt-convention to name the variables in a λ-term it is easier to see the binding relation between variables. Instead of writing the terms* $\lambda x.\lambda x.x$ *and* $x(\lambda x.x)$ *we write* $\lambda x.\lambda y.y$ *and* $x(\lambda y.y)$.

3.1.3 β-reduction

Besides α-equivalence, untyped λ-calculus contains a second equality, called β-equivalence. Before we can give the definition of β-equivalence we first define a β-reduction step.

Definition 3.1.9 *One-step β-reduction,* $\rightarrow_\beta$
(1) (Basis:) $(\lambda x.M)N \rightarrow_\beta M[x := N]$.
(2) (Compatibility:) If $M \rightarrow_\beta N$, *then* $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LN$ *and* $\lambda x.M \rightarrow_\beta \lambda x.N$

A subterm of the form $(\lambda x.M)N$ is called a *redex*, while the subterm $M[x := N]$ is called the *contractum* of this redex.

Definition 3.1.10 *β-reduction (zero-or-more-steps),* $\twoheadrightarrow_\beta$
$M \twoheadrightarrow_\beta N$ *if there is an* n *and there are terms* M_0 *to* M_n *such that* $M_0 \equiv M$, $M_n \equiv N$ *and for all* i *such that* $0 \leq i < n$:

$$M_i \rightarrow_\beta M_{i+1} \ .$$

With the definition of β-reduction we are able to define β-equivalence.

Definition 3.1.11 *β-equivalence or β-conversion,* $=_\beta$
$M =_\beta N$ *if there is an* n *and there are terms* M_0 *to* M_n *such that* $M_0 \equiv M, M_n \equiv N$ *and for all* i *such that* $0 \leq i < n$:

$$\textit{either } M_i \rightarrow_\beta M_{i+1} \textit{ or } M_{i+1} \rightarrow_\beta M_i \ .$$

We have seen β-reduction in the introduction of the untyped λ-calculus. There we used β-reduction to reduce the term $(\lambda x\,.\,x^2+1)3$ to 3^2+1. We can say that β-reduction mimics a calculation: by applying a function to an argument, using reduction, we get a kind of (temporary) outcome, which hopefully is closer to some final outcome. In the final outcome no new β-reduction step is possible. Such a term is called a β-normal form.

Definition 3.1.12 *β-normal form, β-nf.*
(1) M is in β-normal form if M does not contain any redex.
(2) M has a β-normal form if there is a N in β-normal form such that $M =_\beta N$.

To reduce a term M to its β-normal form we apply a 'reduction-path' of β-reduction steps till M contains no β-redex anymore.

Definition 3.1.13 *Reduction path*
A finite reduction path from M is finite sequence of terms N_0, N_1, N_2,...,N_n such that $N_0 \equiv M$ and $N_i \to_\beta N_{i+1}$ for each i with $0 \leq i < n$.
An infinite reduction path from M is infinite sequence of terms N_0, N_1, N_2,... with $N_0 \equiv M$ and $N_i \to_\beta N_{i+1}$ for all $i \in \mathbb{N}$.

Definition 3.1.14 *Strong normalizing*
M is strongly normalizing if all reduction paths from M are finite.

In the untyped λ-calculus not all terms are strongly normalizing. Take for example the term $(\lambda x.xx)(\lambda y.yy)$. After each β-reduction step a new β-redex is introduced. We will see in λC and λD^+ that all terms are strongly normalized. This is very important since we do not want infinite loops in our system.

3.2 λC

We can extend the untyped λ-calculus with types ([Bar2]). The most simple typed λ-calculus is called $\lambda\to$. Each λ-calculus in the so called λ-cube or Barendregt cube is an extension of $\lambda\to$. The λ-cube is shown in figure 3.1.

For a description of $\lambda\to$, $\lambda 2$, $\lambda\underline{\omega}$ and λP we refer to [Bar2] or [Ned]. This master thesis only contains a description of the complete system called λC. The λ-calculus λC is a combination of $\lambda\to$, $\lambda 2$, $\lambda\underline{\omega}$ and λP. The complete system is also called the *Calculus of Constructions* or λ-Coquand, after its founder Thierry Coqand. Therefore the letter 'C' in the name has multiple references.

3.2.1 Judgements

In λC we define a judgement. To construct a λC judgement we use the derivation rules given in section 3.2.2. In a λC judgement we can distinguish the following components.

Definition 3.2.1 *Statement, declaration, context, judgement*

1. *A statement is of the form $M : N$. In such a statement, M is called the subject and N the type.*
2. *A declaration is a statement with a variable as subject.*

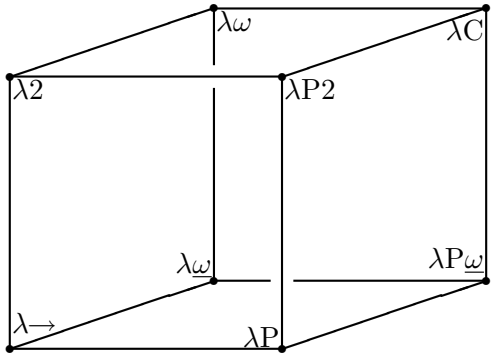

Figure 3.1: The λ-cube or Barendregt cube

3. *A context is a list of declarations with different subjects.*

4. *A judgement has the form* $\Gamma \vdash M : N$*, with* Γ *a context and* $M : N$ *a statement.*

The domain of the context Γ is the set of variables that are declared in Γ.

Definition 3.2.2 *domain (dom).*
If $\Gamma \equiv x_1 : N_1, \ldots, x_n : N_n$ *then the domain of* Γ *is the set* $\{x_1, \ldots, x_n\}$.

3.2.2 Derivation rules

The λC-calculus contains seven derivation rules. With the derivation rules we can construct all λC judgements. The derivation rules are given in figure 3.2. We shall describe the derivation rules by an example. The example is given in section 3.2.3.

Name	Derivation rule
(ax)	$\emptyset \vdash * : \Box$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$ if $x \notin \Gamma$
(weak)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$ if $x \notin \Gamma$
(form)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$
(appl)	$\dfrac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$
(abst)	$\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$
(conv)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$ if $B =_\beta B'$

Figure 3.2: Derivation rules of λC.

3.2.3 λC-example

For the description of the λC derivation rule we shall give a small example. In [Ned] the logical operators are translated to λC-terms. In the example we use the logical operator $\forall$ and $\Rightarrow$. The logical operators are defined by:

- $\forall_{x \in S} Px$ is translated to $\Pi x : S.Px$, where $S : *$ and $P : \Pi y : S.*$.
- $A \Rightarrow B$ is translated to $\Pi x : A.B$, where $A : *$ and $B : *$.

In the example we prove that if $\forall_{x \in S} Px$ holds and we have an element, say u in S, then also Pu holds. We can translate this proof to λC in judgement J:

$$S : *, u : S, P : \Pi v : S.* \quad \vdash \quad \lambda w : (\Pi x : S.Px).wu \quad : \quad \Pi y : (\Pi z : S.Pz).Pu$$

Note that 'a given element u of S' is hidden in the context declaration $u : S$ and '$(\forall_{x \in S} Px) \Rightarrow Pu$' is hidden in the type of the statement. In the example we will try to derive judgement J by using the derivation rules of λC. The derivation is shown in list notation.

The beginning of every λC judgement starts with the *ax* derivation rule, since this is the only axiom in λC. All other derivation rules need judgements as premisses before we can use them. So the first line in the derivation is always equal to:

(1) $\emptyset \vdash * : \Box$ (ax)

With this judgement we will try to compute more complex judgements that will eventually result in the final judgement J. The conclusion of the *ax* derivation rule can be used as a premiss for the *start* and *weak* derivation rule. Both derivation rules are intended to introduce variables in the judgement. Judgement J contains the variables S, P, u, v, w x, y and z. All those variables are introduced by the *start* and *weak* derivation rules.

To execute the *start* derivation rule we must have a judgement of the form $\Gamma \vdash A : s$ that is already derived. This premiss is added to ensure that the new introduced variables has a derivable type. To add the variable S of type $*$ in the example we must know that $*$ is a 'valid' type. This is already proven in line (1). Therefore we can add the declaration $S : *$ to the context by the *start* derivation rule. By using the *start* derivation rule we also know that S is derivable and we can add the declaration of u to the context.

(2) $S : * \vdash S : *$ (start on (1))

(3) $S : *, u : S \vdash u : S$ (start on (2))

In the result of the *start* derivation rule, the statement is always equal to the declaration that is added to the context. Therefore we are not able to construct for example the judgement $S : * \vdash * : \Box$. To regain this possibility λC contain the *weak* derivation rule. The *weak* derivation rule has two premisses. The premiss $\Gamma \vdash C : s$ corresponds to the premiss of the *start* rule and checks the validity of the type. And the second premiss contains the statement of the conclusion of the *weak* derivation rule.

Note that when using the *weak* derivation rule to construct the judgement $S : * \vdash * : \Box$, line (1) is used for both premisses.

(4) $S : * \vdash * : \Box$ (*weak* on (1) and (1))

We need the *start* and the *weak* derivation rules yet some more to introduce more variables in the judgement. We will not describe the steps, since they are identical to the first four lines.

(5) $S : *, u : S \vdash S : *$ (*weak* on (2) and (2))

(6) $S : *, u : S \vdash * : \Box$ (*weak* on (2) and (4))

(7) $S : *, u : S, v : S \vdash * : \Box$ (*weak* on (5) and (6))

Note that by only using the *start* and *weak* derivation rule we can only introduce variables with a simple type. It is not possible to add a variable with a function type. Therefore we are not able to add the variable P to the context of judgement J with the *start* and *weak* derivation rules alone. We can create a function type $\Pi p : A.B$ with the *form* derivation rule.

The first premiss of the *form* derivation rule guarentees that A is a derivable type in λC, while the second premiss guarantees this for B. Note that x can occur in B and therefore x is added to the context of the second premiss.

With the *form* derivation rule we can derive the type of P and introduce P by the *start* and *weak* derivation rule to the judgement, since we know that the function type is derivable.

(8) $S : *, u : S \vdash \Pi v : S.* : \Box$ (*form* on (5) and (7))

(9) $S : *, u : S, P : \Pi v : S.* \vdash S : *$ (*weak* on (5) and (8))

(10) $S : *, u : S, P : \Pi v : S.* \vdash P : \Pi v : S.*$ (*start* on (8))

We now give some less important steps in the derivation, without comment. All the judgement can be created by the derivation rules that are already described.

(11) $S : *, u : S, P : \Pi v : S.*, x : S \vdash P : \Pi v : S.*$ (*weak* on (9) and (10))

(12) $S : *, u : S, P : \Pi v : S.*, x : S \vdash x : S$ (*start* on (9))

In section 3.1 we saw that it is possible to add an argument to a function, by the application constructor. In λC the *appl* derivation rule is added for this. To use the *appl* derivation rule, we must have a derivable function, say M. A function in λC has always a type of the form $\Pi x : A.B$. To add an argument N to the function M, the type of N must be equal to A. If this is not the case then N cannot be used as an argument for M.

Since the input of the function is known, the type of MN is equal to the type of the output of the function M. In this example this will be B.

In the previous steps we constructed a function called P, with type $\Pi v : S.*$. It is possible with the *appl* derivation to give an argument to the function, if the argument has type S.

(13) $S : *, u : S, P : \Pi v : S.*, x : S \vdash P\ x : *$ (*appl* on (11) and (12))

Yet again we list some more steps in the derivation. All the judgements can be constructed by using the derivation rules *start*, *weak*, *form* and *appl*.

(14) $S : *, u : S, P : \Pi v : S.* \vdash \Pi x : S.Px : *$ (*form* on (9) and (13))

(15) $S : *, u : S, P : \Pi v : S.*, w : \Pi x : S.Px \vdash w : \Pi x : S.Px$ (*start* on (14))

(16) $S : *, u : S, P : \Pi v : S.* \vdash u : S$ (*weak* on (3) and (8))

(17) $S : *, u : S, P : \Pi v : S.*, w : \Pi x : S.Px \vdash u : S$ (*weak* on (14) and (16))

(18) $S : *, u : S, P : \Pi v : S.*, w : \Pi x : S.Px \vdash wu : Pu$ (*appl* on (15) and (17))

In section 3.1 we have described that equality in λ-calculus means modulo α-equivalence. In the previous steps we never used α-conversion, but in the lines (19), (20) and (21) we use it.

(19) $S : *, u : S, P : \Pi v : S.*, y : \Pi z : S.Pz \vdash P : \Pi v : S.*$ (*weak* on (10) and (14))

(20) $S : *, u : S, P : \Pi v : S.*, y : \Pi z : S.Pz \vdash Pu : *$ (*appl* on (17) and (19))

(21) $S : *, u : S, P : \Pi v : S.* \vdash \Pi y : (\Pi z : S.Pz).Pu : *$ (*form* on (14) and (20))

Besides the application, a second constructor was added in the untyped λ-calculus. With the second constructor one was able to construct a function of the form $\lambda x.M$. Since λC is typed, we must assign types to the function. A function in λC has the structure $\lambda x : A.M$ with the type equal to $\Pi x : A.B$, where B is equal to the type of M. To create such a function, the *abst* derivation rule is added to λC.

The *abst* derivation rule contains two premisses. The premiss $\Gamma, x : A \vdash M : B$ makes sure that the new function is derivable, while the premiss $\Gamma \vdash \Pi x : A.B : s$ is added to check if the type of the new function is 'correct'.

By using the *abst* derivation rule we can complete the derivation of the judgement J.

(22) $S : *, u : S, P : \Pi v : S.* \vdash \lambda w : (\Pi x : S.Px).wu :$ (*abst* on (18) and (21))
$\Pi y : (\Pi z : S.Pz).Pu$

The only rule that we did not describe in the example is the *conv* derivation rule. The reason that we choose an example where the *conv* derivation rule was not used is because we want a small and simple example which can easily be extended to λD^+ (see section 3.3.3).

With the *conv* derivation rule we are able to replace the type of the judgement by a new type. For the new type we cannot use all terms. The new type must be derivable with the λC derivation rules and should β-equivalent to the existing type of the judgement.

We can replace the type of judgement J by the term
$(\lambda Q : (\Pi a : S.*).\Pi y : (\Pi z : S.Qz).Qu)P$. We shall not give the derivation of the new term but note that both terms are β-equivalent.

3.2.4 Properties of λC

In this section we will give some properties of λC. The properties are necessary for the rest of the master thesis. We will not give the proofs for the properties. For the proofs we refer to [Ned].

Definition 3.2.3 *Legal λC-term*
A term M is λC is called legal if there exist a context Γ and a type N such that

$$\Gamma \vdash M : N \ .$$

Definition 3.2.4 *Free Variable Lemma*
If $\Gamma \vdash A : B$, then $FV(A) \cup FV(B) \subseteq dom(\Gamma)$.

Lemma 3.2.5 *Thinning Lemma, Condensing Lemma*
(1) (Thinning) Let Γ' and Γ'' be contexts such that $\Gamma' \subseteq \Gamma''$. Now if $\Gamma' \vdash A : B$ then also $\Gamma'' \vdash A : B$.
(2) (Condensing) If $\Gamma', x : A, \Gamma'' \vdash B : C$ and x does not occur in Γ'', B or C, then also $\Gamma, \Gamma'' \vdash B : C$.

We have to extend the α-conversion definition, since in λC we are able to use α conversion for Π-terms.

Definition 3.2.6 *α-conversion or α-equivalence*
(1a) (Renaming:) $\lambda x : A.M =_\alpha \lambda y : A.M[x := y]$ if $y \notin FV(M)$.
(1a) (Renaming:) $\Pi x : A.M =_\alpha \Pi y : A.M[x := y]$ if $y \notin FV(M)$.
(2) (Compatibility:) If $M =_\alpha N$, then $ML =_\alpha NL$, $LM =_\alpha LN$, $\lambda x : M.A =_\alpha \lambda x : N.A$, $\lambda x : A.M =_\alpha \lambda x : A.N$, $\Pi x : M.A =_\alpha \Pi x : N.A$ and $\Pi x : A.M =_\alpha \Pi x : A.N$.
Reflexivity, Symmetry and Transitivity still hold.

We can still apply the definitions for zero-or-more β-reduction steps, β-equivalence, β-normal form, Reduction-path and Strong Normalizing still hold for λC. To use the definition of one-step β-reduction, we must introduce types and update the compatibility component.

For the one-step β-reduction no major adjustments are necessary, we only have to add a type to the β-redex.

Definition 3.2.7 *One-step β-reduction, $\rightarrow_\beta$*
(1) (Basis:) $(\lambda x : A.M)N \rightarrow_\beta M[x := N]$.
(2) (Compatibility:) If $M =_\beta N$, then $ML =_\beta NL$, $LM =_\beta LN$, $\lambda x : M.A =_\beta \lambda x : N.A$, $\lambda x : A.M =_\beta \lambda x : A.N$, $\Pi x : M.A =_\beta \Pi x : N.A$ and $\Pi x : A.M =_\beta \Pi x : A.N$.

The following additional properties hold for λC concerning β-equivalence.

Lemma 3.2.8 *Uniqueness of types up to β-conversion.*
If $\Gamma \vdash A : B_1$ and $\Gamma \vdash A : B_2$, then $B_1 =_\beta B_2$

Theorem 3.2.9 *Church-Rosser; CR; Confluence*
Suppose that for a given λ-term M holds that $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$. Then there is a λ-term N_3 such that $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$.

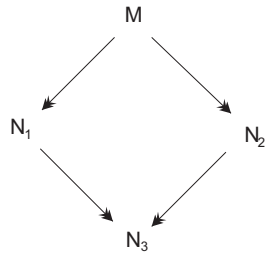

Figure 3.3: A pictorial representation of the Church-Rosser Theorem.

Theorem 3.2.10 *Strong Normalizing Theorem or Termination Theorem*
Every legal M is strongly normalizing.

Since the Strong Normalizing Theorem holds for λC we know that no infinite β-reduction paths are possible when trying to retrieve the β-normal form of a given term.

3.3 λD^+

3.3.1 Axioms and definitions

The λ-calculus λD^+ is an extension of λC. In λD^+ it is possible to define axioms and definitions. For this an extra component is added to the judgement, called the *environment*. To define the structure of an axiom and a definition, we will first give some notations.

Notation 3.3.1 *(1) We write $\overline{x}$ for the list $x_1, \ldots, x_n$ of variables.*
(2) We write $\overline{A}$ for the list $A_1, \ldots, A_n$ of expressions.
(3) We write $\overline{x} : \overline{A}$ for the list of statements (or the context) $x_1 : A_1, \ldots, x_n : A_n$.

We can define an axiom and a definition in λD^+ as follows.

Definition 3.3.2 *Axioms and Definitions in λD^+.*
(1) An axiom in λD^+ is an expression $\overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N$. In the axiom $\overline{x} : \overline{A}$ is called the context, a the constant, $\overline{x}$ the parameter list and $\perp\!\!\!\perp : N$ the statement.
(2) A definition in λD^+ is an expression $\overline{x} : \overline{A} \rhd a(\overline{x}) := M : N$. In the definition $\overline{x} : \overline{A}$ is called the context, a the constant, $\overline{x}$ the parameter list and $M : N$ the statement.
(3) An environment Δ is a finite list of axioms and/or definitions.
(4) An ('extended') judgement has the form $\Delta; \Gamma \vdash M : N$, with Δ an environment, Γ a context and $M : N$ a statement.

Constants defined in the environment can be used in the context and the statement of the judgement. A constant a defined by an axiom or definition D can also be used in another definition D'. To use a in D', the axiom or definition D must be defined earlier in the environment than D'. For using constants in other axioms or definitions we can define the following notions.

Definition 3.3.3 *Direct dependency*
Let D_i be an axiom or definition, where
$D_i \equiv \overline{x} : \overline{A} \rhd a(\overline{x}) := M : N$ *or* $D_i \equiv \overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N$
(1) Let D_j be an axiom, where $D_j \equiv \overline{y} : \overline{B} \rhd b(\overline{y}) := \perp\!\!\!\perp : V$, with $a \not\equiv b$.
Then D_j directly depends on D_i if the constant a occurs (one or more times) in D_j (so a occurs in $\overline{B}$ and/or in V).
(2) Let D_k be an definition, where $D_k \equiv \overline{z} : \overline{C} \rhd c(\overline{z}) := W : X$, with $a \not\equiv c$.
Then D_k directly depends on D_i if the constant a occurs (one or more times) in D_k (so a occurs in $\overline{C}$ and/or in W and/or in X).

Definition 3.3.4 *Dependency*
The notion 'dependency' between axioms/definitions, denoted by the symbol '$<$', is the irreflexive ordering generated by the notion 'direct dependency'. Otherwise said:

$D' < D''$ if there are $n > 0$ and $D_0, \ldots, D_n$ such that $D' \equiv D_0$, $D'' \equiv D_n$ and for $0 \leq i < n$, each D_{i+1} directly depends on D_i

3.3.2 Derivation rules

Since λD^+ is an extension of λC all derivation rules of λC are also valid for λD^+. We must only adjust the λC derivation rules, since judgements λD^+ contain an extra component, called the environment. Before we list the derivation rules of λD^+, we give two more extra notations used in the derivation rules.

Notation 3.3.5 *(1) We write $[\overline{x} := \overline{U}]$ as an abbreviation for the simultaneous substitution $[x_1 := U_1, \ldots, x_n := U_n]$.*
(2) We write $\Delta; \Gamma \vdash \overline{U} : \overline{V}$ for the list of extended judgements.

$$\begin{array}{l} \Delta; \Gamma \vdash U_1 : V_1, \\ \vdots \\ \Delta; \Gamma \vdash U_n : V_n \end{array}$$

The derivation rules of λD^+ are listed in figure 3.4. We shall explain the λD^+ derivation rules by an example just like with we did with the derivation rules of λC. The example in given in section 3.3.3.

Name	Derivation rule
(ax)	$\emptyset;\emptyset \vdash * : \Box$
(start)	$\dfrac{\Delta;\Gamma \vdash A : s}{\Delta;\Gamma, x : A \vdash x : A}$ if $x \notin \Gamma$
(weak)	$\dfrac{\Delta;\Gamma \vdash A : B \qquad \Delta;\Gamma \vdash C : s}{\Delta;\Gamma, x : C \vdash A : B}$ if $x \notin \Gamma$
(form)	$\dfrac{\Delta;\Gamma \vdash A : s_1 \qquad \Delta;\Gamma, x : A \vdash B : s_2}{\Delta;\Gamma \vdash \Pi x : A.B : s_2}$
(appl)	$\dfrac{\Delta;\Gamma \vdash M : \Pi x : A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B[x := N]}$
(abst)	$\dfrac{\Delta;\Gamma, x : A \vdash M : B \qquad \Delta;\Gamma \vdash \Pi x : A.B : s}{\Delta;\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$
(conv)	$\dfrac{\Delta;\Gamma \vdash A : B \qquad \Delta;\Gamma \vdash B' : s}{\Delta;\Gamma \vdash A : B'}$ if $B =_{\beta\delta} B'$
(env-weak)	$\dfrac{\Delta;\Gamma \vdash K : L \qquad \Delta;\overline{x} : \overline{A} \vdash M : N}{\Delta;\overline{x} : \overline{A} \rhd a(\overline{x}) := M : N;\Gamma \vdash K : L}$ if $a \notin \Delta$
(env-weak$^{\perp\!\!\!\perp}$)	$\dfrac{\Delta;\Gamma \vdash K : L \qquad \Delta;\overline{x} : \overline{A} \vdash N : s}{\Delta;\overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N;\Gamma \vdash K : L}$ if $a \notin \Delta$
(inst)	$\dfrac{\Delta;\Gamma \vdash * : \Box \qquad \Delta;\Gamma \vdash \overline{U} : \overline{A[\overline{x} := \overline{U}]}}{\Delta;\Gamma \vdash a(\overline{U}) : N[\overline{x} := \overline{U}]}$ if $\overline{x} : \overline{A} \rhd a(\overline{x}) := M : N \in \Delta$
(inst$^{\perp\!\!\!\perp}$)	$\dfrac{\Delta;\Gamma \vdash * : \Box \qquad \Delta;\Gamma \vdash \overline{U} : \overline{A[\overline{x} := \overline{U}]}}{\Delta;\Gamma \vdash a(\overline{U}) : N[\overline{x} := \overline{U}]}$ if $\overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N \in \Delta$

Figure 3.4: Derivation rules λD^+.

3.3.3 Example λD^+

In section 3.2.3 we gave an example for λC. In the λC example we proved that: 'if $\forall_{x \in S} Px$ holds and we a have an element $u \in S$ then also Pu holds.' In the example of λC we translate the proof to a λC judgement by using the translations of the definitions $\forall$ and $\Rightarrow$ given in [Ned].

But in λD^+ we are able to add definitions to the environment of a judgement. So in the λD^+ example we will add the definition of $\forall$ and $\Rightarrow$ to the environment and use the constants in the statement of the judgement. This results is:

$$D1, D2;\ S : *, u : S, P : \Pi v : S.* \ \vdash\ \lambda w : (\Pi x : S.Px).wu \ :\ impl(forall(S, P), Pu)$$

where $D1 \equiv A : *, Q : \Pi a : A. * \rhd forall(A, Q) := \Pi b : A.Qb : *$
and $D2 \equiv A : *, B * \rhd impl(A, B) := \Pi a : A.B : *$

Also in λD^+ we must start with the *ax* derivation rule, since it is the only rule without premisses.

(1) $\emptyset; \emptyset \vdash * : \Box$ (ax)

The first thing that we have to do is to make sure that the definitions $forall$ and *impl* are added to the environment. To add a definition to the environment, λD^+ contains the *env-weak* derivation rule. The *env-weak*$^{\perp\!\!\!\perp}$ derivation rule is used to add an axiom to the environment.

To add a new definition D of the form $\overline{x} : \overline{A} \rhd a(\overline{x}) := M : N$ to the environment we must prove that D is derivable with the derivation rules of λD^+. To prove this it is sufficient to prove that $M : N$ is derivable, with respect to $\overline{x} : \overline{A}$ and the environment Δ, since D can have a dependency relation with another axiom or definition of Δ. The last restriction is that Δ cannot already contain an axiom or a definition for a constant with name a. When the restrictions are satisfied we can add the definition.

So when we want to add the definition of $forall$ to the environment of the judgement of line (1), we must proof that $A : *, Q : \Pi a : A.* \vdash \Pi b : A.Qb : *$ is derivable. For this we use the extended λC derivation rules in line (2) till (11). In line (12) the $forall$ definition is added to the environment by the *env-weak* derivation rule.

(2)	$\emptyset;\ A : * \vdash A : *$	(*start* on (1))
(3)	$\emptyset;\ A : * \vdash * : \Box$	(*weak* on (1) and (1))
(4)	$\emptyset;\ A : *, p : A \vdash * : \Box$	(*weak* on (2) and (3))
(5)	$\emptyset;\ A : * \vdash \Pi p : A.* : \Box$	(*form* on (2) and (4))
(6)	$\emptyset;\ A : *, Q : \Pi p : A.* \vdash A : *$	(*weak* on (2) and (5))
(7)	$\emptyset;\ A : *, Q : \Pi p : A.* \vdash Q : \Pi p : A.*$	(*start* on (5))
(8)	$\emptyset;\ A : *, Q : \Pi p : A.*, x : A \vdash Q : \Pi p : A.*$	(*weak* on (6) and (7))
(9)	$\emptyset;\ A : *, Q : \Pi p : A.*, x : A \vdash x : A$	(*start* on (6))
(10)	$\emptyset;\ A : *, Q : \Pi p : A.*, x : A \vdash Qx : *$	(*appl* on (8) and (9))
(11)	$\emptyset;\ A : *, Q : \Pi p : A.* \vdash \Pi x : A.Qx : *$	(*form* on (6) and (10))
(12)	$D1; \emptyset \vdash * : \Box$	(*env-weak on (1) and (11)*)

For the definition *impl* we can do the same. After we add both the definition of $forall$ and *impl*, we will reuse the derivation of the λC example. Only now the judgements contain an environment with the definitions of $forall$ and *impl*. We will give this derivation without a further explanation.

(13)	$D1;\ B : * \vdash B : *$	(*start* on (12))
(14)	$D1;\ B : * \vdash * : \Box$	(*weak* on (12) and (12))
(15)	$D1;\ B : *, C : * \vdash B : *$	(*weak* on (13) and (14))
(16)	$D1;\ B : *, C : * \vdash C : *$	(*start* on (14))
(17)	$D1;\ B : *, C : *, b : B \vdash C : *$	(*weak* on (15) and (16))
(18)	$D1;\ B : *, C : * \vdash \Pi b : B.C : *$	(*form* on (15) and (17))

(19)	$D1, D2; \emptyset \vdash * : \Box$	(*env-weak* on (12) and (18))
(20)	$D1, D2;\ S : * \vdash S : *$	(*start* on (19))
(21)	$D1, D2;\ S : *, u : S \vdash u : S$	(*start* on (20))
(22)	$D1, D2;\ S : * \vdash * : \Box$	(*weak* on (19) and (19))
(23)	$D1, D2;\ S : *, u : S \vdash S : *$	(*weak* on (20) and (20))
(24)	$D1, D2;\ S : *, u : S \vdash * : \Box$	(*weak* on (20) and (22))
(25)	$D1, D2;\ S : *, u : S, v : S \vdash * : \Box$	(*weak* on (23) and (24))
(26)	$D1, D2;\ S : *, u : S \vdash \Pi v : S.* : \Box$	(*form* on (23) and (25))
(27)	$D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash S : *$	(*weak* on (23) and (26))
(28)	$D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash P : \Pi v : S.*$	(*start* on (25))
(29)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, x : S \vdash P : \Pi v : S.*$	(*weak* on (27) and (28))
(30)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, x : S \vdash x : S$	(*start* on (27))
(31)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, x : S \vdash P\ x : *$	(*appl* on (29) and (30))
(32)	$D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash \Pi x : S.Px : *$	(*form* on (27) and (31))
(33)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, w : \Pi x : S.Px \vdash w : \Pi x : S.Px$	(*start* on (32))
(34)	$D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash u : S$	(*weak* on (21) and (26))
(35)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, w : \Pi x : S.Px \vdash u : S$	(*weak* on (32) and (34))
(36)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, w : \Pi x : S.Px \vdash wu : Pu$	(*appl* on (33) and (35))
(37)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, y : \Pi z : S.Pz \vdash P : \Pi v : S.*$	(*weak* on (28) and (32))
(38)	$D1, D2;\ S : *, u : S, P : \Pi v : S.*, y : \Pi z : S.Pz \vdash Pu : *$	(*appl* on (35) and (37))
(39)	$D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash \Pi y : (\Pi z : S.Pz).Pu : *$	(*form* on (32) and (38))
(40)	$D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash \lambda w : (\Pi x : S.Px).wu :$ $\Pi y : (\Pi z : S.Pz).Pu$	(*abst* on (36) and (39))

In the last part of the derivation we must replace the type of the judgement in line (40) by the term $impl(forall(S, P), Pu)$. The only derivation rule that is applicable for this is the *conv* derivation rule. The *conv* derivation rule in λD^+ is extended in comparison with the *conv* rule of λC. Now we cannot only replace the type by a derivable β-equivalent term but also by a derivable $\beta\delta$-equivalent term. The definition of $\beta\delta$-equivalence is given in section 3.3.4. For now we just assume that $impl(forall(S, P), Pu)$ is $\beta\delta$-equivalent to $\Pi y : (\Pi z : S.Pz).Pu$. To use the *conv* derivation we must first prove that there exist a derivable term $impl(forall(S, P), Pu)$. To create such a term, we must use the definitions of the environment. We shall start by trying to create the term $forall(S, P)$. The term

$forall(S, P)$ is an instantiated version of the definition $D1$.

When using a definition $\overline{x} : \overline{A} \rhd a(\overline{x}) := M : N$ from the a derivable environment Δ, to introduce the constant a in the statement of the judgement, we must prove that $\overline{U}$ is a correct instantiation for the variable list of the constant a. The instantiation is correct if all types in $\overline{U}$ are correct with respect to $\overline{x} : \overline{A}$. Because variables of $\overline{x}$ can occur in $\overline{A}$ and we instantiate the variables of $\overline{x}$ by $\overline{U}$, we must proof that $\overline{U} : \overline{A}[\overline{x} := \overline{U}]$ is derivable.

If we look at our current example, we must proof that $S : *$ and $P : \Pi a : A. * [A := S]$ are derivable. In the lines (27) and (28) we already did this. The last needed premisses for the *inst* derivation rule guarantees that the statement before the instantiation is derivable. This occurs in line (41).

(41) $D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash * : \Box$ (*weak* on (24) and (26))

(42) $D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash forall(S, P) : *$ (*inst* on (27), (28) and (41))

In the last steps of the derivation we instantiate the constant *impl* with $forall(S, P)$ as the first instantiation and Pu as the second. In the definition of *impl* we can see that both terms must have a type equal to $*$. Now we have proven that $impl(forall(S, P), Pu)$ is derivable, we can use the *conv* derivation rule and derive the desired judgement.

(43) $D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash Pu : *$ (*appl* on (28) and (34))

(44) $D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash impl(forall(S, P), Pu) : *$ (*inst* on (41), (42) and (43))

(45) $D1, D2;\ S : *, u : S, P : \Pi v : S.* \vdash \lambda w : (\Pi x : S.Px).wu :$ (*conv* on (40) and (44))
$impl(forall(S, P), Pu)$

In the example the *env-weak*$^{\perp\!\!\!\perp}$ and the *inst*$^{\perp\!\!\!\perp}$ derivation rules are not discussed. The rules are comparable to the *env-weak* and the *inst* derivation rules. Only the *env-weak*$^{\perp\!\!\!\perp}$ and *inst*$^{\perp\!\!\!\perp}$ are used for axioms instead of definitions. Because an axiom has the the structure $\overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N$, we only have to check for the *env-weak*$^{\perp\!\!\!\perp}$ derivation rule if the type is derivable instead of the complete statement.

3.3.4 Reduction

The λ-calculus λD^+ contains besides α-equivalence and β-equivalence two more equivalence relation, called δ-equivalence and $\beta\delta$-equivalence. Because α-equivalence and β-equivalence are also valid in λD^+, we can apply the following λC properties for λD^+: zero-or-more β -reduction steps, β-equivalence, β-normal form, Reduction path, Strong Normalizing, Uniqueness of Types up to β-conversion and Strong Normalizing Theorem. For the definitions of α-equivalence and one-step β-reduction we must adapt the compatibility element.

Definition 3.3.6 *α-conversion or α-equivalence*
(2) (Compatibility:) If $M =_\alpha N$, then $ML =_\alpha NL$, $LM =_\alpha LN$, $\lambda x : M.A =_\alpha \lambda x : N.A$, $\lambda x : A.M =_\alpha \lambda x : A.N$, $\Pi x : M.A =_\alpha \Pi x : N.A$, $\Pi x : A.M =_\alpha \Pi x : A.N$ and $a(\overline{A}, M, \overline{B}) =_\alpha a(\overline{A}, N, \overline{B})$.
Renaming, Reflexivity, Symmetry and Transitivity still hold.

Definition 3.3.7 *One-step β-reduction, $\rightarrow_\beta$*
(1) (Basis:) Equal to λC.
(2) (Compatibility:) If $M \rightarrow_\beta N$, then $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LN$, $\lambda x : M.A \rightarrow_\beta \lambda x : N.A$, $\lambda x : A.M \rightarrow_\beta \lambda x : A.N$, $\Pi x : M.A \rightarrow_\beta \Pi x : N.A$, $\Pi x : A.M \rightarrow_\beta \Pi x : A.N$ and $a(\overline{A}, M, \overline{B}) \rightarrow_\beta a(\overline{A}, N, \overline{B})$.

Now we have defined α-equivalence and β-equivalence for λD^+, we can concentrate on the new equivalence relations. We start by describing δ-equivalence. Before we can give the definition of δ-equivalence, we define the definition for one-step δ-reduction.

Definition 3.3.8 *One-step δ-reduction, $\rightarrow_\delta$*
If $\Gamma \rhd a(\overline{x}) := M : N$ is an element of environment Δ, then
(1) (Basis:) $a(\overline{U}) \rightarrow_\delta M[\overline{x} := \overline{U}]$
(2) (Compatibility:) If $M \rightarrow_\delta N$, then $ML \rightarrow_\delta NL$, $LM \rightarrow_\delta LN$, $\lambda x : M.A \rightarrow_\delta \lambda x : N.A$, $\lambda x : A.M \rightarrow_\delta \lambda x : A.N$, $\Pi x : M.A \rightarrow_\delta \Pi x : N.A$, $\Pi x : A.M \rightarrow_\delta \Pi x : A.N$ and $a(\overline{A}, M, \overline{B}) \rightarrow_\delta a(\overline{A}, N, \overline{B})$.

Also for δ-reduction we can perform multiple reduction steps.

Definition 3.3.9 *δ-reduction (zero-or-more-steps), $\twoheadrightarrow_\delta$*
$M \twoheadrightarrow_\delta N$ if there is an n and there are terms M_0 to M_n such that $M_0 \equiv M$, $M_n \equiv N$ and for all i such that $0 \leq i < n$:

$$M_i \rightarrow_\delta M_{i+1} \ .$$

Now we are able to use a δ-reduction, we can define δ-equivalence.

Definition 3.3.10 *δ-equivalence, $=_\delta$*
$M =_\delta N$ if there is an n and there are terms M_0 to M_n such that $M_0 \equiv M$, $M_n \equiv N$ and for all i such that $0 \leq i < n$:

$$\textit{either } M_i \rightarrow_\delta M_{i+1} \textit{ or } M_{i+1} \rightarrow_\delta M_i \ .$$

Just like in β-conversion we can define a δ-normal form by:

Definition 3.3.11 *unfoldable, δ-normal form, δ-nf*
Let Δ be an environment.
(1) A constant a is unfoldable with respect to Δ, if a is bound to a definition in Δ of the form $\overline{x} : \overline{A} \rhd a(\overline{x}) := M : N$.
(2) K is in δ-normal form with respect to Δ, if there occurs no unfoldable constant in K.
(3) K has a δ-normal form with respect to Δ, if there is an L in δ-nf such that $K =_\delta L$. One also says in this case: K is δ-normalizing and L is a δ-normal form of K (with respect to Δ).

By combining β-reduction and δ-reduction we can create the last equivalence relation, called $\beta\delta$-equivalence. The equivalence is defined as follows:

Definition 3.3.12 *βδ-equivalence or βδ-conversion, $=_{\beta\delta}$*
We say the M βδ-converts to N with respect to Δ, or $M =_{\beta\delta} N$, if there is an n and there are terms M_0 to M_n such that $M_0 \equiv M$, $M_n \equiv N$ and for all i such that $0 \leq i < n$:

$$M_i \to_\beta M_{i+1} \text{ or } M_{i+1} \to_\beta M_i \text{ or } M_i \to_\delta M_{i+1} \text{ or } M_{i+1} \to_\delta M_i \ .$$

3.3.5 Properties of λD^+

In this subsection we give a list of properties that hold for λD^+. The properties are necessary for the rest of the master thesis. We will not give the proofs for the properties. For the proofs we refer to [Ned]. The first property says that λD^+ is an extension of λC.

Lemma 3.3.13 *Conservativity of λD^+ over λC.*
If $\Gamma \vdash M : N$ with respect to λC, then $\emptyset; \Gamma \vdash M : N$ with respect to λD^+.

For the variables and the constant in a judgement we give the following properties.

Lemma 3.3.14 *Free Variables and Constants Lemma*
Let $\Delta; \Gamma \vdash M : N$, where $\Delta \equiv \Delta', \overline{x} : \overline{A} \rhd a(\overline{x}) := K/\perp\!\!\!\perp : L, \Delta''$ and $\Gamma \equiv \overline{y} : \overline{B}$, then:
(1) For all i, $FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ and $FV(K), FV(L) \in \overline{x}$.
(2) For all j, $FV(B_j) \subseteq \{y_1, \ldots, y_{j-1}\}$ and $FV(M), FV(N) \in \overline{y}$.
(3) Constant a does not occur in Δ'
(4) If constant b occurs in $\overline{A}$, K or L then $b \neq a$ and b is the defined constant of some $D \in \Delta'$.
(5) If constant b occurs in $\overline{B}$, M or N, then b is the defined constant of some $D \in \Delta$.

Next we define the legality for expressions, environments and contexts, which in each case means that they are derivable.

Definition 3.3.15 *Legal environment, legal context*
(1) An expression M is called legal, if there exist an environment Δ, a context Γ and N such that $\Delta; \Gamma \vdash M : N$ or $\Delta; \Gamma \vdash N : M$.
(2) An environment Δ is called legal, if there exist a context Γ and M and N such that $\Delta; \Gamma \vdash M : N$.
(3) A context Γ is called legal, if there exist an environment Δ, and M and N such that $\Delta; \Gamma \vdash M : N$.

The following additional properties hold for λD^+ concerning the $\beta\delta$-equivalence relation.

Lemma 3.3.16 *Uniqueness of types up to βδ-conversion.*
If $\Delta; \Gamma \vdash K : B_1$ and $\Delta; \Gamma \vdash K : B_2$, then $B_1 =_{\beta\delta} B_2$

Theorem 3.3.17 *Church-Rosser; CR; Confluence*
Suppose that for a given λ-term M holds that $M \twoheadrightarrow_{\beta\delta} N_1$ and $M \twoheadrightarrow_{\beta\delta} N_2$. Then there is a λ-term N_3 such that $N_1 \twoheadrightarrow_{\beta\delta} N_3$ and $N_2 \twoheadrightarrow_{\beta\delta} N_3$.

Theorem 3.3.18 *Strong Normalizing Theorem or Termination Theorem with respect βδ-conversion.*
Every legal M is strongly normalizing.

Chapter 4

Planning the λC proof checker

4.1 Introduction

For this master thesis two proof checkers are designed and implemented. The proof checkers are implemented in Microsoft Visual Studio with the computer language C#. The first proof checker is implemented for the λ-calculus λC and the second proof checker for λD^+.

Both proof checker are described by a functional and technical description. The functional description describes the design of the proof checker as formulated before the proof checker was implemented. The implementation itself is described in the technical description. In the technical description the used user interfaces, class diagrams and algorithms are explained. The technical description contains no C# code. In the description only UML diagrams and pseudo code are given.

In both the functional and the technical description, we will use the terms 'syntax check' and 'derivability check' very often. Therefore it is needed to make a clear distinction. *Syntax checking* is used to check if a judgement belongs to a given grammar. And the *derivability check* checks whether a judgement that is syntactically correct, can be derived from its derivation rules.

This Chapter will continue by given the functional description of the λC proof checker. The technical description of the λC proof checker is given in Chapter 5. After we have explained the λC proof checker, we give the functional and technical description of the λD^+ proof checker in Chapter 6 and Chapter 7.

4.2 Global overview

In section 2.3 we have seen that existing proof checkers are implemented by two components: a user interface and a kernel component. The user interface is used to communicate with the user, while the real computations are done by the kernel component. Both components are also present in the structure of the λC proof checker. Next to the user interface and the kernel component, the λC proof checker contains a third component, named the eye component. The function of the eye component is to examine the computations of the kernel component. The structure of the λC proof checker is depicted in figure 4.1.

To give a better insight in of the functionality of the various components, we shall give

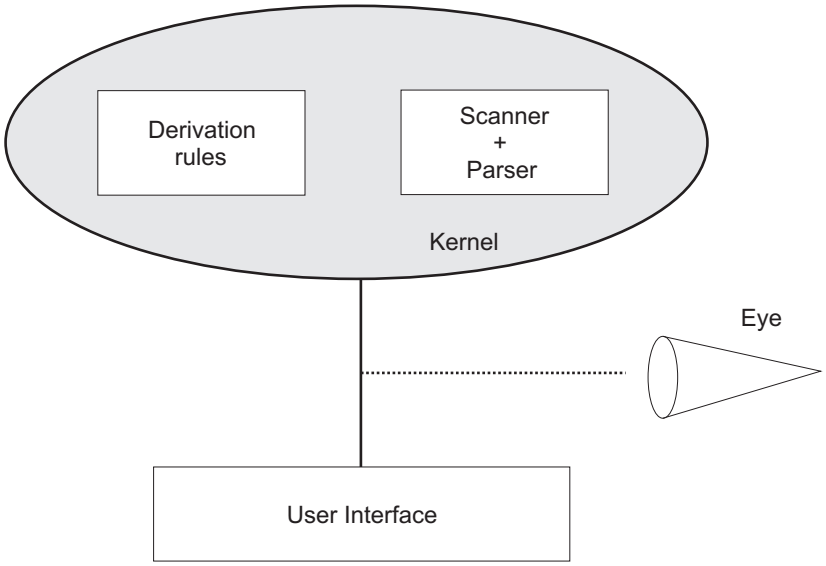

Figure 4.1: Structure of λC proof checker.

a process outline. In the outline the process of checking the derivability of a judgement is given. The outline does not contain a detailed description of the implementation. The outline only illustrates the functionality per component and the communication between the distinct components.

Outline 4.2.1 *In the user interface, one is able to enter an input judgement. The input judgement is offered to the kernel component, where the judgement is checked on its syntax. If the judgement contains one or more syntax errors the process stops and an error message is shown in the user interface to inform the user. If the input judgement is syntactically correct, the kernel component checks the derivability of the corresponding judgement. After the derivability check the focus returns to the user interface component. Based on the outcome of the derivability check two situations are distinguished, namely the input judgement is derivable or the judgement is not derivable. When the input judgement is derivable, the complete derivation is shown in the user interface. If the input judgement is not derivable, the derivation is shown from which one can conclude that the input judgement is not derivable by the derivation rules of λC.*

To examine the derivability check of the kernel component, the eye component is added. The eye component stores a 'trace' of internal steps that were executed during the derivability check. With the trace the user can verify the correctness of the derivability check. The 'trace' of internal steps is reported in the user interface component.

From the given process outline we can easily see the main tasks of the components. The core component of the λC proof checker is the kernel component. The kernel component implements the algorithms that can verify the syntax and the derivability of a judgement. To execute the tasks, the kernel component uses the user interface and the eye component. Both components are mainly added to communicate with the user. In the user interface one is able to input a judgement and to report on the results of the syntax and derivability check. The

eye component is added to give the user some extra information concerning the execution of the derivability check.

4.3 Functional specification

In this section the functional specification of the λC proof checker is given. As already mentioned in subsection 4.2, the λC proof checker is divided in three separate components: user interface, kernel and eye component. In this section the functional specification is described per component.

4.3.1 User interface component

Requirement 4.3.1 *Create judgement.*
The user is able to construct a judgement.

Requirement 4.3.2 *Judgements are conform a grammar.*
The input judgement must satisfy the following grammar:

$$
\begin{array}{lll}
\textit{Judgement} & ::= & \textit{Context} \vdash \textit{Statement} \\
\textit{Context} & ::= & \emptyset \mid \textit{Declaration} \mid \textit{Context, Context} \\
\textit{Declaration} & ::= & \textit{Variable : Term} \\
\textit{Statement} & ::= & \textit{Term : Term} \\
\textit{Term} & ::= & \textit{Variable} \mid \Box \mid * \mid \textit{(TermTerm)} \mid \lambda\ \textit{Declaration . Term} \mid \\
 & & \Pi\ \textit{Declaration. Term} \\
\textit{Variable} & ::= & [a \ldots z]^{+}\textit{, where } ^{+} = \textit{one or more}
\end{array}
$$

Requirement 4.3.3 *Check derivability of judgement.*
In the user interface the user is able to start the derivability check implemented in the kernel component (see requirement 4.3.13) for the defined judgement. A judgement is derivable if one can derive the judgement using the derivation rules.

Requirement 4.3.4 *Show output of kernel component.*
A judgement is checked on its derivability by the kernel component (see requirement 4.3.3). The kernel component returns an output value based on the derivability check. The output can be divided into two types:

1. *The input judgement is derivable, see requirement 4.3.5.*
2. *The input judgement is not derivable, see requirement 4.3.6.*

Requirement 4.3.5 *Show correct judgement derivation.*
The user interface can report the derivation tree of a derivable input judgement. The derivation tree is reported in list notation. An example of the list notation is given in figure 4.2.

Requirement 4.3.6 *Show incorrect judgement derivation.*
When the judgement is not derivable with the derivation rules, the interface reports the derivation step of the derivation tree in which no new iteration is possible. The incomplete derivation tree of the input judgement is reported in list notation (see Figure 4.2).

(1)	$\emptyset \vdash * : \Box$	(ax)
(2)	$\alpha : * \vdash \alpha : *$	(start (1))
(3)	$\alpha : *, x : \alpha \vdash x : \alpha$	(start (2))
(4)	$\alpha : *, x : \alpha \vdash \alpha : *$	(weak (2), (2))
(5)	$\alpha : * \vdash \Pi x : \alpha.\alpha : *$	(form (2), (4))
(6)	$\alpha : * \vdash \lambda x : \alpha.x : \Pi x : \alpha.\alpha$	(abst (3), (5))

Figure 4.2: Example of derivation tree in list notation.

4.3.2 Kernel component

Requirement 4.3.7 *Retrieve input judgement.*
The kernel component is able to retrieve an input judgement, entered in the user interface (see requirement 4.3.1).

Requirement 4.3.8 *Check syntax of input judgement.*
The kernel component is able to verify the syntax of an input judgement. The judgement is syntactically correct if it can be derived with the grammar described in requirement 4.3.2. Otherwise the judgement is not syntactically correct.

Requirement 4.3.9 *Parse syntactically correct input judgement.*
If an input judgement is syntactically correct (see requirement 4.3.8), the kernel component is able to parse the input judgement into a data structure called a parse tree.

Requirement 4.3.10 *Notify syntactically incorrect input judgement.*
If an input judgement is syntactically incorrect then the kernel component is unable to parse the judgement into a parse tree. The kernel component returns an indication of the syntax error.

Requirement 4.3.11 *Ensure Barendregt-convention.*
When parsing the input statement into a tree structure, the Barendregt-convention must be satisfied. To ensure the Barendregt-convention it is allowed to rename binding variables in the judgement using α-conversion (see requirement 4.3.17).

Requirement 4.3.12 *No free variables in judgement.*
When the judgement is parsed into a tree structure, the tree structure cannot contain any free variables. Each variable is bound by:

1. *λ-term, or*
2. *Π-term, or*
3. *context declaration.*

If the input judgement contains free variables, the kernel component is unable to execute the derivability check.

Requirement 4.3.13 *Check derivability.*
The kernel component is able to check if the input judgement is derivable with the derivation rules of λC. To check the derivability, the proof checker constructs the derivation tree of the input judgement. The derivability check is proceeded by:

1. *Syntax check of input judgement (see requirement 4.3.8).*
2. *Check of free variable lemma (see requirement 4.3.12).*
3. *Ensure Barendregt-convention (see requirement 4.3.11).*

Requirement 4.3.14 *Check each judgement on its derivability at most once.*
Each unique judgement in the derivation tree is checked at most once on its derivability. If judgement $J1$ and $J2$ are equal, the judgements are combined in the derivation tree.

Requirement 4.3.15 *Derivation rules.*
In the kernel component all the derivation rules (see section 3.2) of λC are included. This means that all judgements of the grammar described in requirement 4.3.2 can be checked on their derivability.

Requirement 4.3.16 *α-equivalence.*
The kernel component is able to see if two statements are α-equivalent.

Requirement 4.3.17 *Perform α-conversion.*
The kernel component is able to execute α-conversion. The α-conversion is used in two ways:

1. *Although α-conversion is not part of the derivation rules, it must be included in the derivability check. For example the kernel is able to see that the following two terms are α-equivalent:*
   (a) $\lambda\alpha : *.\lambda x : \alpha.x$
   (b) $\lambda\beta : *.\lambda x : \beta.x$
2. *For establishing the Barendregt-convention (see requirement 4.3.11).*

Requirement 4.3.18 *Perform one-step β-reduction.*
The kernel is able to perform an one-step β-reduction.

Requirement 4.3.19 *β-equivalence.*
The kernel component is able to see if two statements are β-equivalent. In the conv derivation rule β-equivalence is needed.

Requirement 4.3.20 *Store derivation tree.*
The kernel component stores the derivation tree of the input judgement, when checking the derivability. In the derivation tree every derivation rule is explicitly labeled with its name, premisses and conclusion.

Requirement 4.3.21 *Store incorrect judgements.*
If a derivation cannot be finished, because a judgement cannot be derived by a derivation rule then the kernel must store this judgement.

Requirement 4.3.22 *Internal steps as input for eye component.*
All the internal steps of the kernel component in the derivability check are sent to the eye component (see requirement 4.3.23).With this information the user can overview and inspect the complete process in the eye component.

4.3.3 Eye component

Requirement 4.3.23 *Retrieve input from kernel component.*
The eye component is able to retrieve input from the kernel component.

Requirement 4.3.24 *Show derivation steps.*
The eye component is able to show each internal action of the kernel component. In this way the user is able to monitor the derivability check.

4.4 Iterative schedule

The λC proof checker is implemented in four iterations. This section gives a description of each iteration. In the description the requirements are mentioned that are covered by the iteration. The four iterations will cover all the requirements of the functional specification described in section 4.3 and will lead to the complete λC proof checker.

4.4.1 Iteration 1

In the first iteration a simple user interface will be implemented. In the user interface, the user is able to enter an input judgement. This input judgement is used as input for the kernel component. The kernel component is able to check the input judgement on its syntax and on its derivability. Only the derivability check is not completely implemented:

1. β-reduction is not possible. Therefore the following *conv* derivation rule is not implemented.

2. A premiss judgement can occur more than once in the derivation tree. In this iteration two judgements cannot share their computed derivation. This means that each judgement will be recursively checked by means of the full derivation tree.

3. In the first iteration, we assume that the Barendregt-convention (see requirement 4.3.11) holds for the input judgement. And that the input judgement does not contain variables that are not defined in the context or by a λ or Π term.

For the first iteration the following requirements are implemented:

User interface component:
Requirement 4.3.1, Create judgement.
Requirement 4.3.2, Judgements are conform a grammar.
Requirement 4.3.3 (partly), Check derivability of judgement.
Requirement 4.3.4, Show output of kernel component.
Requirement 4.3.5, Show correct judgement derivation.
Requirement 4.3.6, Show incorrect judgement derivation.
Kernel component:
Requirement 4.3.7, Retrieve input judgement.
Requirement 4.3.8, Check syntax of input judgement.
Requirement 4.3.9, Parse syntactically correct input judgement.
Requirement 4.3.10, Notify syntactically incorrect input judgement.
Requirement 4.3.13 (partly), Check derivability.

Requirement 4.3.15 (partly), Derivation rules.
Requirement 4.3.20, Store derivation tree.
Requirement 4.3.21, Store incorrect judgements.

4.4.2 Iteration 2

The second iteration will implement the correctness check completely. When this iteration is implemented all λC judgements, can be checked for their derivability. The following functionality is constructed:

1. α-equivalence
2. β-equivalence and therefore also the conversion rule.
3. Type checking algorithm.

The second iteration implements the following requirements:

User interface component:
Requirement 4.3.3, Check derivability of judgement.
Kernel component:
Requirement 4.3.11, Ensure Barendregt-convention.
Requirement 4.3.12, No free variables in judgement.
Requirement 4.3.13, Check derivability.
Requirement 4.3.16, α-equivalence.
Requirement 4.3.17, Perform α-conversion.
Requirement 4.3.18, Perform one-step β-reduction.
Requirement 4.3.19, β-equivalence.

4.4.3 Iteration 3

In the first two iterations the syntax and correctness check are implemented. In this iteration we try to increase the time performance of the derivability check. In the first two iterations each judgement is computed completely. So if a judgement occurs twice as a premiss in some derivation step then the same judgement is checked twice. To increase the performance and the readability of the derivation, every judgement is checked at most one time.
In the third iteration the following requirement is implemented:

Kernel component:
Requirement 4.3.14, Check each judgement on its derivability at most once.

4.4.4 Iteration 4

In the last iteration the eye component is implemented. With the eye component the user can see all the internal steps of the kernel component. When this iteration is finished, the λC proof checker is fully operational and contains the three implemented components: user interface, kernel and eye.
In the fourth iteration the following requirements are implemented:

Kernel component:

Requirement 4.3.22, Internal steps as input for eye component.

Eye component:

Requirement 4.3.23, Retrieve input from kernel component.

Requirement 4.3.24, Show derivation steps.

Chapter 5

Proof checker λC

5.1 User interface λC proof checker

Before we can give an actual description of the algorithm, we shall give the user interface of the λC proof checker. The user interface contains three forms. In the first form the user is able to give an input judgement for the λC proof checker and one can start the syntax and the derivibility check for the input judgement. This form is described in section 5.1.1. In the second form the outcome of the derivability check is reported. A description of the form is given in section 5.1.2. Finally a form is added to implement the eye component. The description of this form is given in section 5.1.3.

5.1.1 Input judgement

In this section we describe how one gives an input judgement to the λC proof checker. The input judgement shall serve as input for the syntax check and the derivability check. In the main form of the λC proof checker, the user can enter a judgement. The main form is given in figure 5.1.

The language that is used to define a judgement is an one-to-one translation of the λC grammar that has been described in section 3.2. The translation is given in table 5.1.

Non-terminal	Element	Translation
Judgement	$Context \vdash Statement$	\judgement{Context}{Statement}
Context	$Declaration_1, \ldots, Declaration_n$	\context{{Declaration$_1$}...{Declaration$_n$}}
Context	$\emptyset$	\context{ {\$\emptyset$\$} }
Statement	$Term$: $Term$	\statement{Term}{Term}
Declaration	V: $Term$	\declaration{V}{Term}
Term	(λ $Declaration.Term$)	\abstraction{Declaration}{Term}
Term	($TermTerm$)	\application {Term}{Term}
Term	(Π $Declaration.$ $Term$)	\typebinder{Declaration}{Term}
Term	Variable x	\variable{x}
Term	$*$	\ast{}
Term	$\Box$	\square{}

Table 5.1: Translation of λC grammar.

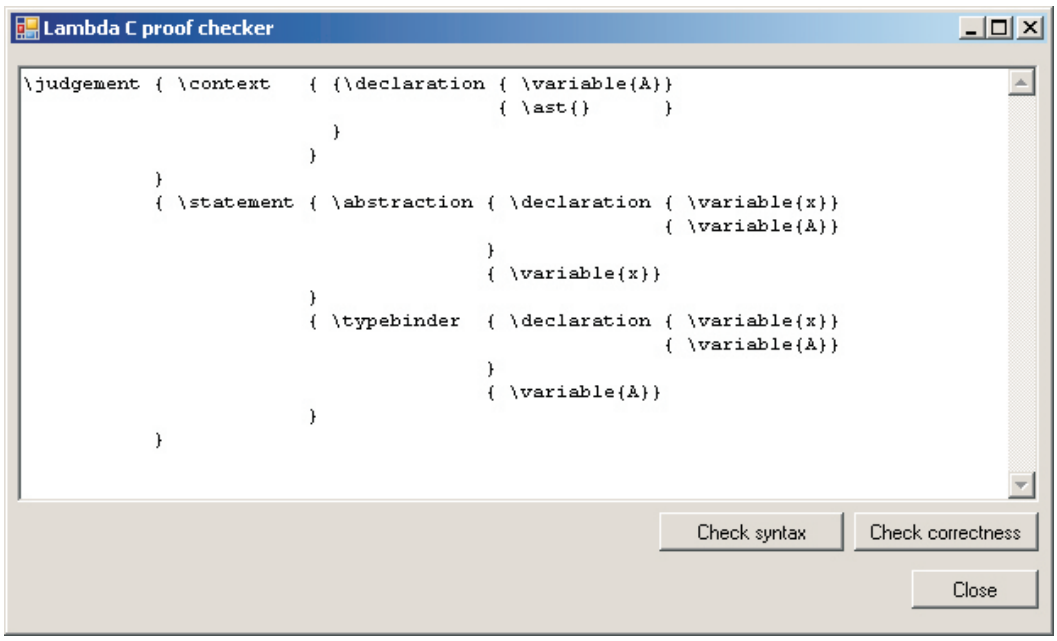

Figure 5.1: Main form of λC proof checker.

The main reason to use a new format of the λC grammar is that we want to construct an efficient and deterministic parsing algorithm. In section 5.2 this will become clear when we describe the parser of the λC proof checker. Another reason is that the λC grammar contains the Greek letters λ and Π for abstraction and typebinding. Because the Greek alphabet is not part of a standard keyboard, the process of defining a judgement is fairly hard. Therefore the translation only uses characters that are on a standard keyboard.

With the translations, given in table 5.1, we can formulate the context-free grammar $G = (N, \Sigma, P, S)$ for the input language of the λC proof checker.

- The set of non-terminals N is defined by the elements *Abstraction*, *Application*, *Context*, *Declaration*, *Declarations*, *Judgement*, *KindCat*, *Statement*, *Term*, *TypeBinder*, and *TypeCat*.
- The set of terminals Σ is defined by the elements *\abstraction*, *\application*, *\ast*, *\context*, *\declaration*, *\judgement*, *\square*, *\statement*, *\typebinder*, *\variable*, { and }.
- The start element S is the non-terminal symbol *Judgement*.
- The production rules P are defined as shown in figure 5.2.

Because the λC proof checker uses a direct (one-to-one) translation of the λC grammar, one can enter all valid λC judgements in the system. The only restriction of the translations is, that the names of the variables are formed by a defined set of characters.

Judgement := *judgement* { Context } { Statement }
Context := *context* { Declarations } | *context* { { \emptyset{ } } }
Declarations := { Declaration } Declarations | { Declaration }
Declaration := *declaration* { Variable } { Term }
Statement := *statement* { Term } { Term }
Term := Abstraction | Application | TypeBinder | TypeCat | KindCat | Variable
Abstraction := *abstraction* { Declaration } { Term }
Application := *application* { Term } { Term }
TypeBinder := *typebinder* { Declaration } { Term }
TypeCat := *ast* {}
KindCat := *square* {}
Variable := *variable* { ([a...z] + [A...Z] + [0...9])$^+$ }

Figure 5.2: Production rules of context-free grammar for input language λC proof checker.

If one wants to verify the syntax of the defined judgement one can press the button 'Check syntax'. To start the derivability check the user must press the button 'Check correctness'. Before the derivability check is started, always the syntax of the input judgement if verified. If the input judgement contains syntax errors, the derivability check is not executed.

Example 5.1.1 *If we want to check the judgement* $A : * \vdash \lambda x : A.x : \Pi x : A.A$ *with the* λC *proof checker, one must enter the following judgement:*

```
\judgement { \context    { { \declaration { \variable{A}}
                                          { \ast{}       }
                           }
                         }
           }
           { \statement {   \abstraction { \declaration { \variable{x} }
                                                        { \variable{A}}
                                          }
                                          { \variable{x} }
                         }
                         {   \typebinder { \declaration { \variable{x} }
                                                        { \variable{A}}
                                          }
                                          { \variable{A}}
                         }
           }
```

5.1.2 Output of derivability check

The user interface of the λC proof checker contains a form to report the outcome of the derivability check. In the form the derivation tree is reported in list notation. If the input judgement is not derivable the form also reports the errors. In figure 5.3 the output form of the λC proof checker is depicted. In the figure the derivation tree of the judgement in

example 5.1.1 is reported. Note that this judgement is derivable in λC. One can also report the outcome of the derivability check in Microsoft Excel by pressing the button 'Copy To'.

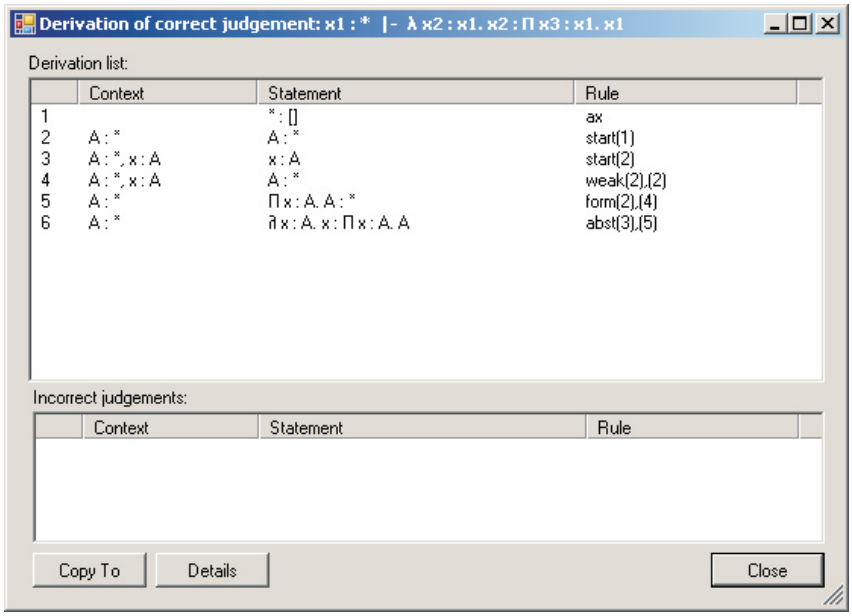

Figure 5.3: Output form of λC proof checker.

To open the eye component for a judgement, one must select an item in the list and press the button 'Details'. When pressing the button the form described in subsection 5.1.3 is opened.

5.1.3 Eye component

For the eye component a form is added to the user interface. The user is able to open the form after the derivability check of an input judgement J. The form is opened from the output form by the button 'Details' (see figure 5.3). The form reports all available data of one judgement J' from the derivation tree, after J is checked on its derivability. This includes:

- The derivation rule where J' is the conclusion.
- The premisses of the derivation rule where J' is the conclusion.
- The internal representation of J' (see section 5.3.2).
- The internal steps of the derivability checked needed to derive J'.

In figure 5.4 the eye component form is opened for the judgement $A : * \vdash \lambda x : A.x : \Pi x : A.A$.

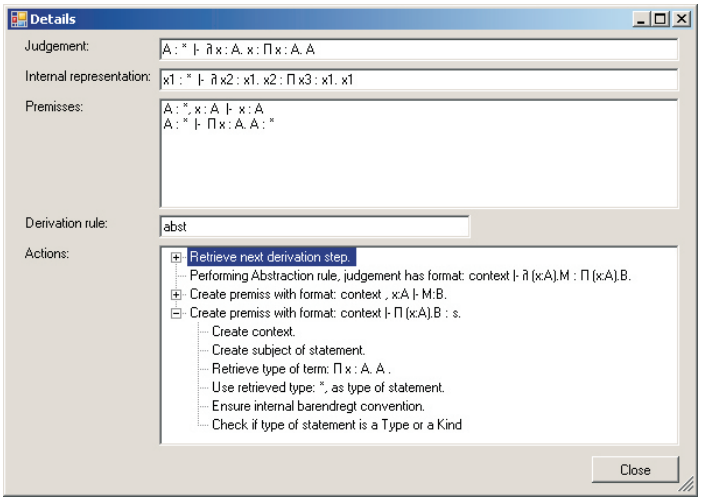

Figure 5.4: User interface of eye component.

5.2 Checking the syntax

In the previous section we have described how a user can input a judgement into the λC proof checker. Before the system checks if the input judgement is derivable with the derivation rules of λC, the judgement is checked on its syntax. The syntax check is executed in two steps:

- In the first step the input judgement is processed by the *lexical scanner*. The input judgement consist of a row of characters. The task of the lexical analyser is to read, group and categorize the row of characters into objects, called *tokens*. The lexical scanner is described in paragraph 5.2.1.

- In the second step the *parser* checks the syntax of the input judgement according to the row of defined tokens. To check the syntax, one must find a derivation of production rules for the input judgement. Such a derivation can also be depicted in a tree structure, called a *parse tree*. The parse tree of the input judgement will eventually be used as input to determine the derivability of the corresponding judgement. The parser is described in subsection 5.2.2.

5.2.1 Scanner

The judgement that is defined in the user interface component consists of a row of characters. Initially the characters do not have any meaning for the kernel component. The task of the lexical scanner within the kernel component is to read, group and divide the row of characters into objects called tokens. The lexical scanner works in service of the parser. This means that the lexical scanner only returns a token, if asked by the parser. The task of the lexical analyzer is to maintain the position of the next token. A token that is categorized by the lexical scanner can be recognized by the parser when checking the syntax.

For each token rules are defined. The rules define which chain of characters can form a valid token. The lexical scanner reads the input judgement from left to right, character by character. It is known after each character, whether the sequence of succeeding characters

forms a token or a substring of a token. If the lexical scanner finds a sequence of succeeding characters that does not belong to a token, a message is given to the parser. The message is given because the sequence of characters causes a syntax error. Therefore the lexical scanner is also used as a first check when finding syntax errors.

Now we know what the functionality of a lexical scanner is, we can look at the implementation that is used for the λC proof checker. The class diagram of the lexical scanner is given in figure 5.5.

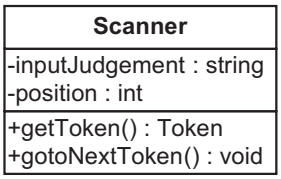

Figure 5.5: Class diagram lexical scanner.

The attribute *inputJudgement* stores the judgement that is entered by the user in the user interface. The attribute *position* is used to indicate which characters of *inputJudgement* are already read and categorized into tokens. The characters from position zero till *position* are already processed by the lexical scanner.

Next to the two attributes, the lexical scanner also contains two functions:

- Function GETTOKEN. The function returns the next token in *inputJudgement* after position *position*, without changing the value of *position*.
- Procedure GOTONEXTTOKEN. The procedure moves the cursor *position* to the end of the current token in *inputJudgement*.

The tokens that can be categorized by the lexical scanner are based on the terminals (see section 5.1.1) of the context-free grammar that defines the input language of the λC proof checker. In table 5.2 a summary is given of the available tokens, with the corresponding criteria. Note that the lexical scanner returns a token ERROROBJECT if the scanner cannot define a token. The token ERROROBJECT contains a suitable error message that is used to inform the user about the syntax error.

5.2.2 Parsing

The context-free grammar (see section 5.1.1) that defines the input language of the λC proof checker, contains production rules that describe the syntactical structure of a well-formed judgement. The task of the parser is to verify, whether a given row of tokens can be deduced from these production rules. One deduction step is defined by:

Definition 5.2.1 *One deduction step*
Let $G = (N, \Sigma, P, S)$ be a context-free grammar. A deduction step $\alpha \Rightarrow \beta$ is valid if:

- *α and $\beta \in (N \cup \Sigma)^+$.*
- *α is defined by $\delta_1\ C\ \delta_2$ and β is defined by $\delta_1\ \gamma\ \delta_2$, where $C \in N$, $\delta_1, \delta_2 \in (N \cup \Sigma)^*$, $\gamma \in (N \cup \Sigma)^+$ and $C \rightarrow \gamma \in P$.*

Character sequence	Token type
\abstraction	ABSTRACTION
\application	APPLICATION
\ast	TYPECAT
\context	CONTEXT
\declaration	DECLARATION
\emptyset	EMPTYSET
\judgement	JUDGEMENT
\square	KINDCAT
\statement	STATEMENT
\typebinder	TYPEBINDER
\variable	VARIABLE
{	BEGINBRACKET
}	ENDBRACKET
$[a \dots z]^+$	TEXT, with *value* property equal to $[a \dots z]^+$
position exceeds length of *inputJudgement*	EOF
Otherwise	ERROROBJECT, with *message*.

Table 5.2: Definition of token in λC proof checker.

A complete deduction is called *leftmost* if the leftmost non-terminal is always the one used in the next deduction step. A leftmost deduction only contains deduction steps where $\delta_1 \in \Sigma^*$. A deduction is called *rightmost* if the deduction only contains deduction steps where $\delta_2 \in \Sigma^*$.

LL(1) grammar

There exist several types of parsing algorithms, all trying to find a deduction of production rules for a given row of tokens. One of the design decisions for the λC proof checker is that the implementation must be structured and well-ordered. In this way, one is able to check the code by hand on its correctness and therefore the trust in an error-free system is guaranteed. This design decision also holds for the parsing algoritm. For this reason we choose a well-structured and deterministic parsing algorithm for the λC proof checker. This means that at every moment in time during the parsing process it is clear what the next step in the algorithm is. The parsing algorithm that is used is called an LL(1) parser.

The abbreviation LL(1) stands for 'Leftmost with a Lookahead of 1'. As the name says, the LL(1) parser tries to find a leftmost deduction. An LL(1) parser is a top-down algorithm, which means that the parsing process starts with the start symbol of the context-free grammar. The LL(1) parser chooses every next deduction step by looking one token ahead.

Not every context-free grammar can be checked on its syntax by an LL(1) parser. An LL(1) parser can only parse an LL(1) grammar. An LL(1) grammar is a context-free grammar with extra restrictions. To define an LL(1) grammar, we first give the definition of a *Lookahead* set (see [Jeu]).

Definition 5.2.2 *Lookahead set*
For the non-empty context-free grammar $G = (N, \Sigma, P, S)$, the lookahead set of a production $K \to \alpha$ is the set of terminal symbols that can appear as the first symbol of a string that can be derived, when K is the start symbol of G and the deduction is started with the production rule $K \to \alpha$. So

$$\textit{Lookahead}(K \to \alpha) := \{x \in N | K \Rightarrow \alpha \Rightarrow^* x\beta\ \}, \textit{ for some } \beta \in (N \cup \Sigma)^*.$$

Now that we are able to define a Lookahead set for each production rule, we can formalize the definition of an LL(1) grammar (see [Jeu]).

Definition 5.2.3 *LL(1) grammar*
A context-free grammar $G = (N, \Sigma, P, S)$ is LL(1) if all pairs of productions of the same non-terminal have disjoint LookAhead *sets, that is for all productions $K \to \alpha$, $K \to \beta \in P$ and $\alpha \neq \beta$:*

$$\text{LookAhead}(K \to \alpha) \cap \text{LookAhead}(K \to \beta) = \emptyset$$

The last task is to check if the input language of the λC proof checker satisfies the definition of an LL(1) grammar. If we construct the lookahead set for each production rule, we observe that the intersection of the lookahead sets of the non-terminals Context and Declarations is not empty. The intersection of the lookahead sets for the production rules for Context contains the terminal *context*. And the intersection of the lookahead sets for the production rules for Declarations contains the terminal }. By rewriting the production rules of the non-terminals Context and Declarations, we can nevertheless satisfy the definition of an LL(1) grammar. The new production rules are given in figure 5.6.

```
Context        := \context { { Declarations_1
Declarations_1 := Declaration } Declaration_2 | \emptyset{} } }
Declarations_2 := } | { Declaration } Declarations_2
```

Figure 5.6: Adaptation of production rules for input language.

With the new production rules, the input language of the λC proof checker is an LL(1) grammar and an LL(1) parser can be implemented. The implementation of the LL(1) parser uses the same structure for each production rule. A production rule can be written as $C \to \alpha$, where C is a non-terminal symbol and α a sequence of terminal and/or non-terminal symbols with a minimal length of one. For each symbol in α code is added for the parser of the corresponding production rule. When adding code, we distinguish between a terminal and a non-terminal symbol. For each terminal symbol T we add the following pseudo code:

Algorithm 1: Terminal parsing

```
switch scanner.getToken do
    case T
        scanner.gotoNextToken
    case ErrorObject
        Stop parsing algorithm and return error message of ErrorObject
    otherwise
        Stop parsing algoritm en return message of the syntax error.
```

And for each non-terminal symbol V:

Algorithm 2: Non-terminal parsing

```
switch scanner.getToken do
    case An element of ⋃{Lookahead(V → α)|V → α ∈ P}
        Go into recursion for production rule V → α.
    case ErrorObject
        Stop parsing algorithm and return error message of ErrorObject
    otherwise
        Stop parsing algoritm en return message of the syntax error.
```

Because the LL(1) parser checks the syntax after each individual token, syntax errors are found quickly and a suitable error message can be given to the user. If the parser finds a syntax error, the parsing algorithm will stop immediately.

Remark 5.2.4 *The reason that we choose an LL(1) parser in the λC proof checker is completely based on the fact that it is a deterministic algorithm. An additional advantage is the efficiency of the LL(1) parser. The LL(1) parser has a running time of $O(n)$, where n is defined as the number of tokens in the input string. The LL(1) parser is linear since the algorithm does not use backtracking and each token is read a constant (two times) number of times.*

Other parsing algorithm are mostly less efficient. In [Hop] is proven that for each context-free grammar, one can construct a parser with a running time of $O(n^3)$.

Parse tree

The parser has a second task next to the task of checking the syntax for a judgement. The parser transforms the input judgement into a data structure that is used to determine the derivability of the corresponding judgement (see section 5.4). Such a data structure is called a *parse tree*. In the parse tree the structure of the syntax deduction is stored.

The available nodes of the parse tree are an extension of the tokens that can be recognized by the lexical scanner. In figure 5.7 the class diagram is given of the classes that are used in the parse tree.

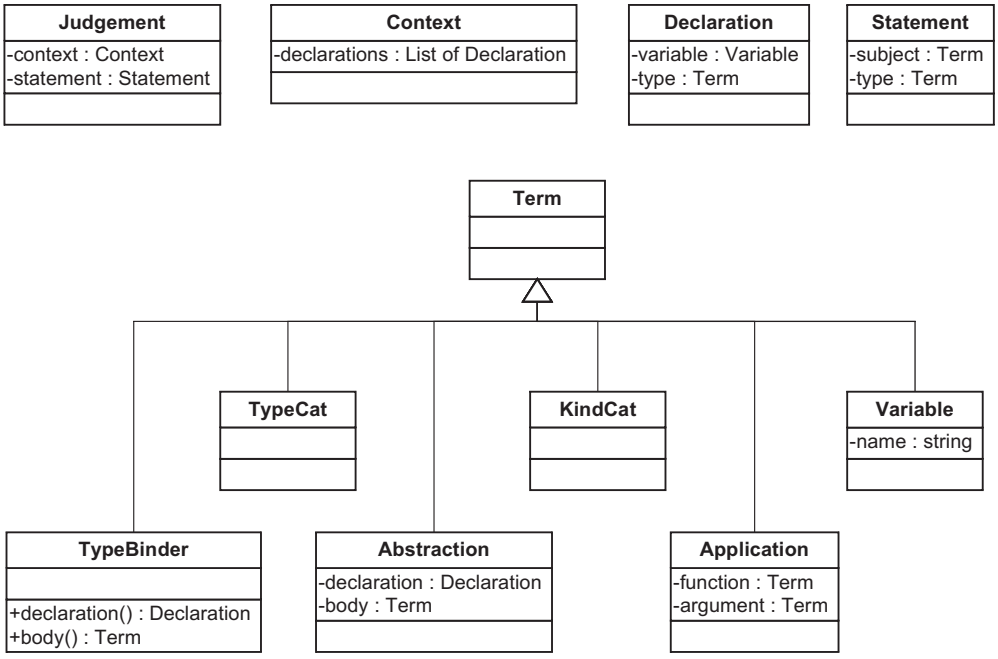

Figure 5.7: Class diagram of parse tree.

Example 5.2.5 *The parse tree of the judgement, given in example 5.1.1 is depicted in figure 5.8.*

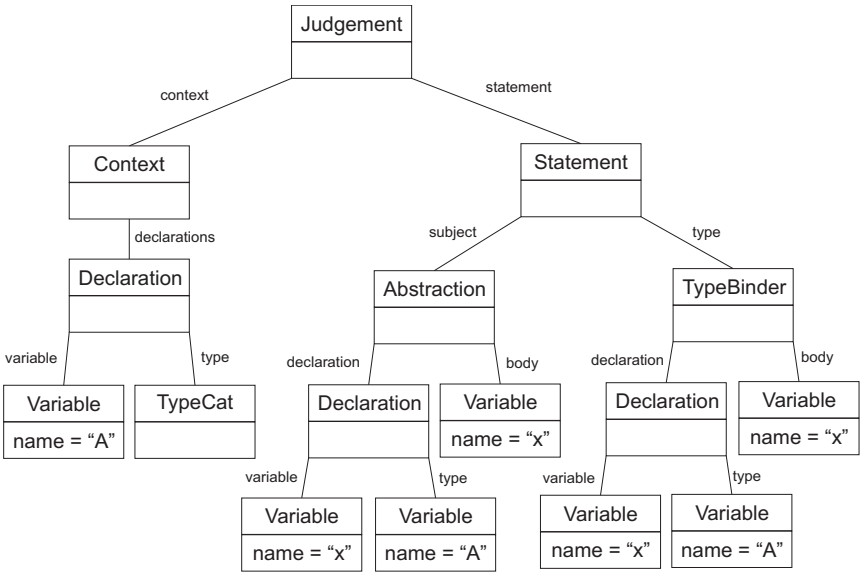

Figure 5.8: Parse tree of judgement from example 5.1.1.

5.3 Variables and reduction

We have given a description of the user interface and the parser of the λC proof checker. We saw that during the parsing algorithm the input judgement is transformed into a parse tree. With the parse tree we are able to check the derivability of the input judgement. Before the λC proof checker executes the derivability check, we check if all variables in the judgement are declared in the context, by a λ-term or a Π-term. If the input judgement contains free variables, an error message is given. If all variables are declared in the input judgement, the judgement is adapted to make sure that the Barendregt-convention holds. The parse tree of the input judgement is given as input for both algorithms. The free variable lemma is described in subsection 5.3.1 and the Barendregt-convention in subsection 5.3.2.

Next to the free variable check and the Barendregt-convention, this section also gives a description of the implementation for α-reduction and β-reduction. Both reductions are described in subsection 5.3.3.

5.3.1 Free variables lemma

Before we determine the derivability of a judgement, the system checks if the input judgement does not contain any free variables. We have the Free Variable lemma in λC:

Lemma 5.3.1 *Free Variable Lemma*
If $\Gamma \vdash A : B$*, then* $FV(A) \cup FV(B) \subseteq dom(\Gamma)$.

The free variable lemma states that a judgement $\Gamma \vdash A : B$ is only derivable if all free variables of the statement $A : B$ are declared in context Γ. How do we know that a judgement is not derivable from the derivation rules when it contains free variables?

If we look at the derivation rules of λC, we can see that only in the *abst*, *form* and *start* derivation rules, variables are introduced in the statement of the judgement. Within all three derivation rules, the variables are bound. In the *start* derivation rule the variable is bound to a context declaration, in the *form* rule the variable is bound to a Π-term and in the *abst* rule the variable is bound by a λ-term.

Therefore we know that a judgement is not derivable, if it contains free variables. The proof checker will find this when checking the derivability of the corresponding judgement. The reason that we have chosen to check the free variable lemma before the derivability check is out of practical reasons. In practice it turns out that typing errors are often made by the user of the λC proof checker. Therefore it can occur that an input judgement contains free variables, while this was not the intention of the user. By informing the user in time, one is able to adjust the input judgement without waiting for the execution of the derivability check.

The algorithm that implements the check for free variables, will not be given in the master thesis. The algorithm is a simple top-down algorithm that searches free variables in the parse tree of the input judgement.

5.3.2 Barendregt convention

The Barendregt-convention is not enforced in the definition of the context-free grammar that defines the input language of the λC proof checker. This means that the Barendregt-convention does not have to hold for a syntactically correct judgement. It is only desirable

that the Barendregt-convention holds for the input judgement. In section 5.3.3 we will explain this further. We can define the Barendregt-convention by:

Convention 5.3.2 *Barendregt-convention*
We choose the names of the binding (and the corresponding bound) variables in a λ-term in such a manner, that all binding variables (behind the λ's en Π's) are mutually different, and such that each of them differs from all free variables occurring in the term.

To make sure that the Barendregt-convention holds for the given input judgement, all variables in the judgement are renamed. In the renaming no distinction is made between a context and a λ or Π variable. If we see the judgement as a textual value instead of a parse tree, the variables are renamed from left-to-right. The first declared variable is renamed to $x1$, the second to $x2$, etc. All bound variables are renamed to the new name of their corresponding binding variable. Note that after the renaming, all declarations declare a variable with a unique name. Because of this, we know that the Barendregt-convention holds for the input judgement.

Example 5.3.3 *To clarify the renaming of the variables within the input judgement, we shall give an example. In the example the judgement* $A : * \vdash \lambda x : A.(\lambda x : A.x)x : \Pi x : A.A$ *is used. Note that the judgement is syntactically correct, but still the Barendregt-convention does not hold since the variable x is declared three times. The judgement is renamed to:* $x1 : * \vdash \lambda x2 : x1.(\lambda x3 : x1.x3)x2 : \Pi x4 : x1.x1$.

One could ask if it is practical to rename all variables to meaningless names. The user could have chosen the names in such a way that they have a special meaning. The process outline in section 4.2 describes that a derivation is shown after the judgement is checked on its derivability. In the output form (see section 5.1.2) the names of the variables occur in the derivation. Because the variable names can have a meaning, it is desirable to show the names of the variables that have been chosen by the user in the derivation.

To implement this and to make sure that the Barendregt-convention still holds for the input judgement, the class VARIABLE (class of parse tree) is extended with an extra attribute called *externalName*. The attribute stores the name of the variable that is given by user. The new class diagram of the class VARIABLE is given in figure 5.9.

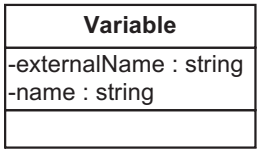

Figure 5.9: Class diagram of VARIABLE class.

In the parsing process, the attributes *name* and *externalName* initially get the same value. The value is equal to the variable name that is defined by the user. When all the variables are renamed to satisfy the Barendregt-convention, only the attribute *name* is used and the value of *externalName* is maintained.

In all the algorithms of the λC proof checker the attribute *name* is used per VARIABLE object, while the attribute *externalName* is only used to communicate with the user, for example in the output form.

Remark 5.3.4 *Note that by using the same method for each judgement, all α-equivalent judgements are renamed equally.*

5.3.3 Reduction

α-conversie

In the derivation rules of λC, α-equivalence is implicitly added. If we say in λC that a term A is equal to a term B then we mean that the terms A and B are α-equivalent. In the next section, where a description of the derivability check is given, we use the equality between two terms very often. Therefore the λC proof checker contains an algorithm that verifies the α-equivalence between two terms.

The algorithm determines the α-equivalence between two terms by comparing the parse tree of both terms. If term A is α-equivalent to term B then the parse tree of A is isomorphic to the parse tree of B. The structure of the parse tree is defined by the node types.

But the isomorphism relation is not enough to define α-equivalence, since the parse tree of A can have a VARIABLE object with name x where the parse tree of B contains a VARIABLE object with the name y. So we must add the extra restriction that all the VARIABLE objects of x in the parse of A are equal to the variable y. With this extra restriction we implement the α-equivalence relation between two terms by using a top-down algorithm.

One β-reduction step

Not only α-conversion but also β-reduction and β-equivalence are implemented in the λC proof checker. Before we give the algorithm for β-equivalence, we define the algorithm to perform one β-reduction step. One β-reduction step is defined by:

Definition 5.3.5 *One-step β-reduction, $\rightarrow_\beta$*
(1) (Basis:) $(\lambda x : A.M)N \rightarrow_\beta M[x := N]$
(2) (Compatibility:) If $M \rightarrow_\beta N$, *then* $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LM$, $\lambda y : B.M \rightarrow_\beta \lambda y : B.N$ *and* $\Pi y : B.M \rightarrow_\beta \Pi y : B.N$.

The β-reduction steps are executed in the parse tree of a λC-term. One can perform a β-reduction step if the parse tree contains a β-redex. A β-redex is a λC-term with the format $(\lambda x : A.M)N$. In the parse tree a β-redex is depicted as shown in figure 5.10.

When executing a β-reduction step, all the VARIABLE objects with *name* equal to x in the parse tree of M, are replaced by a copy of the parse tree of term N. We can simply replace all occurrences of x, because we know that the Barendregt-convention holds for the corresponding term. Because of the Barendregt-convention, all occurrences of x in M are bound by the variable declaration in the β-redex.

Note that after the β-reduction step the Barendregt-convention does not have to hold anymore and we have to rename all binding and bound variables in the contractum of the β-redex.

β-equivalence

The λC proof checker implements a simple algorithm to verify whether two terms A and B are β-equivalent. The algorithm reduces both terms A and B to their β-normal form (see subsection 5.3.3). If the corresponding β-normal forms are α-equivalent the terms A and B

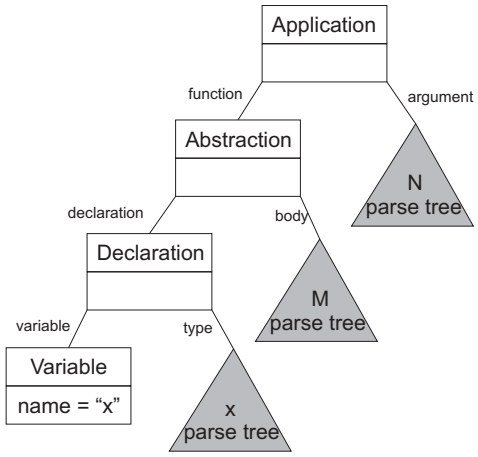

Figure 5.10: Parse of β-redex.

are β-equivalent. If the β-normal forms are not α-equivalent the terms A and B are also not β-equivalent. We can use this approach, since we know that the Church-Rosser Theorem and the Strong Normalizing Theorem hold for λC (see section 3.2.4).

Remark 5.3.6 *The algorithm for β-equivalence is not very efficient. In practice it turns out that extra β-reduction steps are needed to reduce both terms to their β-normal form. It can be that we are able to observe the β-equivalence between the terms sooner.*

In the λD^+ proof check the β-equivalence algorithm is optimized. There we will see that no optimal algorithm can be implemented to determine the β-equivalence between two terms.

Computing the β-normal form

The algorithm that defines the β-equivalence between two terms uses the β-normal form of both terms. To compute the β-normal form of a term T, the system performs β-reduction steps on T as long as T contains a β-redex. Since we know that the Strong Normalization Theorem (see section 3.2.4) holds for λC, only a finite number of β-reduction steps are necessary to compute the β-normal form of T.

The algorithm looks for a leftmost β-redex (see figure 5.10) in T, by performing a top-down scan in the parse tree. If a β-redex is found the corresponding β-reduction step is executed. After the β-reduction step the focus of the algorithm returns to the root of the parse tree, where the algorithm tries to find a new β-redex. This continues till T contains no β-redex anymore. If T has no β-redex anymore, T is in β-normal form and the algorithm stops.

After the execution of a β-reduction step it can occur that T does not satisfy the Barendregt-convention anymore. Take for example the β-redex $(\lambda x : \Pi p : A.A.x\ x\ x2)(\lambda z : A.z)$, where the declarations $A : *$ and $x2 : A$ are added to the context. The result of the β-reduction step is $(\lambda z : A.z)(\lambda z : A.z)x2$. In the result term the Barendregt-convention does not hold, since the variable z is declared twice.

To regain the Barendregt-convention, all variables that are bound by a λ or a Π-term are renamed after each β-reduction step. To rename the variables, we will use the same method as is described in section 5.3.2.

Because only the binding variables of a λ and Π-term are renamed, the system checks before each rename action if the new variable name, for example x_k, is not already declared in the context. If the variable x_k is already declared in the context, the system will check if the name x_{k+1} is still available. This repeats till the algorithm finds a variable name that is not declared in the context. This name will be used to rename the next variable.

Example 5.3.7 *The judgement* $(\lambda z : A.z)(\lambda z : A.z)x2$*, where* $x2$ *is declared in the context is renamed to* $(\lambda x1 : A.x1)(\lambda x3 : A.x3)x2$*.*

Computing the λ-head form

Another notion related to β-reduction is the λ-head form. Although we need this not earlier than in the λD^+ proof checker, we describe it here already, since this section is concerned with β-reduction.

A term in λ-head form has the structure $\lambda x : A.B$. Not every λC term can be reduced to a λ-head form. Take for example the variable z. The λC proof checker contains an algorithm to check if an arbitrary term T has a λ-head form. If T has a λ-head form, the algorithm returns the corresponding λ-head form. Otherwise the algorithm returns false.

To compute the λ-head form of the T, β-reduction steps are executed on T as long as T is a β-redex. If the resulting term has the form $\lambda x : A.B$, the λ-head form of T is found and returned by the algorithm.

When the resulting term is not a λ-head form but an application MN, where MN is not a β-redex, the algorithm goes into recursion for the term M. If M can be reduced to a λ-head form $(\lambda x : A.B)$, we continue the algorithm with the term $(\lambda x : A.B)N$.

If T was not of the form MN or M could not be reduced to a λ-head form, the algorithm return false. Note that in this situation, we are not able to reduce T to a λ-head form.

Just as in computation of the β-normal form the binding and corresponding bound variables in T are renamed after each β-reduction step.

Computing the Π-head form

A related notion is the Π-head normal form, which is used in the typing algorithm, both in the λC and λD^+ proof checker (see section 5.4.7 and section 7.2.5).

A term in Π-head form has the structure $\Pi x : A.B$. Not every λC term can be reduced to a Π-head form. The λC proof checker contains an algorithm to check if an arbitrary term T has a Π-head form. If T can be reduced to a Π-normal form, the algorithm returns the corresponding Π-head form. Otherwise the algorithm returns false.

The algorithm for computing the Π-head form is similar to the algorithm to compute the λ-head form and will therefore not be given.

5.4 The derivability check

In the previous sections of this Chapter we gave a description of the necessary preparation steps of the derivability check. The preparation steps contain the insertion of a judgement, the syntax check, the free variable check and the procedure to ensure the Barendregt-convention. The preparation steps are added as pseudo code to algorithm 3. In the last line of algorithm 3, the function isCorrectJudgement is added. The function isCorrectJudgement checks the derivability of the defined judgement and is described in this section.

Algorithm 3: Preparation steps

```
inputJudgement ← Define input judgement in user interface ;
Create LL(1) parser LL1parser;
if not LL1parser.checkSyntax(inputJudgement, out J) then
    Show message of syntax error ;
    return
else
    J is the root object of the parse tree of inputJudgement
if J contains free variables then
    Show message;
    return
J.ensureBarendregtConvention() ;
if isCorrectJudgement(J) then
    J is derivable with the λC derivation rules.
else
    J is not derivable with the λC derivation rules.
```

5.4.1 The structure of the derivation tree

The derivation rules of λC, described in section 3.2, all have the format depicted in figure 5.11.

$$\frac{P_1, \ldots, P_n}{R} \text{ , with } n \geq 0$$

Figure 5.11: Format derivation rules

The elements P_1,...,P_n and R are judgements. The conclusion R can follow if the premisses $P_1, \ldots, P_n$ hold. The judgement R itself can also be a premiss for another derivation, where one can derive a new judgement. In this way a judgement J is built by the successive use of the derivation rules.

The construction starts with the axioms. An axiom is a derivation rule without premisses. Therefore the conclusion of am axiom is always valid and can be used to construct more complex judgements. The λC language contains one axiom rule, namely the *ax* derivation rule. As a consequence the conclusion of the *ax* derivation rule $\emptyset \vdash * : \Box$ is the starting point of all λC derivations.

The derivation steps of a judgement J, can be stored in a tree structure. Such a data structure is called a *derivation tree*. In the derivation tree every derivation rule is explicitly

labeled with its name, premisses and conclusion. To store both the derivation rules and the judgements in the derivation tree, a derivation tree can contain two types of nodes. The class diagram of the derivation tree is given in figure 5.12.

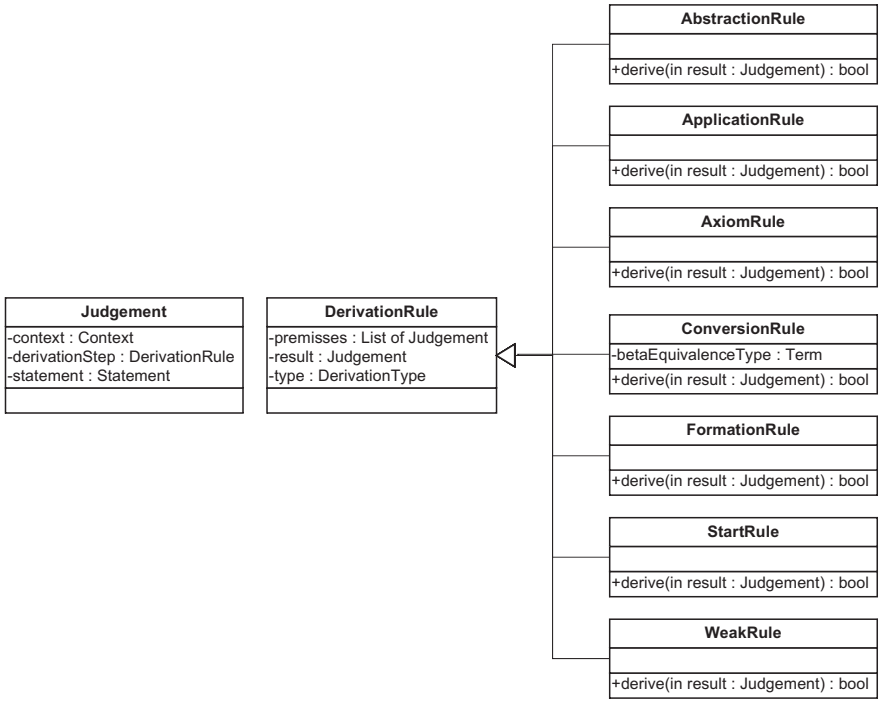

Figure 5.12: Class diagram of derivation tree.

In the class diagram the two types of nodes are depicted by the classes JUDGEMENT and DERIVATIONRULE. The class DERIVATIONRULE is used to represent a derivation rule in the derivation tree. In the class diagram, a class is added for each derivation rule of λC. All the derivation rule classes have the attributes *premisses*, *result* and *type*, since they inherit from the class DERIVATIONRULE. The reason to design a class for each derivation rule is motivated in section 5.4. In this section also a description is given for the function DERIVE. For now we will only concentrate on describing the structure of the derivation tree.

The attribute *premisses* stores the premisses of the corresponding derivation rule. While the attribute *result* stores the conclusion. Next to the attributes *premisses* and *result* the class contains a third attribute called *type*. In the attribute the type of the derivation rule is stored. The type is equal to the name of the corresponding derivation rule.

The class JUDGEMENT is the same class that is used in the parse tree (see section 5.2.2). Objects of the class JUDGEMENT serve as root object in the parse tree of a judgement. The class contains the attribute *derivationStep*. In the attribute the derivation rule is stored from which the judgement is the result.

Example 5.4.1 *A derivation rule with a structure as given in figure 5.11 is translated in the derivation tree as depicted in figure 5.13. Note that the premisses $P_0, \ldots, P_n$ are the result of another derivation rule and the conclusion R can be used as a premiss in another derivation rule. These relations are shown by dashed lines in the figure.*

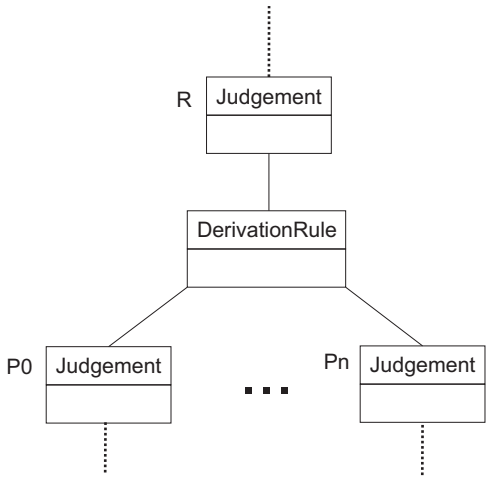

Figure 5.13: Derivation tree of derivation given in figure 5.11.

5.4.2 The implementation of the derivation tree

The algorithm to determine the derivability of a judgement that is implemented in the λC proof checker, tries to generate a derivation tree. In the previous section, we described the structure of a derivation tree. To compute the derivation tree one starts with the *ax* derivation rule. The conclusion of the *ax* derivation rule is used to create more complex judgements by the successive use of the derivation rules. With this approach the derivation tree of judgement J is constructed bottom-up.

This approach can be used when constructing a derivation tree with less than one-hundred nodes. When the derivation tree becomes larger it is almost impossible to find a strategy that finally results in the judgement J. The judgements that will be checked by the λC proof checker, can easily have a derivation tree of more than ten-thousand nodes.

Therefore we are forced to use a different approach when constructing the derivation tree. The approach of the λC proof checker starts with the root instead of the leafs of the derivation tree. From the root node the derivation tree is constructed downwards, ending at the axioms. To give a clear description of the algorithm that constructs the derivation tree, we ignore the *conv* derivation rule. The *conv* derivation rule is described individually in section 5.4.8.

The algorithm that checks the derivability of an input judgement by constructing the derivation tree is called isCorrectJudgement and is given in pseudo-code in algorithm 4. The algorithm isCorrectJudgement checks if the judgement J is derivable with the derivation rules of λC. If judgement J is derivable then the algorithm returns true. If the judgement is not derivable the algorithm returns false.

Algorithm 4: BOOL ISCORRECTJUDGEMENT(JUDGEMENT J)

```
nextDer := getNextDerivationRule(J) ;
if nextDer = 'noDerivation' then
    return true
nextDerivationRule := Create DERIVATIONRULE with type nextDer ;
nextDerivationStep.result := J ;
J.derivationStep := nextDerivationRule ;
if nextDerivationRule.derive(J) then
    return True
else
    return False
```

The first step in the algorithm is to find a suitable derivation rule that eventually can conclude judgement J. A derivation rule is suitable if the structure of the conclusion matches the structure of judgement J. To find the next derivation, the algorithm GETNEXTDERIVATIONRULE is added. The function returns a derivation rule by means of the classes that are given in figure 5.12. If no suitable derivation rule is found for J, then J is not derivable with the derivation rules of λC and the algorithm returns false. If the algorithm is able to find a next derivation rule for J, J is used as the conclusion of the corresponding derivation rule.

For obtaining a conclusion, the derivation rule may need one or more premisses. To create the premisses the function DERIVE is implemented and invoked by the function ISCORRECTJUDGEMENT. Depending on the definition of the derivation rule, JUDGEMENT objects are created and added to the attribute *premisses* of the corresponding derivation rule. The JUDGEMENT objects define the premisses of the derivation rule.

After the creation of the premisses in the function DERIVE, the function DERIVE starts the recursion of the algorithm ISCORRECTJUDGEMENT. In the recursion each premiss is checked on its derivability. When checking the derivability for the premisses, the derivation tree for the input judgement is generated. If all the premisses of the derivation rule are derivable or the derivation rule contains no premiss (*ax* derivation rule) then the conclusion may follow and the algorithm returns true. If it turns out that one or more premisses are not derivable then the definition of the derivation rule is not satisfied and we cannot arrive at the conclusion of the derivation rule. In this situation the algorithm will immediately stop and return false.

The above description of the algorithm ISCORRECTJUDGEMENT is not complete. In section 5.4.3 we describe the function GETNEXTDERIVATIONRULE and in section 5.4.6 the algorithm DERIVE is described.

5.4.3 Finding the next derivation step

As described in section 5.4.2 the next derivation rule is selected by looking at the structure of the input judgement J. The structure of J must match the structure of the conclusion of the derivation rule. If we are not able to find a next derivation rule for J, we must conclude that J is not derivable.

The function that selects that the next derivation rule for J is called GETNEXTDERIVATIONRULE. The JUDGEMENT object of J is the input attribute of the algorithm. The function returns the name of the next derivation rule for J. The structure of a λC judgement is defined

in section 3.2. We recall the definition for clarity.

Definition 5.4.2 *Statement, declaration, context, judgement*

1. *A statement is of the form $M : N$. In such a statement, M is called the subject and N the type.*
2. *A declaration is a statement with a variable as subject.*
3. *A context is a list of declarations with different subjects.*
4. *A judgement has the form $\Gamma \vdash M : N$, with Γ a context and $M : N$ a statement.*

All the components of the judgement are important in finding the next derivation rule. Let us first look at the structure of the conclusions of the λC derivation rules. The conclusions are listed in table 5.4.3.

Derivation rule	Conclusion
ax	$\emptyset \vdash * : \Box$
start	$\Gamma, x : A \vdash x : A$
weak	$\Gamma, x : C \vdash A : B$
form	$\Gamma \vdash \Pi x : A.B : s$
appl	$\Gamma \vdash MN : B$
abst	$\Gamma \vdash \lambda x : A.M : \Pi x : A.B$

Table 5.3: Conclusions of derivation rulesλC.

To describe the algorithm, we use the *start* derivation rule as example. The selection process of all other derivation rules is comparable and will therefore not be given.

The conclusion of the *start* derivation rule has the format $\Gamma, x : A \vdash x : A$. What constraints are specified on the structure of the context and the statement for the conclusion of the *start* rule? First of all, the context of the input judgement cannot be empty, since the context must contain the declaration $x : A$. The second constraint restricts the type of the subject for the input judgement. The subject is equal to a variable and the variable is declared in the last declaration of the context. In the conclusion no restrictions are defined for the structure of the type. The type must only be equal to the type of the last declaration in the context of the judgement. If the input judgement satisfies these constraints the algorithm will choose the *start* derivation rule.

For all other derivation rules except the *weak* derivation ruel, one can define similar constraints. If the input judgement meets those constraints the corresponding derivation rule is returned by the algorithm.

5.4.4 Strengthening of weak derivation rule

There exist judgements for which multiple derivation rules are possible. Take for example the judgement $A : *, y : A \vdash \lambda x : A.x : \Pi x : A.A$. For this judgement one could chose the *weak* and the *abst* derivation rule. For the λC proof checker we want to implement a syntax-directed algorithm. A syntax-directed algorithm is deterministic and generates a unique derivation tree for each judgement.

The reason that multiple derivation rules are possible is because of the *weak* derivation rule. In the *weak* derivation rule no restrictions are defined for the statement of the conclusion, while the restrictions on the context have an overlap with other derivation rules. Therefore it is possible that besides the *weak* derivation rule, also the *start*, *form*, *appl* or *abst* derivation rule is applicable. Note that the *ax* derivation rule is blocked by the restrictions on the context.

To regain the determinism of the algorithm getNextDerivationRule, we strengthen the *weak* derivation rule by adding restrictions on the statement of the conclusion. Only which restrictions are sufficient? It is obvious that the new restrictions cannot have an overlap with the restrictions of the derivation rules *abst*, *appl*, *form* and *start*. If the restriction would overlap then there still exist judgements where multiple derivation rules are possible.

The restrictions on the statement of the *form*, *appl* and *abst* derivation rule are mainly concerned with the subject. The subject must start with a typebinder or be an application or an abstraction. The first restriction that we define for the conclusion of the *weak* derivation rule, is that the subject cannot contain start with typebinder or be an application or an abstraction.

The restrictions for the *start* rule are more complex than the restrictions of the other derivation rules. This is because the restrictions of the *start* derivation rule contain a fixed relation between the context and the statement. The subject of the statement must be equal to a variable and this variable is declared in the last declaration of the context. If in the last declaration another variable is introduced then the *start* derivation rule would not be applicable.

Summarizing the restrictions, we can conclude that the conclusion of the *weak* derivation rule can only contain a statement of the form $* : \Box$ or $y : B$, where y is not introduced in the last declaration of the context. If we add the restrictions to the existing derivation rule, we can define a new *weak* derivation rule. The new *weak* derivation rules are given in figure 5.14.

$$weak_1 \quad \frac{\Gamma \vdash * : \Box \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash * : \Box} \text{ if } x \notin \Gamma$$

$$weak_2 \quad \frac{\Gamma \vdash y : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash y : B} \text{ if } x \notin \Gamma \text{ and } x \neq y$$

Figure 5.14: Strengthened *weak* derivation rule.

When introducing a new *weak* derivation rule, we must prove that the resulting derivation rules result in a calculus with the same set of derivable judgements. The proof of this is given in [Ben] and will not be included in the master thesis.

5.4.5 The algorithm getNextDerivationRule

With the new strengthened *weak* derivation rule, we can give the complete getNextDerivationRule algorithm. The algorithm returns at most one derivation rule per judgement. If the algorithm is not able to find a suitable derivation rule, the algorithm will return 'noDerivation'. In algorithm 5 the pseudo code of getNextDerivationRule is given. In the pseudo

code the parse tree classes are reflected by a textual value instead of the objects.

Algorithm 5: getNextDerivationRule(Judgement J)

```
switch Statement of J do
    case ∗ : □
        if Context of J = ∅ then
            return ax
        else
            return weak
    case x : A
        if Context of J ≠ ∅ then
            lastDeclaration := Last declaration of context of J ;
            switch lastDeclaration do
                case x : A
                    return start
                case y : B
                    return weak
    case Πx : A.B : s
        return form
    case MN : B
        return appl
    case λx : A.M : Πx : A.B
        return abst
return noDerivation
```

5.4.6 Create premisses for derivation rule

Apart from finding the next derivation rule, the algorithm isCorrectJudgement had a second problem, namely: how can we create premisses per derivation rule and check if they are derivable with the λC derivation rules?

For this, we added the function derive(Judgement *result*) to the algorithm isCorrectJudgement. Each derivation rule has a different definition and therefore each derivation rule has its own derive function (see the class diagram 5.12). The input attribute *result* is defined as the conclusion of the corresponding derivation rule. Based on *result* and the definition of the derivation rule the premisses are created and added to the derivation tree. When the premisses are added, the function isCorrectJudgement is called recursively for each premiss by the function derive. In the recursion the premisses are checked on their derivability. If all premisses are derivable, the function derive will return true. If it turns out that a premiss is not derivable then the function derive will immediately stop and return false. This section describes the algorithm derive.

To describe the algorithm we shall use the *appl* derivation rule as an example. The definition of the *appl* derivation rule is given in figure 5.4.6.

$$\textit{(appl)} \quad \frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

Figure 5.15: *Appl* derivation rule.

In the definition of the *appl* derivation rule two premisses appear. We shall denote $\Gamma \vdash M : \Pi x : A.B$ as premiss 1 and $\Gamma \vdash N : A$ as premiss 2. Between the premisses and the conclusion of the derivation rule, relations are defined. When creating the premisses, one must maintain those relations. The *appl* derivation rule contains the following relations:

- The context of premiss 1 and 2 is equal to the context of the conclusion.
- The subject of premiss 1 is equal to the first component (M) of the statement of the conclusion.
- The subject of premiss 2 is equal to the second component (N) of the statement of the conclusion.

Since the conclusion is added as input attribute to the algorithm, the context and the subject of the statement can simply be copied for both premisses. On the other hand the type of both premisses cannot be copied, since they contain subterms that do not occur in the conclusion of the *appl* derivation rule.

If we do not concentrate on the *appl* derivation rule only, we can see that all unknown terms in the premisses of the derivation rules occur in the types. To find the unknown term, the λC proof checker contains a typing algorithm called getType. The typing algorithm finds, given a context Γ and a term M, the type of M. If the typing algorithm is not able to find a type for M, the algorithm will return false. To complete the description of the *appl* derivation rule, we shall give no further explanation of the typing algorithm in this section. The description of the typing algorithm is given in section 5.4.7. For now, we just assume that there exists a typing algorithm getType.

With the typing algorithm, we are able to obtain the types for both premisses of the *appl* derivation rule. It is sufficient for both premisses to retrieve the type of the subject. If the typing algorithm is not able to find a type for one of the premisses, the corresponding premiss cannot be added to the derivation tree. In section 5.4.7 is described that the typing algorithm is not able to return a type for a term M, if M is not derivable. Since the subject is not derivable, we can immediately conclude that the complete premiss and therefore also the conclusion of the derivation rule is not derivable. In this situation the algorithm returns false. If the typing algorithm can find both types, a Judgement object is created for both premisses and added to the derivation tree.

It can happen that the type retrieved by the typing algorithm contains variable names that are already used in the context or the subject of the premiss. In this way the Barendregt-convention does not hold for the premiss. To maintain the Barendregt-convention for each judgement in the derivation tree all the variable of the premisses are renames as described in section 5.3.2.

In the last step of the algorithm the function isCorrectJudgement is called recursively for all premisses of the derivation rule. In the recursion the derivability of each premiss is checked.

The pseudo code of the derive algorithm for the *appl* derivation rule is given in algorithm 6. The pseudo code of the other derivation rules is comparable and will therefore not be given in the master thesis.

Algorithm 6: DERIVE(Judgement *result*)

```
Create JUDGEMENT object prem1 and prem2 ;
;
prem1.context := Copy of J.context ;
prem1.statement.subject := Copy of function of J.statement.subject ;
if GETTYPE(prem1.context, prem1.statement.subject, out type1) == false then
    return false
else
    prem1.statement.type := type1
Ensure Barendregt-convention prem1 ;
Add prem1 to premisses;
;
prem2.context := Copy of J.context ;
prem2.statement.subject := Copy of argument of J.statement.subject ;
if GETTYPE(prem2.context, prem2.statement.subject, out type2) == false then
    return false
else
    prem2.statement.type := type2
Ensure Barendregt-convention prem2 ;
Add prem2 to premisses;
;
for each premiss in premisses do
    if not ISCORRECTJUDGEMENT(premiss) then
        return false
;
return true
```

5.4.7 Typing algorithm

The typing algorithm getType of the λC proof checker finds, given a context Γ and a term M, the type of M. If the typing algorithm is not able to find a suitable type for M, the algorithm returns false. This occurs if the context Γ and/or the term M is not derivable. If the algorithm can find a type, say A, it return true and the following holds:

$$\text{getType}(\Gamma, M, A) \quad \Rightarrow \quad \Gamma \vdash M : A$$

The typing algorithm implemented in the λC proof checker is based on the typing algorithm that is described in [Geu2] for the λ-calculus λP. The system λC is an extension of λP. The typing algorithm implemented in the λC proof checker is given in pseudo code in algorithm 7 and 8.

The typing algorithm getType plays two roles, namely to find the type A and to check if Γ and M are well-formed. The algorithm can only find the type A if both Γ and M are legal.

The well-formedness check of Γ is executed by the algorithm OK. The algorithm checks for each declaration $x : A$ in Γ, if A is a valid type. A type A is valid, if the type of A is equal to $*$ or $\Box$. The derivability check of M is nested in the algorithm getType by means of the if-conditions. Note that the if-conditions have a big overlap with the derivation rules of λC.

One may consider whether it is necessary to check the derivability of Γ and M in the typing algorithm. The typing algorithm is used to construct the premisses of the derivation rules in the derive function. After the creation, the premisses are recursively checked for their derivability. Since both Γ and M are elements of the premisses, their derivability is checked twice. So maybe we can eliminate one of the checks.

The derivability check of M cannot be skipped entirely, since most if-conditions in getType contain a recursive function call of getType, that is necessary to find the type of M. Take for example the if-conditions in the situation where M equals KL. Both the first and second if-condition are necessary to find the type of KL.

On the other hand it is possible to skip the second if-condition in the situation where M equals $\lambda x : A.K$. The second if-condition is only added to check if term $\Pi x : A.B$ is a valid type. This would also be checked in the recursion of isCorrectJudgement.

To implement one method in the typing algorithm, independently of the structure of M, we have chosen to maintain the derivability check of M. The algorithm assumes that the context Γ is correct and therefore the algorithm OK is superfluous. The pseudo code of the renewed typing algorithm can be found by removing all function calls of OK from the algorithms 7 and 8.

Algorithm 7: GETTYPE(Context Γ, Term M, out Term *type*) : bool

```
switch M do
    case x
        if OK(Γ) and x : A ∈ Γ then
            type := A;
            return true
        else
            return false
    case ∗
        if OK(Γ) then
            type := □ ;
            return true
        else
            return false
    case KL
        if GETTYPE(Γ, K, C) and GETTYPE(Γ, L, D) then
            if C →_β Πx : A.B and D =_β B then
                type := B[x := L] ;
                return true
            else
                return false
        else
            return false
    case λx : A.K
        if GETTYPE((Γ, x : A), K, B) then
            if GETTYPE(Γ, Πx : A.B, termtype) and termtype ∈ {∗, □} then
                type := Πx : A.B ;
                return true
            else
                return false
        else
            return false
    case Πx : A.B
        if GETTYPE(Γ, A, typedec) and GETTYPE((Γ, x : A), B, typebody) then
            if typedec ∈ {∗, □} and typebody ∈ {∗, □} then
                type := typebody ;
                return true
            else
                return false
        else
            return false
    otherwise
        return false
```

Algorithm 8: OK(Context Γ)

```
if Γ = ∅ then
    return true
else
    Note that Γ is not empty and Γ is of form Γ', x : A ;
    if TYPE(Γ, A) ∈ {∗, □} then
        return TYPE(Γ, A)
    else
        return false
```

5.4.8 Conversion derivation rule

The algorithm isCorrectJudgement is described in the previous subsections. In the description, we have excludes the *conv* derivation rule. In this section we describe the adjustments that are necessary to implement the *conv* derivation rule into the derivability check.

With the *conv* derivation rule, one is able to replace the type B of judgement J by another λC term B'. One can only replace the terms if B' is a valid type and when B' is β-equivalent to B. The definition of the *conv* derivation rule is given in figure 5.16.

$$\textit{(conv)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}, \text{ if } B =_\beta B'$$

Figure 5.16: *Conv* derivation rule.

In section 5.4.5 we have seen that the algorithm getNextDerivationRule returns at most one derivation rule for each unique judgement. If we look at the structure of the conclusion of the *conv* derivation rule, we can see that no restrictions are added. To execute the *conv* derivation rule, only a derivable λC term B is necessary, where B is β-equivalent to the type B' of the conclusion. Such a term is easily found for each term B', if we take for B for example the term $(\lambda C : s.C)B'$. As a result the *conv* derivation rule is always available next to the other derivation rules.

The goal of the derivability check in the λC proof checker is to generate a derivation tree for the given judgement J. It is desirable for the performance to choose the derivation rules in such an order that the height of the derivation tree is minimized. Therefore it is forbidden to add the *conv* derivation rule to the derivation tree without creating any 'progress'. The *conv* derivation rule is only executed if no other derivation rule is applicable and the use of the *conv* derivation rule leads to 'progress' in the derivability check. What we mean by 'progress' will become clear further on in this section.

The judgements that are only derivable when the *conv* derivation rule is included in the system, can be divided into three groups:

- $\Gamma, x : A \vdash x : T$. In the judgement T is a derivable λC-term that is not α-equivalent but β-equivalent to A.
- $\Gamma \vdash MN : T$. In the judgement the type of M is equal to C and the type of N is equal to D, where C has a Π-head form $\Pi x : A.B$ and D is β-equivalent to A. The use of the *conv* derivation rule is necessary if T is not α-equivalent but β-equivalent to $B[x := N]$.

- $\Gamma \vdash \lambda x : A.M : T$. In the judgement T is a derivable λC-term that is not α-equivalent but β-equivalent to $\Pi x : A.B$, where B is the type of M.

In the above examples we can see that by choosing the right type in the *conv* derivation rule, the rule is always proceeded by the *start*, *appl* or *abst* derivation rule. For example if we choose in the first situation A as the new type for x, the *conv* derivation rule is preceded by the *start* rule. This is what we mean by 'progress'. We only execute the *conv* derivation rule, if that helps to obtain the result by means of another derivation rule.

In the first situation it is easy to retrieve the new type, since the type is added in the last declaration of the context. In the other two situations it is not that simple, since the type is not a subterm of the conclusion.

So how are we able to find the new type, if we desire that the *conv* derivation rule is followed by the *abst* or *appl* derivation rule? In section 5.4.7 we have described the typing algorithm of the λC proof checker. The typing algorithm finds, given a context Γ and a term M, the type A of term M. In the description is mentioned that the if-conditions in the typing algorithm have an strong overlap with the derivation rules of λC. Because of the overlap the return type A has a structure such that the judgement $\Gamma \vdash M : A$ is derivable by the *ax*, *start*, *weak*, *form*, *appl* or *abst* derivation rule. By using the typing algorithm we can find the correct type in the other two situations.

Since we aim at only one approach in the λC proof checker, the typing algorithm is also used to find the new type in the first situation. If we know the new type, we only check if the new type is α-equivalent or β-equivalent to the existing type of the judgement. If the types are α-equivalent the *start*, *appl* or *abst* derivation rule is chosen. If the types are only β-equivalent the *conv* derivarion is used as the next derivation rule in the derivation tree. If the types are not α-equivalent and β-equivalent we can conclude that the judgement is not derivable and 'noDerivation' is returned.

To add the *conv* derivation rule to the λC proof checker, we add the above adjustments to the algorithm GETNEXTDERIVATIONRULE. The pseudo code of the new algorithm is given in algorithm 9. Note that the algorithm contains one extra output attribute *newType*. In the attribute the new type needed for the *conv* derivation rule is stored. If another derivation rule is returned, the attribute *newType* is ignored.

The function GETNEXTDERIVATIONRULE is invoked by the function ISCORRECTJUDGEMENT. If the function GETNEXTDERIVATIONRULE returns the *conv* derivation rule, a CONVERSIONRULE object is created (see figure 5.12). The new type is stored in the attribute *betaEquivalenceType*. When the *conv* rule is executed by the function DERIVE the attribute *betaEquivalenceType* is used to create the premisses.

The adjustments in the algorithm ISCORRECTJUDGEMENT and DERIVE are so small, that we do not give the pseudo code of the new algorithms.

Now we have completed the description of the implementation of all the derivation rules of λC, the proof checker is complete. For all derivable λC judgments the function ISCORRECTJUDGEMENT returns true and for the underivable judgements the function returns false.

Algorithm 9: getNextDerivationRule(Judgement J, out Term newType)

```
/* J can be written as Γ ⊢ K : L                                    */
switch K do
    case ∗
        if L = □ then
            if Γ = ∅ then
                return ax
            else return weak
    case x
        if Γ ≠ ∅ then
            lastDeclaration := Last declaration of Γ ;
            switch Variable of lastDeclaration do
                case x
                    if getType(Γ, x, out newType) then
                        if newType is α-equivalent to L then
                            return start
                        else if newType is β-equivalent to L then
                            return conv
                otherwise
                    return weak
    case Πx : A.B
        if L is s then
            return form
    case MN
        if getType(Γ, MN, out newType) then
            if newType is α-equivalent to L then
                return appl
            else if newType is β-equivalent to L then
                return conv
    case λx : A.M
        if getType(Γ, x : A.M, out newType) then
            if newType is α-equivalent to L then
                return abst
            else if newType is β-equivalent to L then
                return conv
return noDerivation
```

5.4.9 Sharing

During the derivability check of a given judgement J, the derivation tree of J is constructed. The algorithm starts by adding a JUDGEMENT object for J to the derivation tree. In the rest of the algorithm the derivation tree is constructed from the root till all leafs contain an AXIOMRULE object. In each step the next derivation rule for J is chosen by inspection of the form of J. Based on the definition of the corresponding derivation rule, premisses are added to the derivation tree. The premisses are checked recursively on their derivability.

The algorithm does not check if a new premiss has already been added in another position in the derivation tree. Therefore it can occur that the same judgement is added multiple times to the derivation tree. The algorithm ISCORRECTJUDGEMENT checks the derivability for each new judgement, while one derivability check for each unique judgement would be sufficient. The extra work is not good for the performance of the λC proof checker.

As a consequence of the multiple derivability checks, the derivation tree is unnecessarily large. For each judgement K that is added x times to the derivation tree of J, not only $x-1$ extra JUDGEMENT objects are added. But the complete derivation tree of K occurs x times as a sub-tree in the derivation tree of J.

To resolve this we will use sharing in the derivation tree. With sharing we make sure that each unique judgement is checked at most once in the derivability check. To implement sharing in the derivability check a unique JUDGEMENT object can be applied as a premiss in multiple derivation rules. In this way we will decrease the size of a derivation tree.

We also add an extra reference attribute *calculatedJudgements* to the function DERIVE of all distinct derivation rules and to the function ISCORRECTJUDGEMENT. The new attribute is a HASHTABLE and stores all judgements in the derivability check that are evaluated as derivable. A judgement is derivable if the function ISCORRECTJUDGEMENT returns true for the corresponding judgement. A judgement is therefore added to *calculatedJudgements* after the function call ISCORRECTJUDGEMENT in DERIVE.

Before the function DERIVE checks the premisses one-by-one on their derivability, it checks if the premisses have already been added to *calculatedJudgements*. If a premiss is an element of *calculatedJudgements*, it is replaced by the corresponding judgement in *calculatedJudgments*. When the premiss is not added to *calculatedJudgements* it is unknown if the premiss is derivable and the derivability check is performed.

Each object in a HASHTABLE is identified by a key value. The key value of each judgement in *calculatedJudgements* must be defined in such a way that it is easy to retrieve the corresponding judgment from the HASHTABLE. The key of each judgement is the textual representation of the parse tree. The textual value is comparable to the judgement representation in the output form (see section 5.1.2). The only difference is that each VARIABLE object is depicted by the attribute *name* instead of the attribute *externalName* (see figure 5.9). The advantage is that two α-equivalent judgements are represented by the same value, since we rename all variables in the judgement (to ensure the Barendregt-convention) with the same method. Therefore all the α-equivalent judgements are shared in the derivation tree.

5.4.10 Input eye component

In the global description of the λC proof checker (see section 4.2) we saw that the user is able to verify the correctness of the derivability check by the input of the eye component. The input of the eye component is a 'trace' of internal steps needed to execute the derivability check.

In the description of the derivability check we have seen that every recursive step has the same pattern. First the function GETNEXTDERIVATIONRULE is executed to find the next derivation rule for a judgement J. And second the algorithm DERIVE is executed, where based on J and the definition of the selected derivation rule premisses are created.

We choose to store the internal steps of the algorithms GETNEXTDERIVATIONRULE and DERIVE per judgement in the derivation tree. For this the class JUDGEMENT is extended and a data class ACTION is created. The class diagram is given in figure 5.17.

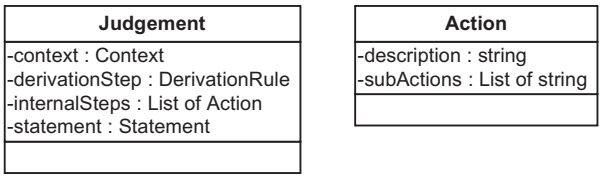

Figure 5.17: Class diagram eye component.

The ACTION class does not store one internal step. The class is more abstract and models a task. For every task one ore more internal steps are needed. The internal steps are stored in the attribute *subActions*. And the attribute *description* is used to store a description of the task.

For each judgement the ACTION objects are stored in the attribute *internalSteps*. We distinguish the following tasks:

- *Find next derivation.* This task stores the internal steps of the algorithm GETNEXTDERIVATIONRULE.
- *Create premiss.* For each new premiss an ACTION is created. In the ACTION object all the internal steps are stored which are necessary to create the corresponding premiss in the function DERIVE.

Chapter 6

Planning λD^+ proof checker

6.1 Global overview

The second proof checker is implemented for the λ-calculus λD^+. The λD^+ proof checker is an extended version of the λC proof checker, with some additional user friendly features. For the λD^+ proof checker a user friendly input language, an optimized $\beta\delta$-equivalence algorithm and a library of legal axioms and definitions is designed and implemented.

The structure of the λD^+ proof checker has the same three components as the λC proof checker: the user interface component, the kernel component and the eye component. The user interface component is adjusted such that one can enter an input judgment using the *input assistant* and the *library*.

The input assistant is a compiler that is added as a subcomponent to the user interface. The input assistant implements the new input language. The user enters a judgement with the new input language in the user interface then the input assistant verifies the grammar and translates the input to the existing input language. This translation is used as input for the kernel component.

With the library the user can store legal axioms and definitions. We can reuse the legal axioms and definitions in the input judgement. In the derivability check the library axioms and definitions are not checked on their derivability. This increases the performance of the proof checker.

The structure of the λD^+ proof checker is depicted in figure 6.1.

6.2 Functional specification

6.2.1 User interface

Requirement 6.2.1 *Requirements of λC proof checker.*
The following requirements of the λC proof checker for the user interface component also apply to the λD^+ proof checker:

Requirement 4.3.1, Create judgement.
Requirement 4.3.3, Check derivability of judgement.
Requirement 4.3.4, Show output of kernel component.
Requirement 4.3.6, Show incorrect judgement derivation.

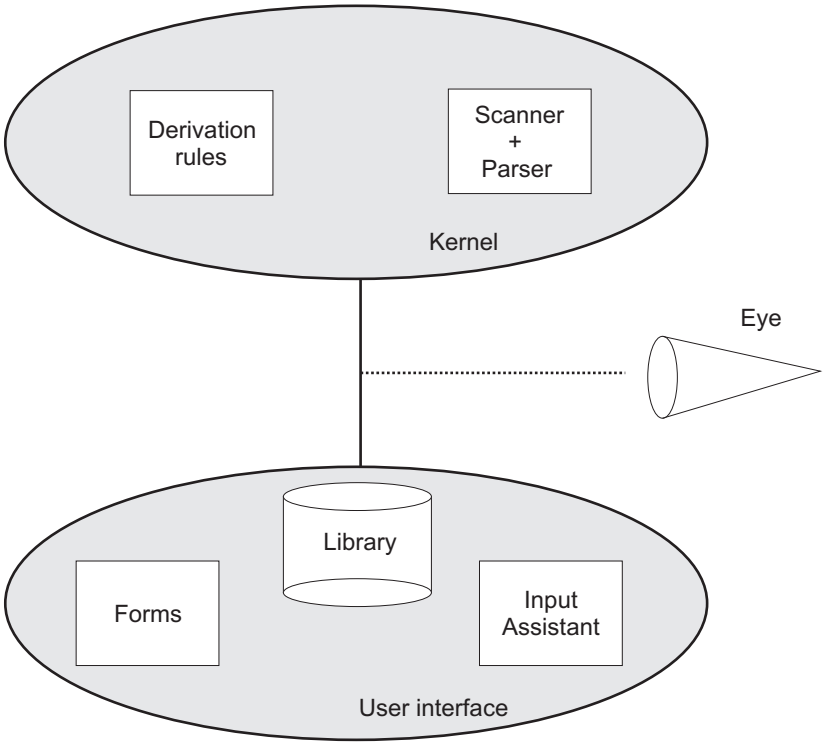

Figure 6.1: Overview λD^+ proof checker.

Requirement 6.2.2 *Grammar of input judgement.*
The judgement must satisfy the following grammar.

$$
\begin{array}{ll}
\textit{Judgement} & ::= \textit{Environment} \ ; \ \textit{Context} \vdash \textit{Statement} \\
\textit{Environment} & ::= \emptyset \mid \textit{DefinitionSuper} \mid \textit{Environment, Environment} \\
\textit{DefinitionSuper} & ::= \textit{Axiom} \mid \textit{Definition} \\
\textit{Definition} & ::= \textit{Context} \ \rhd \ \textit{Constant} \ := \ \textit{Statement} \\
\textit{Axiom} & ::= \textit{Context} \ \rhd \ \textit{Constant} \ := \ \perp\!\!\!\perp \ : \ \textit{Term} \\
\textit{Context} & ::= \emptyset \mid \textit{Declaration} \mid \textit{Context, Context} \\
\textit{Declaration} & ::= \textit{Variable} : \textit{Term} \\
\textit{Statement} & ::= \textit{Term} : \textit{Term} \\
\textit{Term} & ::= \textit{Variable} \mid \square \mid * \mid \textit{(TermTerm)} \mid \lambda \ \textit{Declaration} \ . \ \textit{Term} \mid \\
 & \quad \Pi \ \textit{Declaration.} \ \textit{Term} \mid \textit{Constant} \\
\textit{Variable} & ::= [a \ldots z]^+, \textit{ where } {}^+ = \textit{one or more} \\
\textit{Constant} & ::= [a \ldots z]^+ \ (\ \textit{Parameters} \ ) \mid [a \ldots z]^+ \ (\ ) \\
\textit{Parameters} & ::= \textit{Term} \mid \textit{Parameters,Parameters}
\end{array}
$$

The grammar above has four additional constraints:

1. *In the* Axiom *and* Definition *tag: the number of variables declared in* Context *must be equal to the number of parameters in* Constant.

2. *In the* Axiom *and* Definition *tag: the sequence of the variables in* Context *must be equal to the list of parameters in* Constant.

3. *In the* Axiom *and* Definition *tag: the parameter list of* Constant *can only contain* Variable *objects.*

4. Constant *objects cannot have the same name as* Variable *objects.*

Requirement 6.2.3 *Show correct judgement derivation.*
The user interface is able to show the complete derivation tree of the input judgement. The complete derivation is given in list or flag notation. An example of a derivation in list notation is given in figure 6.2. In figure 6.3 the same example is given in flag notation.

(1)	$\emptyset;\emptyset \vdash * : \Box$	(ax)
(2)	$\emptyset;\alpha : * \vdash \alpha : *$	(start (1))
(3)	$\emptyset;\alpha : *, x : \alpha \vdash x : \alpha$	(start (2))
(4)	$\emptyset;\alpha : *, x : \alpha \vdash \alpha : *$	(weak (2), (2))
(5)	$\emptyset;\alpha : * \vdash \Pi x : \alpha.\alpha : *$	(form (2), (4))
(6)	$\emptyset;\alpha : * \vdash \lambda x : \alpha.x : \Pi x : \alpha.\alpha$	(abst (3), (5))

Figure 6.2: Small example of derivation in list notation

(1)	$\boxed{\alpha : *}$	
(2)	$\quad\boxed{x : \alpha}$	
(3)	$\quad x : \alpha$	(start(1))
(4)	$\lambda x : \alpha.x : \Pi x : \alpha.\alpha$	(abst(2), (3))

Figure 6.3: Small example of derivation in flag notation

Requirement 6.2.4 *Maintain library of definitions.*
The proof checker is able to create and maintain a library of axioms and definitions.

Requirement 6.2.5 *Unique names of axiom and/or definition in library.*
All the constants that are added to the library must have new unique name.

Requirement 6.2.6 *Use axiom or definition from library in input judgement.*
The user is able to use an axiom or a definition, that is incorporated in the library of definitions, to construct an input judgement (see requirement 4.3.1). When checking the derivability for the input judgement, the axioms and the definition from the library are not dealt with. The proof checker assumes that these definitions are correct.

Requirement 6.2.7 *Add axiom to library.*
The user is able to add a new axiom to the library. Only axioms that are proven to be derivable

can be added to the library. To check whether the constant is derivable, the axiom is offered as input to the kernel component (see subsection 4.3.2). In the kernel component the actual computation takes place.

Requirement 6.2.8 *Add definition to library.*
The user is able to add a new definition to the library. Only definitions that are proven to be derivable can be added to the library. To check whether the constant is derivable, the definition is offered as input to the kernel component (see subsection 4.3.2). In the kernel component the actual computation takes place.

Requirement 6.2.9 *Remove axiom from library.*
The user is able to remove an axiom from the library.

Requirement 6.2.10 *Remove definition from library.*
The user is able to remove a definition from the library.

Requirement 6.2.11 *Unable to remove direct dependency.*
The user is unable to remove an axiom or a definition D, if the library contains another axiom or definition D' where D' directly depends on D.

Requirement 6.2.12 *Overview of axioms and definitions of library.*
The user is able to generate an overview of axioms and definitions of the library.

6.2.2 Kernel component

Requirement 6.2.13 *Requirements of λC proof checker.*
The following requirements of the λC proof checker for the kernel component also apply to the λD^+ proof checker:

Requirement 4.3.7, Retrieve input judgement.
Requirement 4.3.9, Parse syntactically correct input judgement.
Requirement 4.3.10, Notify syntactically incorrect input judgement.
Requirement 4.3.11, Ensure Barendregt-convention.
Requirement 4.3.12, No free variables in judgement.
Requirement 4.3.13, Check derivability.
Requirement 4.3.14, Check each judgement on its derivability at most once.
Requirement 4.3.16, α-equivalence.
Requirement 4.3.17, Perform α-conversion.
Requirement 4.3.18, Perform one-step β-reduction.
Requirement 4.3.20, Store derivation tree.
Requirement 4.3.21, Store incorrect judgements.
Requirement 4.3.22, Internal steps as input for eye component.

Requirement 6.2.14 *Check syntax of input judgement.*
The kernel is able to check if an input judgement (see requirement 4.3.7) is syntactically correct. The judgement is syntactically correct if it can be constructed with the grammar described in requirement 6.2.2. If this is not the case then the judgement is syntactically incorrect.

Requirement 6.2.15 *All the constants in the input judgement are defined in the environment.*
All the constants that are added to the input judgement, are added to the environment of the judgement.

Requirement 6.2.16 *No check of axioms and definitions from library in derivability check of input judgement.*
The definitions that are added from the library to the input judgement are not checked on their derivability. The proof checker assumes that these definitions are correct, because only derivable definitions can be added to the library (see requirement 6.2.8).

Requirement 6.2.17 *Check definitions not from the library in correctness check of input judgement.*
The definitions occurring in the environment of the input judgement and which are not part of the library, are checked on their derivability.

Requirement 6.2.18 *Derivation rules.*
In the kernel all the derivation rules (see section 3.3) of λD^+ are included. This means that all judgements of the grammar described in requirement 6.2.2 can be checked on its correctness.

Requirement 6.2.19 *Perform one-step δ-reduction.*
The kernel is able to perform an one-step δ-reduction.

Requirement 6.2.20 *$\beta\delta$-equivalence.*
The kernel component is able to verify if two terms are $\beta\delta$-equivalent.

6.2.3 Eye component

Requirement 6.2.21 *Requirements of λC proof checker.*
The following requirements of the λC proof checker for the eye component also apply to the λD^+ proof checker:

Requirement 4.3.23, Retrieve input from kernel component.
Requirement 4.3.24, Show derivation steps.

6.3 Iterative schedule

6.3.1 First iteration

In the first iteration the λC proof checker is updated to λD^+. After the first iteration the proof checker is able to verify that a given judgement is derivable by the λD^+ derivation rules. The update can be divided into:

1. Extend input language of λC proof language, such that λD^+ judgements can be defined.
2. Update parser of λC proof checker for extended input language.
3. Update typing algorithm. The extended typing algorithm is able to find a type for a constant.

4. Update derivability check:
   (a) Implement the *env-weak*, *env-weak*$^{\perp\!\!\!\perp}$, *inst* and *inst*$^{\perp\!\!\!\perp}$ derivation rule.
   (b) Update λC derivation rule for use of λD^+.
5. Extend the algorithm to verify if two terms are β-equivalent, to $\beta\delta$-equivalence. In this iteration only a simple algorithm is implemented to check whether two terms are $\beta\delta$-equivalent. The algorithm finds first the $\beta\delta$-normal forms of the terms and then checks if these normal forms are α-equivalent.

For the first iteration the following requirements are implemented:

User interface component:
- Requirement 4.3.1, Create Judgement.
- Requirement 4.3.3, Check derivability of judgement.
- Requirement 4.3.4, Show output of kernel component (partly).
- Requirement 4.3.6, Show incorrect judgement derivation.
- Requirement 6.2.2, Grammar of input judgement.
- Requirement 6.2.3, Show correct judgement derivation (partly).

Kernel component:
- Requirement 4.3.7, Retrieve input judgement.
- Requirement 4.3.9, Parse syntactically correct input statement.
- Requirement 4.3.10, Notify syntactically incorrect input judgement.
- Requirement 4.3.11, Ensure Barendregt-convention.
- Requirement 4.3.12, No free variables in judgement.
- Requirement 4.3.13, Check derivability.
- Requirement 4.3.14, Check each judgement on its derivability at most once.
- Requirement 4.3.16, α-equivalence.
- Requirement 4.3.17, Perform α-conversion.
- Requirement 4.3.18, Perform one-step β-reduction.
- Requirement 4.3.20, Store derivation tree.
- Requirement 4.3.21, Store incorrect judgements.
- Requirement 4.3.22, Internal steps as input for eye component.
- Requirement 6.2.14, Check syntax of input judgement.
- Requirement 6.2.15, All the constants in the input judgement are defined in the environment.
- Requirement 6.2.17, Check definitions not from the library in correctness check of input judgement.
- Requirement 6.2.18, Derivation rules.
- Requirement 6.2.20, $\beta\delta$-equivalence.

Eye component:
- Requirement 4.3.23, Retrieve input from kernel component.
- Requirement 4.3.24, Show derivation steps.

6.3.2 Second iteration

In the section iteration a new external language is created for the λD^+ proof checker. The current input language remains the input language for the kernel of the λD^+ proof checker.

In this iteration an external translation module is implemented, to convert the new language into the current language. The user is able to enter input judgements in the new language. This input judgement is converted to the current input language, which is used as input for the kernel component. In this way no extensions are necessary for the kernel component. The new input language must have the following properties:

1. The language is smaller then the current input language.
2. The language is user friendly.

For the second iteration the following requirements are implemented:

User interface component:
Requirement 4.3.1, Create judgement.
Requirement 6.2.2, Grammar of input judgement.

6.3.3 Third iteration

In the third iteration we implement a library of axioms and definitions. In the library, axioms and definitions can be added that are proven to be derivable. The involved constants can be used to define a new input judgement. If the input judgement is checked on its derivability, the definitions from the library are skipped. The proof checker just assumes that these definitions are derivable. In this way the derivability check has a better performance and the derivation tree decreases in size.

For the third iteration the following requirements are implemented:

User interface component:
Requirement 6.2.4, Maintain library of definitions.
Requirement 6.2.5, Unique names of axiom and/or definition in library.
Requirement 6.2.6, Use axiom or definition from library in input judgement.
Requirement 6.2.7, Add axiom to library.
Requirement 6.2.8, Add definition to library.
Requirement 6.2.9, Remove axiom from library.
Requirement 6.2.10, Remove definition from library.
Requirement 6.2.11, Unable to remove direct dependency.
Requirement 6.2.12, Overview of axioms and definitions of library.
Kernel component:
Requirement 6.2.16, No check of axioms and definitions from library in derivability check of input judgement.

6.3.4 Fourth iteration

The fourth iteration optimizes the $\beta\delta$-equivalence algorithm. In the current version of the λD^+ proof checker, a simple but inefficient algorithm is implemented. The current algorithm transforms the terms to its $\beta\delta$-normal form and then checks if the $\beta\delta$-normal forms are α-equivalent. The new algorithm must increase the time performance of the λD^+ proof checker,

by decreasing the number of $\beta\delta$-reduction steps.

For the third iteration the following requirements are implemented:

Kernel component:
Requirement 6.2.19, Perform one-step δ-reduction.
Requirement 6.2.20, $\beta\delta$-equivalence.

6.3.5 Fifth iteration

In the fifth iteration the flag style report is implemented. Till now the user is only able to report the derivation tree in list notation.

For the fifth iteration the following requirements are implemented:

User interface:
Requirement 4.3.4, Show output of kernel component.
Requirement 6.2.3, Show correct judgement derivation.

Chapter 7

λD^+ proof checker

7.1 Main form λD^+

In this chapter we give a description of the implementation of the λD^+ proof checker. The design of the new λD^+ proof checker is very similar to the design of the λC proof checker. Because of this we will describe the λD^+ proof checker by another approach. In the description we concentrate on the differences in the implementation. Each section describes one iteration.

If we look at the user interface of the λD^+ proof checker, we reuse the output form (see section 5.1.2) and the eye component form (see section 5.1.3) and we extend the main form of the λC proof checker. The main form is extended such that one can enter a judgement with the new input language (see section 7.3) and we can start the library module of the λD^+ proof checker (see section 7.4). The extended main form is given in figure 7.1.

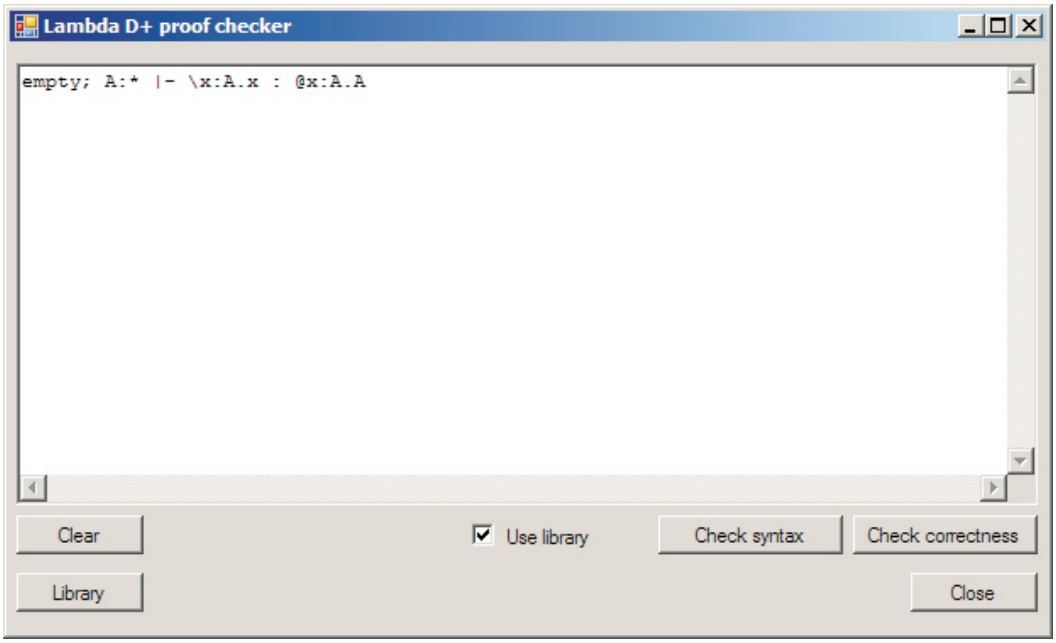

Figure 7.1: Main form λD^+ proof checker.

7.2 Implementation of the first iteration

In section 6.3 we have given the iterative schedule to implement the λD^+ proof checker. In the first iteration we adapt the λC proof checker such that we can use the proof checker for λD^+ judgements. This section describes the adjustments necessary to implement the first iteration of the λD^+ proof checker.

7.2.1 Input of λD^+ judgements

The language λD^+ is an extension of the λ-calculus λC. All λC judgements can be translated to λD^+ judgements. We simply add an empty environment to all λC judgements to find the corresponding λD^+ judgement.

The addition of λD^+ concerns the environment of a judgement. In λD^+ one is able to add axioms and definitions to the environment. The defined constants can be used as an abbreviation for the corresponding axiom or definition in the judgement.

To enter a complete λD^+ judgement, we must extend the input language of the existing λC proof checker. The new input language is an one-to-one translation of the context-free grammar described in Chapter 3.3 and is based on the structure of the λC input language. The input language of the λD^+ proof checker is depicted in figure 7.2.

Note that the given grammar contains the production rules of the non-terminal Context that satisfy the definition of an LL(1) grammar. The same construction is also used for the production rules of the non-terminals Environment and Constant. By defining the lookahead sets for production rules of the context-free grammar we can conclude that the input language of the λD^+ proof checker is an LL(1) grammar.

7.2.2 Parser λD^+ proof checker

The input language of the λD^+ proof checker is an LL(1) grammar as shown in section 7.2.1. Therefore an LL(1) parser is implemented in the λD^+ proof checker to verify the syntax of the input judgement.

Lexical scanner

As described in section 5.2.2 an LL(1) parser uses a lexical scanner. The lexical scanner divides the input judgement into objects called tokens. The tokens are used by the parser to compare the input with the syntax. The defined tokens of the λC proof checker are based on the terminals of the input language. Since the new input language is extended with extra terminals, we must update the lexical scanner. In figure 7.3 an overview is depicted of the defined tokens of the λD^+ proof checker. In the figure the criteria for each token are added.

LL(1) Parser

With the new lexical scanner, we are able to construct the parser of the λD^+ proof checker. The LL(1) parser is implemented in the same way as the parser of the λC proof checker. For each non-terminal or terminal symbol, code is added to the parser as described in section 5.2.2.

Judgement := *judgement* { Environment } { Context } { Statement }
Environment := *environment* { { Declarations-env$_1$
Declarations-env$_1$:= DefinitionSuper } Declarations-env$_2$ | *emptyset* {} } }
Declarations-env$_2$:= } | { DefinitionSuper } Declarations-env$_2$
DefinitionSuper := Definition | Axiom
Axiom := *axiom* { Context } { Constant } { Term }
Definition := *definition* { Context } { Constant } { Statement }
Context := \context { { Declarations-con$_1$
Declarations-con$_1$:= Declaration } Declarations-con$_2$ | *emptyset*{} } }
Declarations-con$_2$:= } | { Declaration } Declarations-con$_2$
Declaration := *declaration* { Variable } { Term }
Statement := *statement* { Term } { Term }
Term := Abstraction | Application | TypeBinder | TypeCat | KindCat | Variable | Constant
Abstraction := *abstraction* { Declaration } { Term }
Application := *application* { Term } { Term }
TypeBinder := *typebinder* { Declaration } { Term }
TypeCat := *ast* {}
KindCat := *square* {}
Variable := *variable* { ([a...z] + [A...Z] + [0...9])$^+$ }
Constant := *constant* { ([a...z] + [A...Z] + [0...9])$^+$ } { { Parameterlist$_1$
Parameterlist$_1$:= Term } Parameterlist$_2$ | *emptyset*{} } }
Parameterlist$_2$:= } | { Term } Parameterlist$_2$

Figure 7.2: Production rules of context-free grammar for input language λD^+ proof checker.

Parse tree

Apart from checking the syntax of an input judgement, the parser has a second task. In the second task the input judgement, in textual representation, is translated to a data structure called a parse tree. With this parse tree the proof checker is able to check the derivability for the corresponding judgement. We must adapt the class diagram of the parse tree classes for the extensions of λD^+. The new class diagram is given in figure 7.4.

In the class diagram the classes Environment, DefinitionSuper, Axiom, Definition and Constant are added and the attribute *environment* is added to the class Judgement.

Extra constraints

In requirement 6.2.2 we have described five extra constraints for the syntax of a syntactically correct judgement. To verify the five constraints, we have extended the algorithm described in section 5.3.1. The current algorithm checks if all variable occurrences in the input judgement are declared in the context, by a λ-term or by a Π-term.

In the top-down scan of the algorithm we simply add the extra syntax constraints. If one of the constraints is not satisfied in the input judgement, a suitable message is given and the proof checker does not proceed with the derivability check.

Character sequence	Token type
\abstraction	ABSTRACTION
\application	APPLICATION
\ast	TYPECAT
\axiom	AXIOM
\constant	CONSTANT
\context	CONTEXT
\declaration	DECLARATION
\definition	DEFINITION
\emptyset	EMPTYSET
\environment	ENVIRONMENT
\judgement	JUDGEMENT
\square	KINDCAT
\statement	STATEMENT
\typebinder	TYPEBINDER
\variable	VARIABLE
{	BEGINBRACKET
}	ENDBRACKET
$[a \dots z]^+$	TEXT, with *value* property equal to $[a \dots z]^+$
position exceeds length of *inputJudgement*	EOF
Otherwise	ERROROBJECT, with *message*.

Figure 7.3: Definition of token in λD^+ proof checker.

7.2.3 Reduction

One δ-reduction step

To implement the *conv* derivation rule, the λD^+ proof checker cannot only deal with α-conversion and β-reduction, but also with δ-reduction. The definition of one-step δ-reduction is given below.

Definition 7.2.1 *One-step δ-reduction, $\rightarrow_\delta$*
If $\Gamma \rhd a(\overline{(x)}) := M : N$ is an element of environment Δ, then
(1) (Basis:) $a(\overline{U}) \rightarrow_\delta M[\overline{x} := \overline{U}]$

The δ-reduction step is, just like the β-reduction step executed, in the parse tree of the judgement. We can perform a δ-reduction step if we can find a CONSTANT object in the parse tree (except for the attribute *constant* of the class DEFINITIONSUPER). If a constant a is found, we must check if the constant is defined by an axiom or a definition in the environment of the judgement. A δ-reduction step is only valid for a definition. If the definition $\Gamma \rhd a(\overline{x}) := M : N$ occurs in the environment, we are able to execute the δ-reduction for the constant a.

To execute the δ-reduction step, the corresponding CONSTANT object is replaced in the parse tree by a new TERM object. The new TERM object defines the term $M[\overline{x} := \overline{U}]$. To create the new object, we begin with a copy of the parse tree of M. In the second step the parameter list $\overline{x}$ is instantiated with the list $\overline{U}$.

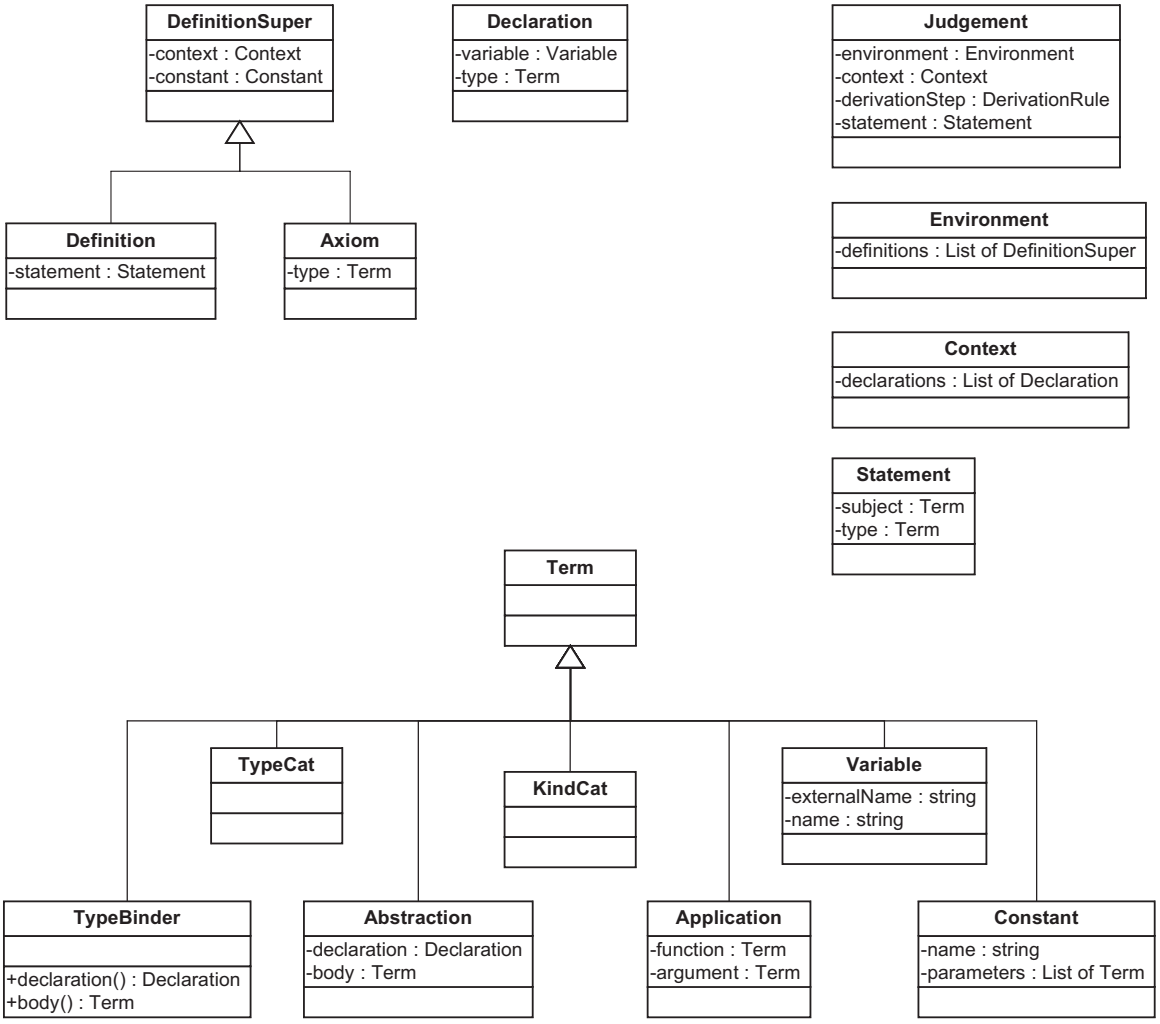

Figure 7.4: Class diagram parse tree λD^+ proof checker.

For instantiating the parameter list we use a method that is comparable to the execution of a β-reduction step. To instantiate the parameter x_1 with U_1, all the VARIABLE objects with *name* equal to x_1 are replaced by a copy of the parse tree of U_1. If parameter x_1 is instantiated, we proceed with the parameter x_2, etc.

We can simply replace all the VARIABLE objects of variable x_k by the term U_k, since we know that the Barendregt-convention holds for the complete judgement. The axiom or definition of the constant a as well the instantiated constant a are an element of the judgement. Therefore we know that the intersection of the declared variables in the axiom or definition and the instantiated constant a is empty. So when instantiating the variable x_k by U_k we know that no occurrences of variables $\overline{x}$ are introduced in the result.

$\beta\delta$-equivalence

To implement the *conv* derivation rule of λD^+ we have to verify if two terms are $\beta\delta$-equivalent. To determine the $\beta\delta$-equivalence relation we use β-reduction and δ-reduction.

In the λC proof checker a simple algorithm is implemented to verify if two terms are β-equivalent. The algorithm performs in most cases too much β-reduction steps, since it reduces both terms to their β-normal form. At that time we indicated that the λD^+ proof checker would contain a more intelligent algorithm to verify the equivalence. This algorithm is designed in the fourth iteration of the λD^+ proof checker (see section 6.3). Since this section contains a description of the first iteration we must also extend the simple algorithm of the λC proof checker. The algorithm is extended such that not only β-equivalence, but also $\beta\delta$-equivalence is included. The more intelligent algorithm of iteration four is described in section 7.5.

To extend the existing β-equivalence algorithm, we add the possibility to execute a δ-reduction step. In the algorithm β-reduction and δ-reduction have the same priority. To verify the $\beta\delta$-equivalence, we still reduce both terms to their normal form by executing β-reduction and δ-reduction on the leftmost $\beta\delta$-redex as long as the term contains a $\beta\delta$-redex. If both terms are reduced to their $\beta\delta$-normal forms we verify if the normal forms are α-equivalent. If the terms are α-equivalent both terms are $\beta\delta$-equivalent, otherwise the terms are not $\beta\delta$-equivalent.

7.2.4 Finding the next derivation rule

The derivation rules of λC are all included (with some extensions) in λD^+. The judgements in λD^+ contain not only a context and a statement, but also an environment. Therefore we must add the environment to the premisses and the conclusion of all the λC derivation rules.

In the λD^+ proof checker we must add the constraints of the environment to the algorithm GETNEXTDERIVATIONRULE. The selection of the next derivation rule does not only depend on the structure of the context and the statement but also on the structure of the environment. For each derivation rule we can define comparable restrictions for the structure of the environment in the conclusion. For example the environment in the conclusion of the *ax* derivation rule is empty. The new restrictions for each derivation rule are added to the algorithm GETNEXTDERIVATIONRULE in the λD^+ proof checker.

The adjustments of the λC derivation rule are not the only thing we need to change in the algorithm GETNEXTDERIVATIONRULE, because in λD^+ four new derivation rules are added, called *env-weak*, *env-weak*$^{\perp\!\!\!\perp}$, *inst* and *inst*$^{\perp\!\!\!\perp}$. To ensure that the algorithm can select the new derivation rules, we must specify the structure of the conclusion for the new derivation rules.

Derivation rules env-weak and env-weak$^{\perp\!\!\!\perp}$

In section 5.4.4 we strengthen the *weak* derivation rule to make the algorithm GETNEXTDERIVATIONRULE deterministic. If we look at the structure of the conclusion of the *env-weak* and the *env-weak*$^{\perp\!\!\!\perp}$, we can see that they have yet another overlap with the other derivation rules. By just adding *env-weak* and *env-weak*$^{\perp\!\!\!\perp}$ derivation we can come once more in the situation where multiple derivation rules are applicable.

To regain the determinism we also strengthen the *env-weak* and the *env-weak*$^{\perp\!\!\!\perp}$ derivation rules just like we strengthen the *weak* derivation rule. The approach is comparable with the *weak* derivation rule. For the *weak* derivation rule we selected the new restrictions such that the *weak* derivation rule is selected if no other derivation is applicable. Therefore a variable

is removed from the context (in the premiss) if no other option is available.

For the *env-weak* and *env-weak*$^{\perp\!\!\!\perp}$ derivation rule we use the same strategy. An axiom or a definition is removed from the environment (in the premisses) if the system has no other option. This occurs when the environment contains one or more axioms and/or definitions, the context is empty and the statement is equal to $* : \Box$. Note that in this situation no other derivation rule is applicable. The new definition of the strengthen *env-weak* and *env-Weak*$^{\perp\!\!\!\perp}$ derivation rule are given in figure 7.2.4.

$$\textit{env-weak} \quad \frac{\Delta;\emptyset \vdash * : \Box \quad \Delta;\overline{x} : \overline{A} \vdash M : N}{\Delta, \overline{x} : \overline{A} \rhd a(\overline{x}) := M : N;\emptyset \vdash * : \Box} \text{ if } a \notin \Delta$$

$$\textit{env-weak}^{\perp\!\!\!\perp} \quad \frac{\Delta;\emptyset \vdash * : \Box \quad \Delta;\overline{x} : \overline{A} \vdash N : s}{\Delta, \overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N;\emptyset \vdash * : \Box} \text{ if } a \notin \Delta$$

Figure 7.5: Strengthened *env-weak* and *env-weak*$^{\perp\!\!\!\perp}$ derivation rules.

Since λD^+ is a new λ-calculus, it is not proven in the literature that the derivation rules with the two new derivation rules have the same expressiveness as the old derivation rules. The proof is also not part of this master assignment. We assume (after consultation with the supervisor of this master thesis) that the two new derivation rules have the same expressiveness power. The proof can probably be constructed by updating the proof of the strengthened *weak* derivation.

Derivation rules *inst* and *inst*$^{\perp\!\!\!\perp}$

To add the *inst* and the *inst*$^{\perp\!\!\!\perp}$ derivation rule we do not have to strengthen the restrictions of the derivation rule, since they do not have an overlap with other derivation rules. But the *inst* and *inst*$^{\perp\!\!\!\perp}$ derivation rule can be preceded by the *conv* derivation rule. Otherwise it would be impossible to define a statement where the subject is equal to a constant a and the type is not α-equivalent but $\beta\delta$-equivalent to the type in the axiom or definition of the constant a.

To check if the next derivation rule is equal to the *conv* derivation or equal to the *inst* or *inst*$^{\perp\!\!\!\perp}$ derivation rule the type of the statement is retrieved by the typing algorithm GET-TYPE. If the retrieved type is α-equivalent to the type of the statement the *inst* or the *inst*$^{\perp\!\!\!\perp}$ derivation rule is selected. And if both terms are β-equivalent the *conv* derivation is executed followed by the *inst* or *inst*$^{\perp\!\!\!\perp}$ rule. This is the same approach we use to check if the *conv* derivation rule is selected before the *abst*, *appl* or *start* derivation rule is applicable.

The existing typing algorithm of the λC proof checker is however not able to retrieve the type of a constant term. For this reason we must adapt the typing algorithm. The new typing algorithm is given in subsection 7.2.5.

Algorithm getNextDerivation

With the adjustment for the λC derivation rule and the *env-weak*, *env-weak*$^{\perp\!\!\!\perp}$, *inst* and *inst*$^{\perp\!\!\!\perp}$ we can give the new algorithm GETNEXTDERIVATIONRULE for the λD^+ proof checker. The algorithm is given in pseudo code in algorithm 10.

Algorithm 10: getNextDerivationRule(Judgement J, out Term newType)

```
/* J can be written as Δ;Γ ⊢ K : L                                   */
switch K do
    case ∗
        if L = □ then
            if Γ = ∅ then
                if Δ = ∅ then
                    return ax
                else if Last declaration in Δ is axiom then
                    return env-weak⊥⊥
                else return env-weak
            else return weak
    case x
        if Γ ≠ ∅ then
            lastDeclaration := Last declaration of Γ ;
            switch Variable of lastDeclaration do
                case x
                    if getType(Γ, x, out newType) then
                        if newType is α-equivalent to L then
                            return start
                        else if type is β-equivalent to L then
                            return conv
                otherwise
                    return weak
    case Πx : A.B
        if L is s then return form
    case MN
        if getType(Γ, MN, out newType) then
            if newType is α-equivalent to L then
                return appl
            else if newType is β-equivalent to L then
                return conv
    case λx : A.M
        if getType(Γ, x : A.M, out newType) then
            if newType is α-equivalent to L then
                return abst
            else if newType is β-equivalent to L then
                return conv
    case a(U⃗)
        if getType(Γ, a(U⃗), out newType) then
            if newType is α-equivalent to L then
                if Last declaration in Δ is axiom then
                    return inst⊥⊥
                else return inst
            else if newType is β-equivalent to L then
                return conv
return noDerivation
```

7.2.5 Typing algorithm

The typing algorithm of λC finds given a context γ and a term M, the type of M. When finding the type of M we assume that Γ is legal and we check the derivability of M. If it appears that Γ is not derivable with the λC derivation rule, then this is found in the main derivability check.

The typing algorithm is implemented by the function GETTYPE (see section 5.4.7). The typing function returns false if M is not derivable and we can therefore not find a type for M. If there is a type A, the function returns true and the type A is stored in the output attribute of the function.

The terms in λD^+ are, in comparison with λC, extended with constants. Since the existing typing algorithm is implemented for λC and λC does not contain constants we must update the typing algorithm for the λD^+ proof checker.

In the environment Δ the constants of the judgement are defined, including their type. For this reason we need to add the environment Δ as an input parameter to the typing algorithm. Just as the context, we assume that the environment is derivable.

But still we have to check if the constant $a(\overline{U})$ is a valid λD^+ term. A constant $a(\overline{U})$ is valid if an axiom $\overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N$ or a definition $\overline{x} : \overline{A} \rhd a(\overline{x}) := M : N$ occurs in Δ. And the instantiations in $\overline{U}$ should have a correct type. This means that the types of $\overline{U}$ are $\beta\delta$-equivalent to $\overline{A}$.

If both conditions hold the type of constant $a(\overline{U})$ is equal to $N[\overline{x} := \overline{U}]$. Note that the conditions for the constants can be deduced from the *inst* and $inst^{\perp\!\!\!\perp}$ derivation rules. The renewed typing algorithm is given in pseudo code in algorithm 11.

Algorithm 11: GETTYPE(Environment Δ, Context Γ, Term M, out Term *type*) : bool

switch M **do**
  case x
    if $x : A \in \Gamma$ **then**
      $type := A$;
      return *true*
    else
      return *false*
  case $*$
    $type := \Box$;
    return *true*
  case KL
    if GETTYPE(Δ, Γ, K, C) *and* GETTYPE(Δ, Γ, L, D) **then**
      if $C \twoheadrightarrow_{\beta\delta} \Pi x : A.B$ *and* $D =_{\beta\delta} B$ **then**
        $type := B[x := L]$;
        return *true*
      else
        return *false*
    else
      return *false*
  case $\lambda x : A.K$
    if GETTYPE$(\Delta, (\Gamma, x : A), K, B)$ **then**
      if GETTYPE$(\Delta, \Gamma, \Pi x : A.B, termtype)$ *and* $termtype \in \{*, \Box\}$ **then**
        $type := \Pi x : A.B$;
        return *true*
      else
        return *false*
    else
      return *false*
  case $\Pi x : A.B$
    if GETTYPE$(\Delta, \Gamma, A, typedec)$ *and* GETTYPE$(\Delta, (\Gamma, x : A), B, typebody)$ **then**
      if $typedec \in \{*, \Box\}$ *and* $typebody \in \{*, \Box\}$ **then**
        $type := typebody$;
        return *true*
      else
        return false
    else
      return false
  case $a(\overline{U})$
    if $\overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : L \in \Delta$ or $\overline{x} : \overline{A} \rhd a(\overline{x}) := K : L \in \Delta$ **then**
      foreach $U_i \in U$ **do**
        if not (GETTYPE(Δ, Γ, U_i, $typeinst_i$) and $typeinst_i =_{\beta\delta} \overline{A_i[\overline{x} := \overline{U}]}$)
        then return *false*
      $type := N[\overline{x} := \overline{U}]$;
      return *true*
    else
      return *false*
  otherwise
    return *false*

7.3 Input assistant

The λC proof checker as well as the λD^+ proof are designed in such a way that the chance of errors in the implementation is minimized. In the design of both systems, some design decisions are made that are maybe less user friendly but guarantee an error-free program.

One of those design decisions is the choice of the existing input language of the λC and λD^+ proof checker. We designed the input language in such a way, that the syntax of the input judgement is verified by an LL(1) parser. In section 5.2.2 we describe that the implementation of the LL(1) parser is easy to verify by hand, since the parser is well-structured. This decreases the probability of errors in the implementation.

The input language is on the other hand not exactly user friendly. The format of the input language does not match the appearance of a λ-judgement as described in Chapter 3. Therefore the user must do some work to translate each λD^+ judgement to the input language. Moreover it costs a lot of effort to represent the input judgement in an orderly fashion. It is almost impossible to represent a syntactically correct input judgement without the use of spaces, tabs, and/or line breaks. Because of the use of extra spaces, tabs and line breaks, the space taken by the input sentence grows rapidly.

What we would like is an input language that corresponds to the grammar of λD^+, without decreasing the reliability of the proof checker. For this we added an external component to the λD^+ proof checker, called the input assistant. The input assistant is a compiler and implements the new input language with the corresponding parser. We shall see that the new input language does not fulfill the definition of an LL(1) grammar and a more complex parser is necessary. Since the input assistant is an external component, the kernel component of the proof checker remains unchanged.

The input assistant communicates only with the user interface component and is used as follows:

> The user enters an input judgement with the new input language. The input assistant checks the input judgement on its syntax. If the input judgement contains one or more syntax errors a message is shown in the user interface to inform the user. If the input judgement is syntactically correct the judgement is translated by the input assistant to the existing input language. This translation shall serve as input for the kernel component

The new input language is based on the language defined in [Zwa]. For the new input language we define the following context-free grammar $G = (N, \Sigma, P, S)$, where:

- The set of non-terminal symbols N is defined by *Abstraction*, *Application*, *Axiom*, *Constant*, *Context*, *Declaration*, *Definition*, *Environment*, *Judgement*, *KindCat*, *Name*, *NonEmptyContext*, *NonEmptyEnvironment*, *Parameters*, *ParenthesisExpression*, *Statement*, *Term*, *TypeBinder*, *TypeCat* and *Variable*.
- The set of terminal symbols Σ is defined by *;*, $|-$, *,*, *>*, *:=*, *||*, *:*, *#*, ***, *(*, *)*, **, *?* and the characters to define a name.
- The production rule P are defined in figure 7.6.
- The start symbool S is the non-terminal *Judgement*.

Judgement	:= Environment ; Context |− Statement
Environment	:= empty | NonEmptyEnvironment
NonEmptyEnvironment	:= NonEmptyEnvironment, NonEmptyEnvironment | DefinitionSuper
DefinitionSuper	:= Axiom | Definition
Definition	:= Context > Constant := Statement
Axiom	:= Context > Constant := || : Term
Context	:= empty | NonEmptyContext
NonEmptyContext	:= NonEmptyContext, NonEmptyContext | Declaration
Declaration	:= Variable : Term
Statement	:= Term : Term
Term	:= Abstraction | Application | Constant | KindCat | ParenthesisExpression | TypeBinder | TypeCat | Variable
Abstraction	:= \ Declaration . Term
Application	:= Term Term
Constant	:= ?Name() | ?Name(Parameters)
KindCat	:= ∗
ParenthesisExpression	:= (Term)
TypeBinder	:= @ Declaration . Term
TypeCat	:= #
Variable	:= Name
Parameters	:= Parameters, Parameters | Term
Name	:= NameName | [a. . . z] | [A. . . Z] | [0. . . 9] | -

Figure 7.6: Production rules for the language of the input assistant.

Note that the language of the input assistant is not an LL(1) grammar, since the intersection of the lookahead sets of the non-terminals Environment, NonEmptyEnvironment, DefinitionSuper, NonEmptyContext, Term, Constant and Paramters are not empty. Some of the production rules can be rewritten, as described in section 5.2.2, to create an empty intersection of the lookahead sets. For the non-terminal Term this is impossible. The production $Term \rightarrow Application$ cannot be rewritten without using some kind of backtracking.

We shall not give a description of the implementation of the input assistant in this master thesis. For a description we refer to the technical specification of the input assistant. The technical specification is added in Appendix A.

7.4 Library

In this section we describe then usefulness of a library for the λD^+ proof checker. In order to clarity why such a library may be helpful, we start with an example.

The definition of the logical operator bi-implication ($A \Leftrightarrow B$) is given as $A \Rightarrow B \wedge B \Rightarrow A$. If we want to check this definition with the λD^+ proof checker we must define a judgement where the definition of the bi-implication as well as the definition of the implication and the conjunction are added to the environment. We must add the implication and conjunction definition to the environment, since they have a dependency relation with the bi-implication

definition. As a result the definition of the implication and the conjunction are checked on their derivability when checking the definition of the bi-implication.

The extra work for this particular example is small, because the amount of work for checking the definitions of the implication and conjunction is fairly small. We can imagine that when mathematics becomes more complex it depends more and more on existing axioms and definitions. These constant can also depend on other axioms and definitions, and so on. If we want to prove a new lemma or theorem all the axioms and definitions on which this depends, must be added to the environment. In the derivability check for the new lemma or theorem the complete environment is included. This decreases the performance of the derivability check of the λD^+ proof checker.

What we would like is a library of derivable axioms and definitions. The library constants can be used in new proofs, without checking the derivability of the corresponding axioms and definitions. Therefore it is important to guarantee the correctness of the library. A library is correct if it contains only derivable axioms and definitions. Note that underivable axioms and definitions could lead to an inconsistent system.

For the λD^+ proof checker such a library is included. The library component is depicted in the global overview of the λD^+ proof checker as a sub-component of the user interface component. The user can use the library to define a new input judgement. The set of axioms and definitions of the library are stored in a Microsoft Office Access database. An overview of the tables is given in figure 7.7.

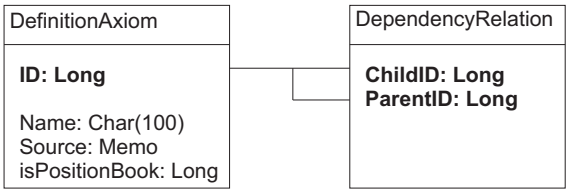

Figure 7.7: Database library.

In the rest of this section, we give a description of the library. First we describe how we can access the library from the user interface. Then we describe how the λD^+ proof checker maintains the correctness of the library when adding and removing axioms and definitions. In the last subsection we show how the user can use the library for constructing an input judgement.

7.4.1 User interface

From the main form (see figure 7.1) of the λD^+ proof checker the user is able to open the library module by the button 'Library'. The user interface of the library is shown in figure 7.8 and figure 7.9.

The user interface consist of two tab pages. In the tab page 'new', depicted in figure 7.8, the user is able to add a new axiom or definition to the library. Note that in the figure the definition of the bi-implication is inserted. The tab page 'overview' shows the user an overview of the axioms and definitions that are already added to the library. In the tab page the user is also able to remove an axiom or definition from the library.

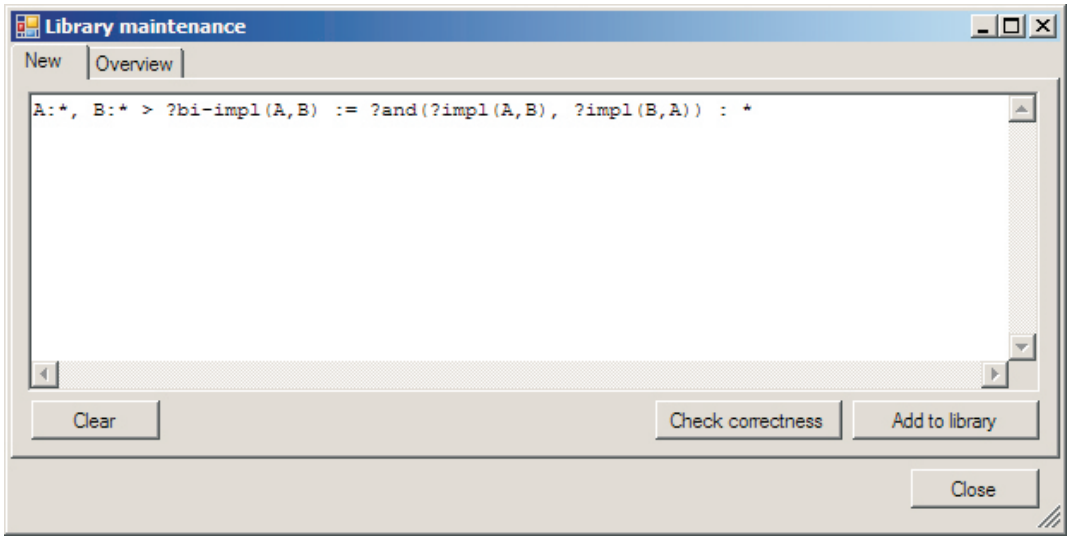

Figure 7.8: User interface library.

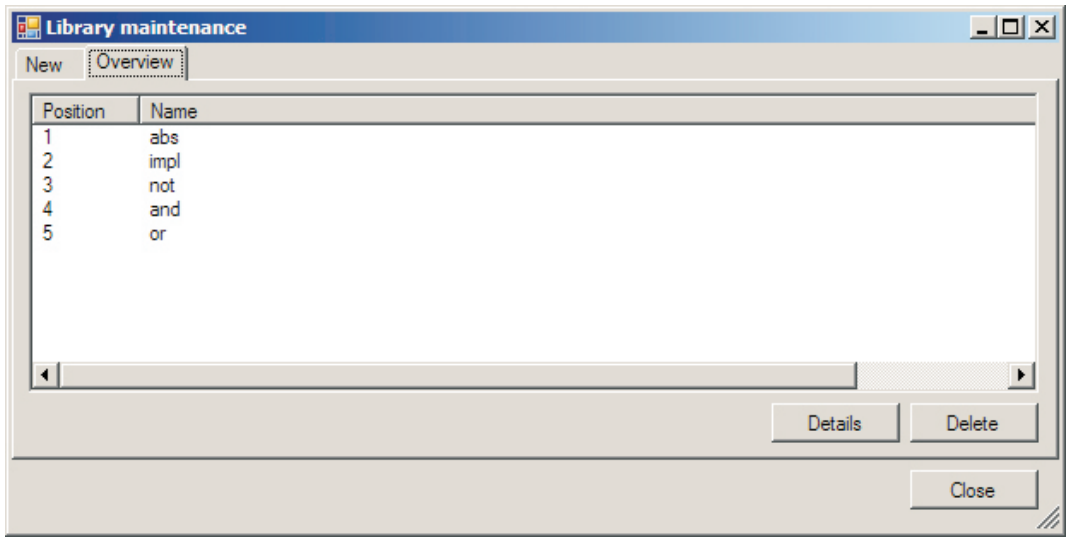

Figure 7.9: User interface library.

7.4.2 Add a new axiom or definition to the library

In the user interface, given in figure 7.8, the user is able to add an axiom or a definition to the library. It is only possible to add one axiom or definition at a time. The name of the new constant must be unique in the library. To define a new axiom or definition the context-free grammar of the input assistant is used (see section 7.3). Since we are not defining a judgement but an axiom or a definition the start symbol is equal to the non-terminal DefinitionSuper. The specification of the new constant can contain constants that have already been added to the library.

In the library only derivable axioms and definitions are allowed. Before a new axiom or

definition D is added to the library, D is checked on its syntax and its derivability. For both checks we use the existing algorithms of the λD^+ proof checker. To use the algorithm, we transform D to the judgement $D; \emptyset \vdash * : \Box$. Since the specification of D can also contain library constants, we are forced to add at least the depend axioms and definitions to the environment of the judgement. In our implementation we have decided to add all the axioms and definitions of the library to the environment. We will see in section 7.4.4 that the addition of all library constants will not influence the performance of the derivability check. The judgement that is used to check the syntax and the derivability of D has therefore the structure $D_1, \ldots, D_n, D; \emptyset \vdash * : \Box$, where D_1 till D_n are the axioms and the definitions of the library sorted on time of insertion in *inPositionBook*.

The syntax of the judgement is checked by the input assistant, while the kernel component checks the derivability. If the judgement contains no syntax errors and is derivable by the derivation rules of λD^+, the definition D and its direct dependencies are stored in the database.

7.4.3 Remove an axiom or definition from the library

Apart from adding an new axiom or definition to the library, it is possible to remove an existing constant from the library. When removing a constant we also make sure that the library maintains correct. A delete action cannot result in 'open holes' in the library. This means that the user is only able to remove an axiom or a definition D from the library if the library contains no other axiom or definition D', where D' directly depends on D. If there are other constants with a direct dependency relation to D a message is given in the user interface to inform the user. In the message the directly depending constants are summarized.

7.4.4 Define input judgement by using the library

The axioms and definitions of the library can be used to define an input judgement in the λD^+ proof checker. For this we added the checkbox 'Use library' in the main form. If the checkbox is not checked, the library is ignored. As a consequence, the user must add the constants of the input judgement to the environment by hand.

If the user wants to use the library to define an input judgement, the checkbox must be checked. The user is able to use all library constants in the input judgement without adding the corresponding axiom or definition to the environment by hand. The λD^+ proof checker adds automatically all the axioms and the definitions of the library in front of the environment. If the user enters the judgement $\Delta; \Gamma \vdash * : \Box$ and Δ' contains the library axioms and definitions sorted on time of insertion in *inPositionBook*, the input judgement will be equal to $\Delta', \Delta; \Gamma \vdash * : \Box$. Note that the intersection of the constant names of Δ and Δ' must be empty.

Next to the advantage not having to define the complete environment by hand, the library increases the performance of the derivability check in the λD^+ proof checker. Because of the strict rules for adding and removing axioms and definitions from the library, we know that the judgement $\Delta'; \emptyset \vdash * : \Box$ is derivable. When the input judgement $\Delta', \Delta; \Gamma \vdash * : \Box$ is checked on its derivabilty, the axioms and definitions of Δ' are not checked once again.

To implement this feature the DERIVE function of the *env-weak* and *env-weak*$^{\perp\!\!\!\perp}$ derivation rule is adapted. The definitions are given, for completeness' sake, in figure 7.4.4.

$$\textit{env-weak} \quad \frac{\Delta;\emptyset \vdash * : \Box \quad \Delta;\overline{x} : \overline{A} \vdash M : N}{\Delta, \overline{x} : \overline{A} \rhd a(\overline{x}) := M : N;\emptyset \vdash * : \Box} \text{ if } a \notin \Delta$$

$$\textit{env-weak}^{\perp\!\!\!\perp} \quad \frac{\Delta;\emptyset \vdash * : \Box \quad \Delta;\overline{x} : \overline{A} \vdash N : s}{\Delta, \overline{x} : \overline{A} \rhd a(\overline{x}) := \perp\!\!\!\perp : N;\emptyset \vdash * : \Box} \text{ if } a \notin \Delta$$

Figure 7.10: Definition *env-weak* and *env-weak*$^{\perp\!\!\!\perp}$ derivation rules.

In the DERIVE function for both derivation rules two premisses are created. The premiss $\Delta;\emptyset \vdash *\Box$ is added to check the axioms and definitions of Δ on their derivability. While the premiss $\Delta;\overline{x} : \overline{A} \vdash M : N$ is added to verify if the specification of the constant a is derivable. Since we already know that a library constant is derivable, the second premiss is skipped. If the *env-weak* or *env-weak*$^{\perp\!\!\!\perp}$ derivation rule is executed for a constant that is not added to the library, the derivation rules are executed normally and both premisses are created and checked on their derivability.

7.5 $\beta\delta$-equivalence

In the first iteration of the λD^+ proof checker we implemented a simple algorithm to verify if the terms A and B are $\beta\delta$-equivalent. The algorithm first reduces A and B to their $\beta\delta$-normal form and then checks if the obtained $\beta\delta$-normal forms are α-equivalent. If the $\beta\delta$-normal forms are α-equivalent we conclude that A and B are $\beta\delta$-equivalent. When the two $\beta\delta$-normal forms are not α-equivalent then A and B are not $\beta\delta$-equivalent.

We can use this strategy since we know that the Strong Normalizing Theorem and the Church-Rosser Theorem hold for $\beta\delta$-equivalence in λD^+. But the algorithm is not very efficient, because we always reduce both terms to their corresponding $\beta\delta$-normal form.

Example 7.5.1 *Take for example the terms $(\lambda x : A.B)M$ and $(\lambda x : A.B)N$, where $M \neq_\alpha N$. To verify if both terms are $\beta\delta$-equivalent, it is sufficient to check if the terms M and N are $\beta\delta$-equivalent. With this approach we do not use β-reduction on the β-redex and we do not have to reduce B to its $\beta\delta$-normal form.*

In the last iteration we studied the options to optimize the algorithm for determining the $\beta\delta$-equivalence relation between two terms. From the research we have implemented one possible solution. Although we have tested the algorithm with several examples from [Ned], we have not proven that the algorithm covers all possible cases.

The new algorithm is designed to use the structure of the terms and only performs β-reduction and δ-reduction steps if α-conversion is not sufficient to verify the $\beta\delta$-equivalence relation between the terms M and N.

In the design of the algorithm, we assume that terms in λD^+ contain a 'minimal' number of β-redexes. This assumption is made because one can use constants in λD^+. If M and N are not α-equivalent and if one of the terms or both terms are a β-redex we therefore always execute the corresponding β-reduction step. Note that this is only an intuitive argument and we can easily formulate examples where the argument does not hold.

The new algorithm is called areBetaDeltaEquivalent. The algorithm has four input parameters:

- The terms M and N for which the algorithm verifies the $\beta\delta$-equivalence relation.
- The environment Δ. The environment is needed to execute δ-reduction steps in M and N.
- The context Γ, where all the free variables of M and N are declared.

The algorithm returns true if the terms M and N are $\beta\delta$-equivalent, with respect to Γ and Δ. If M and N are not $\beta\delta$-equivalent the algorithm returns false. We describe the algorithm by using case distinction on M and N. The list of cases can be seen as a 'switch' statement where only one case is applicable per iteration. To find to correct case for the given terms M and N, we first check the first case, then the second, etc.

After each β-reduction step or δ-reduction step we will rename all the binding and corresponding bound variables in the term. We rename the variables by using the method described in section 5.3.3. In the description of the algorithm we will not mention this after each β-reduction and/or δ-reduction step explicitly.

We distinguish the following cases:

1. The terms M and N are α-equivalent. If both terms are α-equivalent, we can conclude that M and N are $\beta\delta$-equivalent and the algorithm returns true.
2. Term M is equal to$(\lambda x : A.B)C$ and term N is equal to $(\lambda y : D.E)F$. For both terms a β-reduction step is executed. After this the algorithm will continue by checking the $\beta\delta$-equivalence relation between the terms $B[x := C]$ and $E[y := F]$.
3. Term M is equal to$(\lambda x : A.B)C$ and term N is equal to D, where D is not a β-redex. The algorithm performs a β-reduction step for M. After this the algorithm will continue by checking the $\beta\delta$-equivalence relation relation between the terms $B[x := C]$ and D.
4. Term M is equal to D and term N is equal to $(\lambda x : A.B)C$, where D is not a β-redex. See situation 3, with $M \equiv N$ and $N \equiv M$.
5. Term M is equal to $a(\overline{U})$ en N is equal to $a(\overline{V})$, where in Δ the definition $\overline{x} : \overline{A} \rhd a(\overline{x}) := K : L$ is added. Both terms start with the same constant. To verify if both terms are $\beta\delta$-equivalent, we first check for each $U_i \in \overline{U}$ and $V_i \in \overline{V}$ if $U_i =_{\beta\delta} V_i$. If all the parameters are $\beta\delta$-equivalent, we know that M is also $\beta\delta$-equivalent to N and the function returns true.

   If one or more parameters are not $\beta\delta$-equivalent, it does not mean that we can conclude that $M \neq_{\beta\delta} N$. Take for example $plus(1, 1)$ and $plus(0, 2)$. Both terms have parameters that are not $\beta\delta$-equivalent but for both terms the $\beta\delta$-normal form is equal to 2. Therefore we execute a δ-reduction step on M and N. After this the algorithm will continue by checking the $\beta\delta$-equivalence relation relation between the terms $K[\overline{x} := \overline{U}]$ and $K[\overline{x} := \overline{V}]$.
6. Term M is equal to $a(\overline{U})$ and N is equal to $b(\overline{V})$, where in Δ the definitions $\overline{x} : \overline{A} \rhd a(\overline{x}) := K : L$ en $\overline{y} : \overline{B} \rhd b(\overline{y}) := I : J$ are added. To find the next step in the

algorithm we check which constant a or b is declared last in Δ. For the constant that is declared last we perform a δ-reduction step.

(a) If constant a is declared last in Δ, we perform a δ-reduction step on M. We do not perform a δ-reduction step on N, since this will never introduce the constant a, because the constant a is declared before the constant b. After the δ-reduction step on M, the algorithm continues by checking the $\beta\delta$-equivalence relation between the terms $K[\overline{x} := \overline{U}]$ and $b(\overline{V})$.

(b) If constant b is declared last in Δ, we perform a δ-reduction step on N. After this the algorithm will continue by checking the $\beta\delta$-equivalence relation relation between the terms $a(\overline{U})$ and $I[\overline{y} := \overline{V}]$.

7. Term M is equal to $a(\overline{U})$ and N is equal to B, where B is not a β-redex or δ-redex and the definition $\overline{x} : \overline{A} \triangleright a(\overline{x}) := K : L$ is added to Δ. The algorithm performs a δ-reduction on M. After this the algorithm will continue by checking the $\beta\delta$-equivalence relation relation between the terms $K[\overline{x} := \overline{U}]$ and B.

8. Term M is equal to B and N is equal to $a(\overline{U})$, where B is not a β or δ-redex and the definition $\overline{x} : \overline{A} \triangleright a(\overline{x}) := K : L$ is added to Δ. See situation 7, with $M \equiv N$ and $N \equiv M$.

9. Term M is equal to $a(\overline{U})$ and N is equal to $a(\overline{V})$, where the axiom $\overline{x} : \overline{A} \triangleright a(\overline{x}) := \perp\!\!\!\perp : L$ is added to Δ. To verify if both terms are $\beta\delta$-equivalent, we check for each $U_i \in \overline{U}$ and $V_i \in \overline{V}$ if $U_i =_{\beta\delta} V_i$. If all the parameters are $\beta\delta$-equivalent, we know that M is also $\beta\delta$-equivalent to N and the function returns true. Otherwise the function return False.

10. M is equal to $\lambda x : A.B$ and N is equal to $\lambda y : C.D$. The abstraction in both terms will remain even after using β-reduction and δ-reduction. Therefore the algorithm will first check if $A =_{\beta\delta} C$. If both terms are not $\beta\delta$-equivalent we can conclude that $M \neq_{\beta\delta} N$ and the algorithm returns false.

    If A is $\beta\delta$-equivalent to C, then the algorithm continues by checking $B =_{\beta\delta} D[y := x]$. If both terms are $\beta\delta$-equivalent we conclude $M =_{\beta\delta} N$ and the algorithm returns true. Are both terms not $\beta\delta$-equivalent we conclude $M \neq_{\beta\delta} N$ and the algorithm returns false.

11. M is equal to $\Pi x : A.B$ and N is equal to $\Pi y : C.D$. The type binder in both terms will remain even after using β-reduction and δ-reduction. Therefore the algorithm will first check if $A =_{\beta\delta} C$. If both terms are not $\beta\delta$-equivalent we can conclude that $M \neq_{\beta\delta} N$ and the algorithm returns false.

    If A is $\beta\delta$-equivalent to C, then the algorithm continues by checking $B =_{\beta\delta} D[y := x]$. If both terms are $\beta\delta$-equivalent we conclude $M =_{\beta\delta} N$ and the algorithm returns true. Are both terms not $\beta\delta$-equivalent we conclude $M \neq_{\beta\delta} N$ and the algorithm returns false.

12. M is equal to AB and N is equal to CD, where M and N are not a β-redex. The algorithm first checks if $A =_{\beta\delta} C$ and $B =_{\beta\delta} D$. If both checks return true, we conclude that $M =_{\beta\delta} N$ and the algorithm returns true.

If the function or the argument in M and N are not $\beta\delta$-equivalent, we cannot conclude that M is not $\beta\delta$-equivalent to N, since we may be able to reduce A and/or B to its λ-head form. We can distinguish the following four situations:

- Both A and B have no λ-head form. We now can conclude $M \neq_{\beta\delta} N$ and the algorithm return false.
- A has the λ-head form $\lambda x : K.L$ and B has no λ-head form. The algorithm continues by checking the $\beta\delta$-equivalence relation between the terms $(\lambda x : K.L)B$ and CD.
- A has no λ-head form and B has the λ-head form $\lambda x : K.L$. The algorithm continues by checking the $\beta\delta$-equivalence relation between the terms AB and $(\lambda x : K.L)D$.
- A has the λ-head form $\lambda x : K.L$ and B has the λ-head form $\lambda y : I.J$. The algorithm continues by checking the $\beta\delta$-equivalence relation between the terms $(\lambda x : K.L)B$ and $(\lambda y : I.J)D$.

13. M is equal to AB and N is equal to C, where AB is no β-redex and C is not equal to DE and no δ-redex. We can continue the algorithm if we can reduce AB to a β-redex. To create the β-redex we must reduce A to its λ-head form. If A has no λ-head form, the term M will always remain an application, while N is not an application. Therefore we can conclude that $M \neq_{\beta\delta} N$ and the algorithms returns false.

    If A has a λ-head form $\lambda x : D.E$ the algorithm continues by checking the $\beta\delta$-equivalence relation between the terms $(\lambda x : K.E)B$ en C.

14. M is equal to C and N is equal to AB, where AB is no β-redex and C is not equal to DE. See situation 13, with $M \equiv N$ and $N \equiv M$.

15. For all other cases we conclude $M \neq_{\beta\delta} N$ and the algorithm returns false.

Since we have not proven that the algorithm covers all possible cases, we cannot guarantee that the given 14 cases are sufficient. In the future work of this master thesis the proof of the $\beta\delta$-equivalence algorithm is added.

Chapter 8

Conclusions and future work

In this master thesis we implemented two proof checkers in Microsoft Visual Studio in the computer language C#. The first proof checker can verify the correctness of a formal proof formulate in λC. The λC proof checker is designed and implemented as a preparation for the λD^+ proof checker. We designed the λC proof checker is such a way that we were able to extend the proof checker to the λ-calculus λD^+.

Both proof checkers are designed in such a way that the chance of errors is minimized. Errors can lead to a inconsistent system. We discovered that it is very hard to combine this design decision with user friendliness. In the design of the λC proof checker all the decisions are based only on obtaining an error-free system. Whereas in the λD^+ proof checker we also design and implement some user friendly requirements.

8.1 λC proof checker

To verify the correctness of a mathematical proof in λC, the λC proof checker contains three distinct components: a user interface, a kernel and an eye component. The user can enter an input judgement into the user interface of the λC proof checker. This judgement is checked on its syntax and its derivability by the kernel component. The result of the derivability check, the derivation tree, is reported in list notation in the user interface. The eye component stores for each judgement in the derivation tree, all the internal steps that are necessary to verify the derivability of the corresponding judgement. Also the data of the eye component are reported in the user interface.

For the λC proof checker an input language is designed to formulate an input judgement J. The input language is not user friendly but the syntax of the input judgement can be verified by an LL(1) parser. An LL(1) parser is an efficient and well-structured parser. Because the parser is structured, it is easy to check the implementation on errors. Apart from checking the syntax of an input judgement, the LL(1) parser translates the judgement into a parse tree. The parse tree is used as input for all the other algorithms in the proof checker.

For the derivability check, we have designed and implemented a deterministic and recursive algorithm. The algorithm tries to compute a derivation tree for the input judgement J. In the derivation tree every derivation rule is explicitly labeled with its name, premisses and conclusion. In each recursion step one derivation rule in the derivation tree is executed. The derivability check starts with the input judgement.

Every recursion step has a judgement K as input. This judgement is assigned as the conclusion of the derivation rule that is executed in the current recursion step. We divide each recursion step in two parts. In the first part we find a suitable derivation rule for K, by comparing the structure of K with the structure of the conclusion of the distinct derivation rules of λC. To ensure that only one derivation rule is applicable for each judgement, we have strengthened the *weak* derivation rule. If no derivation rule is applicable for K, we conclude that K is not derivable and therefore also the input judgement J is not derivable.

If the algorithm finds a derivation rule for K, we create premisses based on K and the selected derivation rule. If we look at the shape of the derivation rules, we see that we can simply copy the data for the context and the subject from K for each premiss. On the other hand the type of the premisses can contain components that do not occur in K. To retrieve the types for the premisses, a typing algorithm for λC is included in the proof checker. The typing algorithm finds, given a context and a term M, the type of M. By retrieving the type of the subject with the typing algorithm, we can create all premisses. When all premisses are created, the recursion starts for each premiss.

We also use the typing algorithm when selecting the next derivation rule. Therefore it can occur that the typing algorithm is executed multiple times for the same subject. This we can optimize by storing the retrieved type for each judgement. This optimalization is not included in the master project.

In the computed derivation tree every α-equivalent judgement occurs only once, by using sharing. Each judgement that is evaluated as derivable during the derivability check of J is stored. Before the algorithm verifies a judgement K on its derivability, the algorithm checks if a comparable judgement L is already checked. If this is the case, then we replace K by L. By using sharing we can use a judgement multiple times as premiss for a derivation rule in the derivation tree.

To check if a judgement is already evaluated, we use the internal representation. For each judgement in the derivation tree we rename all the variables. Since we use the same method for all the judgements, judgements that are α-equivalent have the same internal representation. With the internal representation it is easy to compare two judgements.

The renaming has also another advantage. By renaming the variables we ensure the Barendregt convention for all judgements in the derivation tree. We use this assumption to create an easy algorithm for β-reduction. We need β-reduction for determining the β-equivalence between two terms A and B. The λC proof checker contains a simple but very inefficient algorithm for this. The algorithm first reduces A and B to their β-normal form and then checks if both β-normal forms are α-equivalent. In the λD^+ proof checker we optimized this algorithm.

8.2 λD^+ proof checker

The λD^+ proof checker is an extended version of the λC proof checker, with some additional user friendly features. The λD^+ proof checker uses the same method to check the derivability of a judgement. But since λD^+ is an extension of λC we had to extend almost every part of the λC proof checker. The existing input language is extended such that an judgement contains an environment, where one can add axioms and definitions. The typing algorithm is now also able to retrieve a type for a constant, and an environment is added to the judgements

in the λC derivation rules.

Next to the λC derivation rules, λD^+ contains four new derivation rules which are implemented in the λD^+ proof checker. To maintain the determinism of the derivability check, we have also strengthened the *env-weak* and the *env-weak*$^{\perp\!\!\!\perp}$ derivation rule. Since λD^+ is a new λ-calculus, it is not proven in the literature that the λD^+ derivation rules including the two strengthened derivation rules have the same expressiveness as the old λD^+ derivation rules. In this master project the proof is not included.

Another extension was needed for the algorithm that determines the β-equivalence relation between two terms. In λD^+ β-equivalence is not sufficient anymore, because we need $\beta\delta$-equivalence. In the λD^+ proof we have extended the simple but very inefficient β-equivalence algorithm to $\beta\delta$-equivalence and we have designed an optimized $\beta\delta$-equivalence algorithm. The optimized $\beta\delta$-equivalence algorithm uses the structure of both terms and performs on average less β-reduction and δ-reduction steps. The correctness of the optimized $\beta\delta$-equivalence algorithm is only tested by the examples of [Ned]. Due to time constraints we were not able to prove that the algorithm covers all possible cases.

Next to extending the λC proof checker to λD^+, the λD^+ proof checker contains two user friendly features. For the λD^+ proof checker a user friendly input language and a library of legal axioms and definitions are designed and implemented.

The input assistant is a compiler that is added as a subcomponent to the user interface. The input assistant implements the new input language. The user enters a judgement with the new input language in the user interface. This input judgement is checked by the input assistant on its grammar and translates the input to the existing input language. This translation is used as input for the kernel component.

With the library the user can store derived axioms and definitions. We can reuse these axioms and definitions in the input judgement. When using the library all the axioms and definitions, stored in the library, are added to the input judgement. In the future we can optimize this by only adding the related library constants. In the derivability check the library axioms and definitions are not checked on their derivability, since we already know that the library axioms and definitions are derivable.

8.3 Future work

We conclude with describing some future work. In the list we also add some items that were not mentioned before.

- Prove that the λD^+ derivation rules with the strengthened environment weakening rules have the same expressiveness as the existing λD^+ derivation rules.
- Prove that optimized $\beta\delta$-equivalence relation is complete and covers all distinct cases.
- Optimize the derivability check, such that the typing algorithm is invoked only once per judgement.
- When using the library, only add related axioms and definitions to the input judgement.

- For the λD^+ proof checker we have not implemented the fifth iteration. With the fifth iteration the result of the derivability check can be reported in flag style notation. This is desirable when the proof checker is used as a comparison tool for the book [Ned].

- In section 2.3 we have described that the kernel component itself is just a computer program, so its correctness can be verified by an existing proof checker or a proof assistant. Now we only guarantee the correctness by designing the kernel well-structured but by using another proof assistant we can guarantee the correctness to a greater extend.

  We foresee, however, that it will cost a lot of effort to formulate the kernel component into a formal language.

- Almost every algorithm in both proof checkers is recursive. When using Microsoft Visual Studio each program has a limited recursion stack. The examples that we have tested did not reach this limited. I assume that when the examples remain to increase, we will eventually reach the limit. For those examples it is impossible to verify the derivability with the proof checkers, hence, a solution has to be found for this.

- Update user interface of λD^+ proof checker in order to become a handy assistant for students using the textbook [Ned]. With the new user interface the proof checker enables the user to do formalization exercises with λD^+, in a gentle way. The following options are recommended for the update:

  - Create a web-based user interface, such that students can access the proof checker by using the internet.
  - Create an electronic tutorial for the λD^+ proof checker.
  - Add interaction between user and proof checker. In the previous item of the future work we have described that each program in Microsoft Visual Studio has a limited recursion stack. If the program reaches the limit of the recursion stack, the program must interact with the user to find a suitable solution.

Bibliography

[Abr] S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors. *Handbook of Logic in Computers Science, Volume 2: Background Computational Structures*. Oxford University Press, 1992.

[Bar1] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics **103**. North-Holland, Amsterdam, revised edition, 1984.

[Bar2] H.P. Barendregt. *Lambda calculi with types*. In [Abr], pages 117-309. Oxford University Press, 1992.

[Ben] L.S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for Pure Type Systems. In Barendregt and Nipkow, editors, TYPES'93: Workshop on Types for Proofs and Programs, Selected Papers, volume 806 of *Lecture Notes in Computer Science*, pages 19-61. Springer-Verlag, 1994.

[Bru] N.G. de Bruijn. *Automath, a language for mathematics*. Technical report 68-WSK-05, T.H.-Reports, Eindhoven University of Technology, 1968.

[Geu1] H.Geuvers. Proof assistants: History, ideas and future. In *Sadahana Journal*, Academy Proceedings in Engineering Sciences, Special Issue on Interactive Theorem Proving and Proof Checking, Indian Academy of Sciences, Vol 34, part 1, February 2009, pages 3-25.

[Geu2] H. Geuvers. Introduction to Type Theory. In *Language Engineering and Rigourous Software Development* (Revised Tutorial Lecture Notes of LerNet ALFA Summer School Piriapolis, Uruguay, 24 February to 1 March, 2008), eds.: Ana Bove, Luis Soares Barbosa, Alberto Pardo, Jorge Sousa Pinto, LNCS 5520, Springer 2009.

[Har] J. Harrison. Formal Proof - Theory and Practice. In *Notices of the American Mathematical Society*, volume 55 isue 11, pages 1408-1414, 2008.

[Hop] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[Jeu] J. Jeuring, S.D. Swierstra. *Grammars and Parsing*, lecture notes Talen en ontleders. Open Universiteit Nederland, 2001.

[Mol] B. Mols. *QED zegt de computer*. In NRC Handelsblad, volume 31-01-2009.

[Ned] R.P. Nederpelt Lazarom. *Basics of Type Theory*, draft version, 2009.

[Wie] F. Wiedijk. Formal proof getting started. In *Notices of the American Mathematical Society*, volume 55 isue 11 pages 1395-1460, 2008.

[Zwa] J. Zwanenburg. *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow*, PhD thesis, Eindhoven University of Technology. University Press Facilities, Eindhoven, 1999.

Appendix A

Input Assistant

A.1 Functional specification

The new input language is able to compile the grammar described in requirement A.1.1. The compiler is able to translate the input language into the format needed for the λD^+ proof checker. The language of the λD^+ proof checker is described in the technical specification of the λD^+ proof checker.

Requirement A.1.1 *Input grammar*

Judgement	:= *Environment ; Context |− Statement*
Environment	:= *empty* | *NonEmptyEnvironment*
NonEmptyEnvironment	:= *NonEmptyEnvironment, NonEmptyEnvironment* | *Definition* | *Axiom*
Definition	:= *Context > Constant := Statement*
Axiom	:= *Context > Constant := || : Term*
Context	:= *empty* | *NonEmptyContext*
NonEmptyContext	:= *NonEmptyContext, NonEmptyContext* | *Declaration*
Declaration	:= *Variable : Term*
Statement	:= *Term : Term*
Term	:= *Variable* | # | ∗ | *Term Term* | *(Term)* | \ *Declaration. Term* @ *Declaration. Term* | *Constant*
Variable	:= *Name*
Constant	:= *?Name(Parameters)* | *?Name()*
Parameters	:= *Parameters, Parameters* | *Term*
Name	:= *NameName* | *[a... z]* | *[A... Z]* | *[0... 9]* | -

A.2 Technical specification

A.2.1 Abstraction

The Abstraction class implements the following part of the input language: \ *Declaration. Term* of the node Term.

Inheritance relation Class ABSTRACTION inherits from class TERM and the class TERM inherits from JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *body* (Term). Reference to the root of the parse tree that describes the body of the abstraction. The body is the abstract output value of the formula.
- *declaration* (Declaration). Reference to the root of the parse tree that describes the declaration of the abstraction. The declaration describes the input of the formula.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - ABSTRACTION. Goto next step.
     - EOF. Return False with *errorMessage* equal to 'EOF found when expecting \ sign of abstraction.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Abstraction must start with \.'.
  2. Create for *declaration* new DECLARATION object.
  3. Call PARSE function for *declaration*. Based on the return value of the recursive function call an action is taken:
     - True. Pointer of output attribute *parseObject* of recursive function call is stored in *declaration* and goto next step.
     - False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
  4. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - HYPHEN. Based on the type of the HYPHEN object an action is taken:
       * DOT. Goto next step.
       * Otherwise. Function returns False, with *errorMessage* equal to: 'Declaration of abstraction must be followed by a dot sign.'.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting dot sign of abstraction.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Declaration of abstraction must be followed by a dot sign.'.
  5. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:

- Class that inherits from class TERM. A pointer to the object is stored in *body*. Call function PARSE for *body*. If the recursive call returns True then store pointer of output variable *parseObject* in *body* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.
- EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting body of abstraction.'.
- ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
- Otherwise. Return False with *errorMessage* equal to 'Body of abstraction must be a term.'.

6. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
   - Class that inherits from class TERM.
     (a) Create APPLICATION object and store current abstraction in *function* property of APPLICATION object.
     (b) Call function PARSE of APPLICATION object. Depending on the return value an action is taken:
         * True. Function returns True and store a pointer of *parseObject* of the recursive function call in *parseObject*.
         * False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call. Also store a pointer of *parseObject* of the recursive function call in *parseObject*.
   - Otherwise. Return True and store pointer of current abstraction in *parseObject*.

- TOINPUTLANGUAGE():STRING. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \ abstraction { *declaration.toInputLanguage()* } { *body.toInputLanguage()* }

- TOSTRING():STRING. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  \ *declaration.toString()* . *body.toString()*.

A.2.2 Application

The APPLICATION class implements the following part of the input language: *Term Term* of the node Term.

Inheritance relation Class APPLICATION inherits from class TERM and the class TERM inherits from JUDGEMENTOBJECT.

Attributes

- *argument*. Reference to the root of the parse tree that describes the argument of the application.

- *classType* (ClassType). For description see parent class.
- *function*. Reference to the root of the parse tree that describes the function of the application.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
     - Class that inherits from class TERM. A pointer to the object is stored in *argument*. Call function PARSE for *argument*. If the recursive call returns True then store pointer of output variable *parseObject* in *argument* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting argument of application.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Argument of application must be a term.'.
  2. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
     - Class that inherits from class TERM.
       (a) Create APPLICATION object and store current application in *function* property of new APPLICATION object.
       (b) Call function PARSE of new APPLICATION object. Depending on the return value an action is taken:
           * True. Function returns True and store a pointer of *parseObject* of the recursive function call in *parseObject*.
           * False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
     - Otherwise. Return True and store pointer of current application in *parseObject*.
- TOINPUTLANGUAGE():STRING. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \ application { *function.toInputLanguage()* } { *argument.toInputLanguage()* }
- TOSTRING():STRING. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  function.toString() argument.toString().

A.2.3 Constant

The CONSTANT class implements the following part of the input language: Node Constant.

Inheritance relation Class CONSTANT inherits from class TERM and the class TERM inherits from JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *name*. (string). Name of constant.
- *parameters* (List of Term). Reference the parameters of the constant.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - CONSTANT. The variable *name* receives the value of *Constant.name*. Goto next step.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting ? sign of constant.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Constant must start with ? sign.'.
  2. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - PARENTHESIS. Goto next step.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting (sign of constant.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Parameter list of constant must be encapsulated by parenthesis.'.
  3. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
     - Class that inherits from class TERM. The object is stored on the last position of *parameters*. Call function PARSE for retrieved object. If the recursive call returns True then store pointer of output variable *parseObject* in object of *parameters* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.
     - HYPHEN. If type is equal to ENDBRACKET then call function GETNEXTTOKEN of *scanner* and goto step 5.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting parameter or end bracket of constant.'.
     - Otherwise. Return False with *errorMessage* equal to 'Parameter list of constant can only contain terms.'.

4. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
   - Hyphen. Depending on the type of the Hyphen object an action is taken:
     (a) endBracket. Goto step 5.
     (b) comma. Goto step 3. Only with the distinction that an Hyphen object is not allowed.
     (c) Otherwise. Return False with *errorMessage* equal to 'Parameters of constant must be separated by a comma.'.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting parameter or end bracket of constant.'.
   - Otherwise. Return False with *errorMessage* equal to 'Parameter list of constant can only contain terms.'.
5. Call getCurrentToken of scanner. Depending on the type of the return value of getCurrentToken an action is taken:
   - Class that inherits from class Term.
     (a) Create Application object and store current constant in *function* property of new Application object.
     (b) Call function parse of new Application object. Depending on the return value an action is taken:
       * True. Function returns True and store a pointer of *parseObject* of the recursive function call in *parseObject*.
       * False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
   - Otherwise. Return True and store pointer of current constant in *parseObject*.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  If *declarations.count* = 0
  Then \ constant { *name* } { { \emptyset{} } }
  Else \ constant { *name* }
  { { *parameters(0).toInputLanguage()* } ... { *parameters(n).toInputLanguage()* } }

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  ?*name*(*parameters(0).toString()*,...,*parameters(n).toString()*).

A.2.4 Context

The Context class implements the following part of the input language: node Context.

Inheritance relation Class Context inherits from class JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.
- *declarations* (List of declarations). List with all variable declarations of context. Each declaration has its own item in the list. The order of the items in the list, defines the order of declarations in the context.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
     - TEXT. Create DECLARATION object and call function PARSE of new DECLARATION object. Depending on the return value of the recursive function call an action is taken:
       * True. Store DECLARATION object on last position in *declarations*. Goto next step.
       * False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
     - EMPTY. Function returns True and store pointer of current context in *parseObject*.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Context can only contain variable declarations.'.
  2. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
     - HYPHEN. Depending on the type of the HYPHEN object an action is taken:
       * COMMA. Call function GETNEXTTOKEN and then Goto step 1, with the distinction that the EMPTY object is not allowed anymore.
       * Otherwise. Return False, with errorMessage equal to: 'Illegal character in context.'.
     - Otherwise. Return True and store pointer of current context in *parseObject*.
- TOINPUTLANGUAGE():STRING. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  If *declarations.count* = 0
  Then \ context { { \emptyset{} } }
  Else \ context { { *declarations(0).toInputLanguage()* } ... { *declarations(n).toInputLanguage()* } }
- TOSTRING():STRING. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  If *declarations.count* = 0
  Then empty
  Else *declarations(0).toString()*,...,*declarations(n).toString()*

A.2.5 Declaration

The DECLARATION class implements the following part of the input language: node Declaration.

Inheritance relation Class DECLARATION inherits from class JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *type* (Term). Reference to the root of the parse tree that defines the type of *variable*.
- *variable* (Variable). Reference to the variable that is declared in the current DECLARATION object.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - TEXT. Create new VARIABLE object with name equal to *Text.value* and store pointer in *variable*.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting variable of declaration.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Declaration must declare a variable.'.
  2. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - HYPHEN. Depending on the type of the HYPHEN object an action is taken:
       * COLON. Goto next step.
       * Otherwise. Return False, with errorMessage equal to: 'Variable of declaration must be followed by colon sign.'.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting : sign of declaration.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Variable of declaration must be followed by colon sign.'.
  3. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
     - Class that inherits from class TERM. A pointer to the object is stored in *type*. Call function PARSE for *body*. If the recursive call returns True then store pointer of output variable *parseObject* in *type* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.

   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting type of declaration.'.
   - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Type of declaration must be a term.'.
4. Return True and store pointer to current declaration in *parseObject*.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \ declaration { *variable.toInputLanguage()* } { *type.toInputLanguage()* }

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  variable.toString() : *type.toString()*.

A.2.6 DefinitionAxiom

The DefinitionAxiom class implements the following parts of the input language: node Definition and node Axiom.

Inheritance relation Class DefinitionAxiom inherits from JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.
- *context* (Context). See parent class.
- *constant* (Constant). See parent class.
- *isDefinition* (Bool). Indicates of the current object defines a definition or an axiom. If value is true, then the current object is a definition. Otherwise the object represents an axiom.
- *statement* (Statement). Reference to the root of the parse tree of the statement that belongs to the current Definition object.

Methods and/pr functions

- parse(in scanner:Scanner, out parseObject:JudgementObject, out errorMessage:string):bool. The function has the following code overview:

  1. Create for *context* new Context object.
  2. Call parse function for *context*. Based on the return value of the recursive function call an action is taken:
     - True. Pointer to *parseObject* of recursive function call is stored in *context* and goto next step.

- False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.

3. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
   - Hyphen. Based on the type of the Hyphen object an action is taken:
     * greaterSign. Goto next step.
     * Otherwise. Function returns False, with *errorMessage* equal to: 'Context of definition and axiom must be followed by > sign.'.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting > sign of definition or axiom.'.
   - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Context of definition and axiom must be followed by > sign.'.
4. Create for *constant* new Constant object.
5. Call parse function for *constant*. Based on the return value of the recursive function call an action is taken:
   - True. Check if output parameter is still of type Constant. If this is the case store pointer to *parseObject* in *constant* and goto next step. Otherwise return False, with errorMessage equal to: 'Definition or axiom must contain constant.'.
   - False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
6. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
   - Hyphen. Based on the type of the Hyphen object an action is taken:
     * assignment. Goto next step.
     * Otherwise. Function returns False, with *errorMessage* equal to: 'Constant of definition and axiom must be followed by := sign.'.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting := sign of definition or axiom.'.
   - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Constant of definition and axiom must be followed by := sign.'.
7. Call getCurrentToken of scanner. Depending on the type of the return value of getCurrentToken an action is taken:
   - Hyphen. Based on the type of the Hyphen object an action is taken:
     * axiomsign. Call getNextToken and goto step 8.
     * Otherwise. Goto 9.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting statement of definition or axiom.'.
   - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Goto step 9.

8. Note that current DefinitionAxiom object represents an axiom. To finish the axiom the following steps are necessary:
   (a) Set *isDefinition* to False.
   (b) Call getNextToken of Scanner. Depending on the type of the return value of getNextToken an action is taken:
      – Hyphen. Based on the type of the Hyphen object an action is taken:
         * colon. Goto next step.
         * Otherwise. Function returns False, with *errorMessage* equal to 'Axiom sign || must be followed by : sign.'.
      – EOF. Return False, with *errorMessage* equal to: 'Axiom sign || must be followed by : sign.'.
      – ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
      – Otherwise. Function returns False, with *errorMessage* equal to: 'Axiom sign || must be followed by : sign.'.
   (c) Call getCurrentToken of Scanner. Depending on the type of the return value of getCurrentToken an action is taken:
      – Class that inherits from class Term. Store pointer to retrieved object in *type*. And call function parse for *type*. If recursive function call returns True, then return True and store pointer of output variable *parseObject* in *type*. Otherwise return False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
      – EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting type of axiom.'.
      – ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
      – Otherwise. Function returns False, with *errorMessage* equal to: 'Type of axiom must be a term.'.
9. Note that current DefinitionAxiom object represents a definition. To finish the definition the following steps are necessary:
   (a) Set *isDefinition* to True.
   (b) Create for *statement* new Statement object.
   (c) Call function parse of *statement*. Depending on the return value of the function call an action is taken:
      – True. Return True.
      – False. Returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  If *isDefinition*
  Then \ definition { *context.toInputLanguage()* } { *constant.toInputLanguage()* }
  { *statement.toInputLanguage()* }

Else \ axiom { *context.toInputLanguage()* } { *constant.toInputLanguage()* } { *term.toInputLanguage()* }

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  If *isDefinition*
  Then *environment.toString()* > *constant.toString()* := *statement.toString()*
  Else *environment.toString()* > *constant.toString()* := || : *type.toString()*

A.2.7 EmptySet

The EmptySet class implements the following part of the input language: empty.

Inheritance relation Class EmptySet inherits from class JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.

Methods and/or functions No methods and/or functions.

A.2.8 Environment

The Environment class implements the following part of the input language: node Environment.

Inheritance relation Class Context inherits from class JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.
- *definitions* (List of DefinitionSuper). List with all the definitions and/or axioms defined in the current environment. Each definition and axioms has its own item in the list. The order of the items in the list, defines the order of definitions and axioms in the environment.

Methods and/or functions

- parse(in scanner:Scanner, out parseObject:JudgementObject, out errorMessage:string):bool. The function has the following code overview:
  1. Call getCurrentToken of scanner. Depending on the type of the return value of getCurrentToken an action is taken:
     - Empty. Call function getCurrentAdjacentToken. Depending on return value an action is taken:
       * Hyphen of type semicolon. Function returns True and store pointer of current context in *parseObject*.

        * Otherwise goto next step.
      - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting definition or axiom in environment.'.
      - ERRROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
      - Otherwise. Goto next step.
    2. Create DEFINITIONAXIOM object and call parse for new object. Depending on the return value an action is taken:
      - True. Store DEFINITIONAXIOM on last position in *definitions*. And goto next step.
      - False. Return False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
    3. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
      - HYPHEN of type COMMA. Goto step 2.
      - ERRROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
      - Otherwise. Return True.

- TOINPUTLANGUAGE():STRING. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  If *definitions.count* = 0
  Then \environment { { \emptyset{} } }
  Else \environment { { *definitions(0).toInputLanguage()* } ... { *definitions(n).toInputLanguage()* } }

- TOSTRING():STRING. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  If *definitions.count* = 0
  Then empty
  Else *definitions(0).toString()*, ... , *definitions(n).toString()*

A.2.9 EOF

The class EOF is used by the SCANNER object. The SCANNER reads an input judgement in string representation and breaks the string into tokens. These tokens are used to create a parse tree. The SCANNER returns an EOF object if the complete input string is read. The class EOF has no other functionality.

Inheritance relation Class EOF inherits from class JUDGEMENTOBJECT.

Attribute

- *classType* (ClassType). For description see parent class.

Methods and/or functions No methods and/or functions.

A.2.10 ErrorObject

When constructing a parse tree of an input judgement in string representation (class PARSER) the SCANNER class identifies the different tokens of the input judgement. This is done with the function GETNEXTTOKEN or GETCURRENTTOKEN of the SCANNER class. The different tokens are described in table A.1. If an character sequence can not be identified to any defined token an object of type ERROROBJECT is returned. This states, that there is an illegal character sequence in the input judgement.

Inheritance relation Class ERROROBJECT inherits from class JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *message*. In the variable an error message is stored.

Methods and/or functions No methods and/or functions.

A.2.11 Hyphen

The class HYPHEN is used for all the hyphen signs in the input language. In the input language the following signs are defined as hyphen:

- ,
- .
- >
- ;
- :
-)
- ||
- |−
- :=

Inheritance relation Class HYPHEN inherits from class JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *type* (TokenType). Variable determines the type of the hyphen. In the introduction of this paragraph the different hyphens are denoted.

Methods and/or functions No methods and/or functions.

A.2.12 Judgement

The JUDGEMENT class implements the following part of the input language: node Judgement.

Inheritance relation Class JUDGEMENT inherits from class JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *context* (Context). Reference to the root of the parse tree of the context, that belong to the current JUDGEMENT object.
- *environment* (Environment). Reference to the root of the parse tree of the environment, that belongs to the current JUDGEMENT object.
- *statement* (Statement). Reference to the root of the parse tree of the statement, that belongs to the current judgement.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Create for *environment* new ENVIRONMENT object.
  2. Call function parse for *environment*. Based on the return value of the recursive function an action is taken:
     - True. Goto next step.
     - False. Function returns False and *errorMessage* receives the value of *errorMessage* of the recursive function call.
  3. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - HYPHEN. Depending on the type of the HYPHEN object an action is taken:
       * SEMICOLON. Goto next step.
       * Otherwise. Return False, with errorMessage equal to: 'Environment of judgement must be followed by ';' sign.'.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting ; sign between environment and context of judgement.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Environment of judgement must be followed by ';' sign.'.
  4. Create for *context* new CONTEXT object.
  5. Call function parse for *context*. Based on the return value of the recursive function an action is taken:
     - True. Goto next step.

- False. Function returns False and *errorMessage* receives the value of *errorMessage* of the recursive function call.

6. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
   - Hyphen. Depending on the type of the Hyphen object an action is taken:
     * vdash. Goto next step.
     * Otherwise. Return False, with errorMessage equal to: 'Context of judgement must be followed by |- sign.'.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting |- sign between context and statement of judgement.'.
   - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Context of judgement must be followed by |- sign.'.
7. Create for *statement* new Statement object.
8. Call function parse for *statement*. Based on the return value of the recursive function an action is taken:
   - True. Goto next step.
   - False. Function returns False and *errorMessage* receives the value of *errorMessage* of the recursive function call.
9. Return True.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \judgement { *environment.toInputLanguage()* } { *context.toInputLanguage()* } { *statement.toInputLanguage()* }

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  environment.toString() ; *context.toString()* |− *statement.toString()*.

A.2.13 JudgementObject

The class JudgementObject is an abstract class and is designed to make the program typed. All the classes that inherit from this class are used to constructed a judgement in tree structure. Therefore the functions getNextToken and getCurrentToken of class Scanner return an object of type JudgementObject. The class JudgementObject has no other use.

Inheritance relation The classes Context, Declaration, DefinitionAxiom EmptySet, Environment ErrorObject, Judgement, Statement and Term inherit of JudgementObject.

Attributes

- *classtype* (ClassTypes). In the attribute the class type is stored. With the value of *classtype*, it is easily to typecast the current JudgementObject object to one of the classes that have an inheritance relation with JudgementObject.

Methods and/or functions No methods and/or functions.

A.2.14 KindCat

The KindCat class implements the following part of the input language: # of node Term.

Inheritance relation Class KindCat inherits from class Term and the class Term inherits from JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.

Methods and/or functions

- parse(in scanner:Scanner, out parseObject:JudgementObject, out errorMessage:string):bool. The function has the following code overview:

  1. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
     - KindCat. Goto next step.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting # of kindcat.'.
     - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'KindCat object must be defined by # sign.'.
  2. Return True and store pointer of current kindcat in *parseObject*.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \square{}

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  #

A.2.15 ParenthesisExpression

The ParenthesisExpression class implements the following part of the input language: *(Term)* of node Term.

Inheritance relation Class ParenthesisExpression inherits from class Term and the class Term inherits from JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.
- *endParenthesisIsRead* (Bool). True if end bracket is read by parser. Otherwise the value is false.
- *term* (Term). Term that is embraced between parenthesis.

Methods and/or functions

- parse(in scanner:Scanner, out parseObject:JudgementObject, out errorMessage:string):bool. The function has the following code overview:
  1. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
     - ParenthesisExpression. Goto next step.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found, when expecting (sign of parenthesis expression.'.
     - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Parenthesis expression must start with (.'.
  2. Call getCurrentToken of scanner. Depending on the type of the return value of getCurrentToken an action is taken:
     - Class that inherits from class Term. A pointer to the object is stored in *term*. Call function parse for *term*. If the recursive call returns True then store pointer of output variable *parseObject* in *term* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting term of parenthesis expression.'.
     - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Expression embraced by parenthesis must be a Term.'.
  3. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
     - Hyphen. Depending on the type of the Hyphen object an action is taken:
       * endparenthesis. Goto next step.
       * Otherwise. Return False, with *errorMessage* equal to: 'Parenthesis expression must have end parenthesis.'.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting end bracket of parenthesis expression.'.
     - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.

- Otherwise. Return False with *errorMessage* equal to 'Parenthesis expression must have end parenthesis.'.

4. Call getCurrentToken of scanner. Depending on the type of the return value of getCurrentToken an action is taken:
   - Class that inherits from class Term.
     (a) Create Application object and store current parenthesis expression in *function* property of Application object.
     (b) Call function parse of Application object. Depending on the return value an action is taken:
         * True. Function returns True and store a pointer of *parseObject* of the recursive function call in *parseObject*.
         * False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  term.toInputLanguage()

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  If *bEndBracketIsRead*
  Then (*term.toString()*)
  Else (*term.toString()*

A.2.16 Parser

The parser class is used to compile an input judgement. The input judgement is syntactically correct if it satisfies the grammar described in requirement A.1.1. If the input judgement is syntactically correct, the Parser class will translate the input judgement to the language needed for the λD^+ proof checker.

Attributes No attributes.

Methods and/or functions

- translateInputJudgement(in inputJudgement:string, out translation:string, out errorMessage:string):bool. The function returns True, if *inputJudgement* belongs to the grammar described in requirement A.1.1. If this is not the case then the function returns False. If the *inputJudgement* is syntactically correct the translation of *inputJudgement* is stored in *translation*. The translation value can be used as input for the λD^+ proof checker. If the input judgement is not syntactically correct an error message is stored in *errorMessage*.

  The function has the following code overview:

  1. Create new Judgement object and store pointer in method variable *rootParseTree*.

2. Call function PARSE of *rootParseTree*. Based on the return value of the function call an action is taken:
   - True. Goto next step.
   - False. Return False, with *errorMessage* equal to: output attributes *errorMessage* of function call plus 'Judgement parse till:' + *rootParseTree.toString*.
3. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
   - EOF. Goto next step.
   - Otherwise. Return False, with *errorMessage* equal to: 'Judgement is finished but there are still tokens left.'.
4. Store in *translation* the value *rootParseTree.toInputLanguage*.
5. Return True.

A.2.17 Scanner

Attributes

- *judgement* (string). Input judgement that is scanned for tokens.
- *position* (int). Position of lexical analyzer. The characters on position zero till *position* are already scanned for tokens. If the function GETNEXTTOKEN or GETCURRENTTOKEN is called the lexical analyzer reads *judgement* starting from the character on position *position*.

Methods and/or functions

- GETCURRENTADJACENTTOKEN. Function returns the second available token in *judgement* starting from *position*, without changing the value of *position*. The result is that executing GETCURRENTADJACENTTOKEN multiple times, will always result in the same return value. The function has the following code overview:
  1. Store value of *position* in *currentPosition*.
  2. Call function GETNEXTTOKEN.
  3. Call function GETNEXTTOKEN and store return value in *returnValue*.
  4. Set *position* to *currentPosition*.
  5. Return *returnValue*.
- GETCURRENTTOKEN():JUDGEMENTOBJECT. Function returns the next token in *judgement* starting from *position*, without changing the value of *position*. The result is that executing GETCURRENTTOKEN multiple times, will always result in the same return value. The function has the following code overview:
  1. Store value of *position* in *currentPosition*.
  2. Call function GETNEXTTOKEN and store return value in *returnValue*.
  3. Set *position* to *currentPosition*.

4. Return *returnValue*.

- getNextToken():JudgementObject. Function returns the next token in *judgement* starting from *position*. After executing the function *position* will be updated. The result is that by calling getNextToken multiple times the complete input judgement *judgement* is read. The return values of the function are shown in table A.1

Character sequence	Token type
.	Hyphen of type dot.
:	Hyphen of type colon.
,	Hyphen of type comma.
)	Hyphen of type endbracket.
>	Hyphen of type greatersign.
||	Hyphen of type axiomsign.
|−	Hyphen of type vdash.
\	Abstraction.
@	TypeBinder.
*	TypeCat.
#	KindCat.
(	ParenthesisExpression.
? ([a...z] + [A...Z] + [0...9] + -)$^+$.	Constant
([a...z] + [A...Z] + [0...9] + -)$^+$.	Variable
Otherwise	ErrorObject

Table A.1: Return values of function getNextToken of class Scanner.

A.2.18 Statement

The Statement class implements the following part of the input language: node Statement.

Inheritance relation Class Statement inherits from class JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.
- *subject* (Term). Reference to the root of the parse tree that describes the subject of the current statement.
- *type* (Term). Reference to the root of the parse tree that describes the type of the current statement.

Methods and/or functions

- parse(in scanner:Scanner, out parseObject:JudgementObject, out errorMessage:string):bool. The function has the following code overview:

1. Call getCurrentToken of scanner. Depending on the type of the return value of getCurrentToken an action is taken:

- Class that inherits from class TERM. A pointer to the object is stored in *subject*. Call function PARSE for *subject*. If the recursive call returns True then store pointer of output variable *parseObject* in *subject* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.
- EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting subject of statement.'.
- ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
- Otherwise. Return False with *errorMessage* equal to 'Subject of statement must be a term.'.

2. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
   - HYPHEN. Depending on the type of the HYPHEN object an action is taken:
     * COLON. Goto next step.
     * Otherwise. Return False, with errorMessage equal to: 'Subject of statement must be followed by colon.'.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting : sign between subject and type of statement.'.
   - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Subject of statement must be followed by colon.'.
3. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
   - Class that inherits from class TERM. A pointer to the object is stored in *subject*. Call function PARSE for *subject*. If the recursive call returns True then store pointer of output variable *parseObject* in *subject* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting type of statement.'.
   - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Type of statement must be a term.'.

- TOINPUTLANGUAGE():STRING. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \statement{ *subject.toInputLanguage()* } { *type.toInputLanguage()* }

- TOSTRING():STRING. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  subject.toString() : *type.toString*

A.2.19 Term

The TERM class implements the following part of the input language: node Term. The class is an abstract class.

Inheritance relation

- Class TERM inherits from class JUDGEMENTOBJECT.
- Classes ABSTRACTION, APPLICATION, CONSTANT, KINDCAT, TYPEBINDER, TYPECAT and VARIABLE inherit from TERM.

Attributes

- *classType* (ClassType). For description see parent class.

Methods and/or functions

- abstract PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL.

A.2.20 TypeBinder

The TYPEBINDER class implements the following part of the input language: *@Declaration.Term* of the node Term.

Inheritance relation Class TYPEBINDER inherits from class TERM and the class TERM inherits from JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *body*. Reference to the body of the parse tree that describes the body of the type binder.
- *declaration*. Reference to the root of the parse tree that describes the declaration of the type binder.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - TYPEBINDER. Goto next step.
     - EOF. Return False with *errorMessage* equal to 'EOF found when expecting @ sign of typebinder.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.

- Otherwise. Return False with *errorMessage* equal to 'Typebinder must start with @ sign'.

2. Create for *declaration* new DECLARATION object.
3. Call PARSE function for *declaration*. Based on the return value of the recursive function call an action is taken:
   - True. Pointer to *parseObject* of recursive function call is stored in *declaration* and goto next step.
   - False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
4. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
   - HYPHEN. Based on the type of the HYPHEN object an action is taken:
     * DOT. Goto next step.
     * Otherwise. Function returns False, with *errorMessage* equal to: 'Declaration of typebinder must be followed by a dot.'.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting dot sign between declaration and body of typebinder.'.
   - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Declaration of typebinder must be followed by a dot.'.
5. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
   - Class that inherits from class TERM. A pointer to the object is stored in *body*. Call function PARSE for *body*. If the recursive call returns True then store pointer of output variable *parseObject* in *body* and goto next step. Otherwise return False and *errorMessage* receives the value of the *errorMessage* of the recursive call.
   - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting body of typebinder.'.
   - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
   - Otherwise. Return False with *errorMessage* equal to 'Body of typebinder must be a term.'.
6. Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
   - Class that inherits from class TERM.
     (a) Create APPLICATION object and store current abstraction in *function* property of APPLICATION object.
     (b) Call function PARSE of APPLICATION object. Depending on the return value an action is taken:
       * True. Function returns True and store a pointer of *parseObject* of the recursive function call in *parseObject*.
       * False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.

  - Otherwise. Return True and store pointer of current abstraction in *parseObject*.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \typebinder { *declaration.toInputLanguage()* } { *body.toInputLanguage()* }

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  @ *declaration.toString(). body.toString()*

A.2.21 TypeCat

The TypeCat class implements the following part of the input language: $*$ of the node Term.

Inheritance relation Class TypeCat inherits from class Term and the class Term inherits from JudgementObject.

Attributes

- *classType* (ClassType). For description see parent class.

Methods and/or functions

- parse(in scanner:Scanner, out parseObject:JudgementObject, out errorMessage:string):bool. The function has the following code overview:

  1. Call getNextToken of scanner. Depending on the type of the return value of getNextToken an action is taken:
     - TypeCat. Goto next step.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting $*$ sign of typecat.'.
     - ErrorObject. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'TypeCat object must be defined by $*$ sign.'.
  2. Return True and store pointer of current typecat in *parseObject*.

- toInputLanguage():string. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \ast{}

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  *

A.2.22 Variable

The VARIABLE class implements the following part of the input language: node Variable.

Inheritance relation Class VARIABLE inherits from class TERM and the class TERM inherits from JUDGEMENTOBJECT.

Attributes

- *classType* (ClassType). For description see parent class.
- *name* (string). Internal name of variable. The internal name, can change during the correctness check. In the correctness check, *name* is renamed per derivation step. The renaming is used to ensure the Barendregt convention, during the correctness check.

Methods and/or functions

- PARSE(IN SCANNER:SCANNER, OUT PARSEOBJECT:JUDGEMENTOBJECT, OUT ERRORMESSAGE:STRING):BOOL. The function has the following code overview:
  1. Call GETNEXTTOKEN of SCANNER. Depending on the type of the return value of GETNEXTTOKEN an action is taken:
     - VARIABLE. Set *name* equal to *name* property of VARIABLE object and goto next step.
     - EOF. Return False, with *errorMessage* equal to: 'EOF found when expecting name of variable.'.
     - ERROROBJECT. Return False and *errorMessage* is equal to *ErrorObject.message*.
     - Otherwise. Return False with *errorMessage* equal to 'Variable must be defined by a name.'.
  2. Return True and store pointer of current kindcat in *parseObject*.
- Call GETCURRENTTOKEN of SCANNER. Depending on the type of the return value of GETCURRENTTOKEN an action is taken:
  - Class that inherits from class TERM.
    1. Create APPLICATION object and store current variable in *function* property of APPLICATION object.
    2. Call function PARSE of APPLICATION object. Depending on the return value an action is taken:
       * True. Function returns True and store a pointer of *parseObject* of the recursive function call in *parseObject*.
       * False. Function returns False, with *errorMessage* equal to the output attribute *errorMessage* of the recursive function call.
  - Otherwise. Return True and store pointer of current variable in *parseObject*.
- TOINPUTLANGUAGE():STRING. Function is a pretty printer and is used to translate the grammar described in requirement A.1.1 to the input language of the λD^+ proof checker. The return value has the following format:
  \variable{ *name* }

- toString():string. Function is a pretty printer, the output satisfies the grammar described in requirement A.1.1. The return value has the following format:
  name