A proof assistant based on terms with binding structures

Smeijers, F.A.M.

Award date:
2009

Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology

Master's Thesis

A Proof Assistant Based on Terms with Binding Structures

F.A.M. Smeijers
August 2009

Supervisors:
dr.ir. C. Hemerik
dr.ir. M.G.J. Franssen

Abstract

A proof assistant is a computer program that is used for proving theorems in an interactive way. Many proof assistants are based on the theory of pure type systems and the propositions as types principle. During this master project a proof assistant has been developed that has such a theoretic foundation, but instead of using the ordinary pure type system framework, it uses a pure type system framework that is extended with additional type constructors. Initially, it was supposed to be implemented by using the FoolProof components to test their usability. One of FoolProof main features is its ability to manipulate terms with binding structures. When it turned out those components were not available in time, the decision was made to use the infrastructure of Cocktail instead, which manipulates terms with binding structures in the same way as FoolProof.

Acknowledgements

I am grateful to my supervisor Kees Hemerik for letting me start immediately on my master project when I asked him to be my supervisor. It did not take him long to come up with a research subject that I liked to investigate. When the progress of the project stalled due to the unavailability of the components that this project depended on, he offered me a solution and introduced me to Michael Franssen. I would also like to thank Michael for discussing his proof assistant implementation with me and guiding me through its source code, which takes a great mind to understand at times. I enjoyed the time working on this project and it was over before I knew it.

Frenkel Smeijers

Contents

Part I

Introduction and theoretic background

Chapter 1

Introduction

A proof assistant is a computer program that lets the user construct proofs of theorems in an interactive way. It does not create a proof automatically. The logics we are interested in are undecidable, hence not all problems expressed in these logics can be solved automatically, so user interaction is necessary. The user must supply the intelligence to guide the program, while the proof assistant does the bookkeeping and makes sure that the user does not perform any invalid actions. This way only proofs that are correct will be constructed.

A proof assistant can be based on the theory of pure type systems (PTSs) [1] and the propositions as types principle [13]. The PTS framework allows the description of various typed lambda calculi. By setting a few parameters in the framework in a specific way, one gets the simply typed lambda calculus [6]. And by using other parameters, one gets more expressively typed lambda calculi with polymorphic and dependent types.

The propositions as types principle, also known as Curry-Howard-De Bruijn isomorphism, states that theorems can be encoded as types. Then finding a proof of a certain theorem is the same as creating a term of a certain type. This term is called the proof object. Although it is usually hard for humans to read and understand a proof object, computers can use it to reconstruct the whole proof derivation, which is more readable for humans.

During this master project a proof assistant has been developed that is based on the theory of PTSs and the propositions as types principle. The PTS framework is extended with additional type constructors [16] that have a one-on-one correspondence to the operators in predicate logic due to the propositions as types principle.

Proof assistants that are based on PTSs and the propositions as types principle basically perform structure editing of terms with bound variables. It would be nice to have a component toolkit that could facilitate manipulating terms with binding structures. FoolProof [11] is a component library created for this purpose. One of the goals of this master project was to build a proof assistant using the FoolProof components to test their usability.

Unfortunately, the FoolProof components were not available in time. So the choice was made to use the infrastructure of Cocktail [10] instead. Cocktail is a tool for deriving correct programs. It includes a proof assistant that is based on a specific instance of the PTS framework without the additional type constructors. The infrastructure of Cocktail could be used, because it has facilities for structure editing with bound variables, just like FoolProof.

1.1 Problem description

The original problem description of this project can be found in Appendix A and is in Dutch. Here is a translation in English:

"FoolProof is a component library for the manipulation of formal languages with binding structures within the Delphi development environment. FoolProof contains (or will contain) components for lexical scanning, syntax highlighting, parsing, tree building, structure editing, textual views, structural views and context management. A characteristic difference with other environments are the facilities for manipulating terms with binding structures, such as copying, substitution, unification, etc. This makes the FoolProof environment particularly suitable for formal languages like lambda calculi, logics, programming and specification languages.

This master project concerns the development of a proof assistant that is based on type theory and the propositions as types principle (like Coq and Cocktail). In essence these types of proof assistants are based on structure editing of typed lambda terms, although the terms themselves are usually not visible. The proof assistant will use the available FoolProof components where possible. It is also a study on the usefulness of these components for the given task. It is explicitly not the intention to develop an automatic prover. The proof assistant will be designed such that it can be incorporated as one or more components in the FoolProof system."

The first part of this master project consisted of studying the theory of proof assistants, PTSs and the propositions as types principle. The second part was to develop an abstract model of a proof assistant system. The basis for the model that is used in this project, came from an internship report by Marco Brassé [3]. His model needed to be reworked, because it was not really suited to an interactive implementation. The third part was to implement the abstract model using the FoolProof components.

Regrettably, halfway through the project it turned out the FoolProof components were not going to be ready in time. So the subject of this master project was changed such that instead of using the FoolProof components, the infrastructure of Cocktail was used, because it allows manipulation of terms with binding structures in the same way as FoolProof.

The requirements of the proof assistant include:

- The tool is interactive. It is based on PTSs with additional type constructors and the propositions as types principle.
- Given a context, a term and a type, the tool is able to check whether the term has the given type in the given context.
- Given a context and a term, the tool can compute the type of the term.
- Given a context and a type, the tool is able to assist the user in constructing a term that has the given type.
- The context can be extended with definitions and declarations, such that its well-formedness is maintained.
- The available type constructors can be enabled and disabled.
- The parameters of the PTS used by the system can be changed.

- These settings can be saved to and loaded from a file.
- Theorems can be saved to and loaded from a file.
- The user can interact with the system through text commands and widgets like buttons and menus.
- The proof assistant has a hybrid interface: the context, terms and types can be viewed as unstructured text, structured text and graphically.

1.2 Thesis outline

The way this master project was carried out, dictates the structure of this document:

The first part of this thesis describes the theoretic background. Chapter 2 contains a description of what a proof assistant is and describes a couple of existing proof assistants. Chapter 3 is about lambda calculi. This includes untyped lambda calculus, typed lambda calculi, ordinary PTSs and PTSs with additional type constructors. Chapter 4 describes the propositions as types principle.

The second part of this thesis contains an abstract model for a proof assistant. Chapter 5 gives a description of how terms can be represented. A type checker to check the type of a term is described in Chapter 6. Chapter 7 contains tactics to construct a term for a given type.

The final part of this thesis is about an implementation of the abstract model from the previous part. Chapter 8 describes the general design of the system. Chapter 9 till 13 describe the implementation details of the term representation, type checker, tactics, structure editor and the user interface. Each part of the system has its own chapter. In these chapters it is described how the abstract model maps to the implementation, what could be taken from Cocktail without modification, what changes had to be made to the Cocktail source code and what was created from scratch.

The last chapter contains an evaluation and presents some ideas for future work.

Finally, the appendix contains the original problem description as it was at the beginning of this project, written in Dutch.

Chapter 2

Proof assistant

A proof assistant is a computer program that helps the user create correct proofs of theorems in an interactive way. The user tells the proof assistant what to do while the proof assistant makes sure that every step taken is allowed. The program keeps track of what remains to be done before a proof is finished. This way only correct proofs are constructed.

Proof assistants are also know as interactive theorem provers. The opposite of an interactive theorem prover is an automated theorem prover. This kind of computer program tries to construct a proof of a conjecture that the user has entered automatically without user involvement. An automated theorem prover is usually based on a first order logic, because for these logics good automation is possible. For higher order logics this is more difficult.

Typed lambda calculi and the propositions as types principle are the subjects of the next two chapters. Proof assistants are usually based on these two concepts, because they allow theorems to be encoded as types and proofs as terms. Unfinished proofs are represented by terms that contain typed holes. Holes represent parts of the proof that still have to be proved. We call these holes goals. When all holes have been filled with terms that have the correct type, a proof is finished. A finished proof is verified by computing the type of the term. When the type is equivalent to the encoded theorem, the proof is correct. The term that encodes the proof is called the proof object. When someone wants to validate the result of a proof assistant or its type checker, the type of the proof object can be computed manually or by another type checker. Computing the type of a proof object reconstructs the whole proof derivation and theorem.

2.1 Other proof assistants

During this master project a number of existing proof assistants were studied. In this section a couple of those tools are described, including their strengths and points for improvement.

2.1.1 Coq

Coq [7] is a proof assistant that is the result of more than twenty years of research. Development of the tool started in 1984 by Thierry Coquand and Gérard Huet at INRIA. In the first few years it was based on the calculus of constructions [9]. Since 1991 the calculus is extended to the calculus of inductive constructions [8]. Coq allows the user to define functions and predicates. It comes with libraries for reasoning with, among others, natural numbers,

integers, rational numbers, lists and finite sets. Coq is written in the programming language Objective Caml with a bit of C. Its source code can be downloaded from its website.

User interaction with Coq is text-based, but the proof assistant comes with a graphical user interface. Third party user interfaces for Coq are available. The graphical user interface of Coq has a menu in which all commands can be found. It is also possible to click on the current goal to get a list of often used commands. This list does not depend on the structure of the term that is clicked on, so it contains commands that cause an error when the user tries to use it. Another weak point of Coq is that when an unfinished proof contains multiple holes it seems to be impossible to select another goal than the goal Coq has selected.

2.1.2 Cocktail

Cocktail [10] is a tool for deriving correct programs from their specification through stepwise refinement. It was developed by Michael Franssen during his PhD research at the Eindhoven University of Technology in 2000. Cocktail contains an interactive and an automated theorem prover, but for this master project only the interactive theorem prover is of interest. Cocktail is based on a typed lambda calculus that models first order logic closely. It is not based on higher order logics, because they are not needed for Cocktail's purpose and good automation for higher order theorems is difficult.

Cocktail is written in Java. It has a graphical user interface that brings the number of times the user has to enter commands through the keyboard to a minimum. Proof derivations are graphically represented in the flag notation and are constructed through drag-and-drop operations and by clicking on buttons and parts of the proof. Cocktail supports both forward and backward reasoning.

Chapter 3

Lambda calculi

Proof assistants can be based on the theory of pure type systems (PTSs) and the propositions as types principle. In this chapter we describe the notation we use to write down PTSs and the propositions as types principle is described in the next chapter. For a more in depth description of PTSs we refer the reader to Barendregt's document on lambda calculi with types [1]. The PTS framework is a method to describe a number of typed lambda calculi. But before we discuss typed lambda calculi, we give a short description of the untyped variant.

3.1 Untyped lambda calculus

The untyped lambda calculus is a formal system designed in the 1930's by Alonzo Church [4, 5]. He wanted to use it as a foundation for a formal theory of mathematics. The calculus defines the input-output behavior of functions in the most abstract view. In the untyped lambda calculus there is a set of variables and the term construction principles abstraction and application to create lambda terms, and a 'calculation rule' called reduction, which uses the notion of substitution. These concepts are explained in the next section where we extend the system with types, because the untyped lambda calculus has some drawbacks that can be resolved by adding types. These drawbacks include counter-intuitive self-applications, terms without normal forms that represent infinite calculations and fixed points for every term, which is in contrast to the usual behavior of functions.

3.2 Pure type systems

In typed lambda calculus there is the notion of typing judgment, which is formally written as $\Gamma \vdash M : \tau$, where Γ is a context and M and τ are terms. It should be read as: "In a context Γ term M has type τ." To clarify what this statement means, we define terms and contexts. We also need a set of type derivation rules to decide whether a typing judgment holds. These type derivation rules depend on how the PTS framework [1] is instantiated.

The set $\mathcal{T}$ of pseudo-terms is defined via the following abstract syntax:

$$\mathcal{T} ::= \mathcal{S} \mid V \mid \lambda V{:}\mathcal{T}.\mathcal{T} \mid \Pi V{:}\mathcal{T}.\mathcal{T} \mid \mathcal{T}\,\mathcal{T}$$

$\mathcal{S}$ is a set of sorts and V is a set of variables. $\lambda V{:}\mathcal{T}.\mathcal{T}$ is used for constructing abstractions over terms. For example, when x is a variable and τ and M are terms, then the term $(\lambda x{:}\tau.\, M)$

is the abstraction of x over M and τ is the type of x. $(\Pi V{:}\mathcal{T}.\mathcal{T})$ is used for constructing abstractions over types. $\mathcal{T}\,\mathcal{T}$ is used for function applications. For example, when f and a are terms then the term $f\,a$ is the application of function f on argument a.

Reduction is defined as:

$$(\lambda x{:}\tau.M)\,N > M[x := N].$$

In words this is: "Applying function $(\lambda x{:}\tau.M)$ on argument N reduces to M in which every occurrence of x is replaced by N.". $M[x := N]$ is defined as:

$$\begin{array}{rcl} x[x := N] & = & N \\ y[x := N] & = & y \\ (\lambda y{:}A.B)[x := N] & = & (\lambda y{:}A[x := N].B[x := N]) \\ (\Pi y{:}A.B)[x := N] & = & (\Pi y{:}A[x := N].B[x := N]) \\ (A\,B)[x := N] & = & (A[x := N])\,(B[x := N]) \end{array}$$

We obey Barendregt's variable convention [1], which has the consequence that the variable y does not occur in N in $(\lambda y{:}A.B)[x := N]$ and $(\Pi y{:}A.B)[x := N]$.

A context is a list that has the structure $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$, where x_i is a variable and τ_i is a term for $i = 1, \ldots, n$. $x_i : \tau_i$ means that x_i is of type τ_i. The empty context is written as $<>$.

A PTS is specified by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R}^{\Pi})$ where $\mathcal{S}$ is again a set of sorts, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of axioms and $\mathcal{R}^{\Pi} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of rules that specify which Π-types may be formed. An axiom has the form $(s_1{:}s_2)$ with $s_1, s_2 \in \mathcal{S}$ and a rule has the the form (s_1, s_2, s_3) with $s_1, s_2, s_3 \in \mathcal{S}$.

By instantiating these parameters a specific type system is formed. For example, the simply typed lambda calculus [6] can be described as a PTS when the parameters are set as follows: $\mathcal{S} = \{*, \Box\}$, $\mathcal{A} = \{(* : \Box)\}$ and $\mathcal{R}^{\Pi} = \{(*, *, *)\}$. And in the same way, the calculus of constructions [9] can be written as a PTS when the parameters are set as: $\mathcal{S} = \{*, \Box\}$, $\mathcal{A} = \{(* : \Box)\}$ and $\mathcal{R}^{\Pi} = \{(*, *, *), (\Box, *, *), (*, \Box, \Box), (\Box, \Box, \Box)\}$.

Now that we have terms, contexts and PTS specifications, we give the type derivation rules that are used to compute the type of a given term:

(axiom) $$<> \vdash s_1 : s_2 \qquad \text{if } (s_1 : s_2) \in \mathcal{A}$$

(start) $$\frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad \text{if } x \notin \Gamma$$

(weaken) $$\frac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x{:}A \vdash B : C} \qquad \text{if } x \notin \Gamma$$

(Π-form) $$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^{\Pi}$$

$$(\Pi\text{-intro}) \quad \frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash (\lambda x{:}A.\, b) : (\Pi x{:}A.\, B)}$$

$$(\Pi\text{-elim}) \quad \frac{\Gamma \vdash F : (\Pi x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B[x := a]}$$

$$(\text{conv}) \quad \frac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1 \simeq \tau_2}{\Gamma \vdash M : \tau_2}$$

In these rules s ranges over $\mathcal{S}$, $x \notin \Gamma$ means that $x : \tau$ does not appear in Γ for any τ and $\tau_1 \simeq \tau_2$ means that τ_1 and τ_2 are equal modulo conversion.

3.3 Additional type constructors

So far, the only type constructor we have seen is the Π-type constructor. In this section we extend the PTS framework with additional type constructors: $\rightarrow, \times, +$ and Σ [16]. To accommodate the PTS framework for these new type constructors we extend the PTS specification to a 7-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}^{\Pi}, \mathcal{R}^{\rightarrow}, \mathcal{R}^{\times}, \mathcal{R}^{+}, \mathcal{R}^{\Sigma})$ where $\mathcal{S}, \mathcal{A}, \mathcal{R}^{\Pi}$ are the same as in the ordinary PTS framework and $\mathcal{R}^{\rightarrow}, \mathcal{R}^{\times}, \mathcal{R}^{+}, \mathcal{R}^{\Sigma} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ are sets of rules for specifying which $\rightarrow, \times, +$ and Σ-types may be formed. These new types can be encoded using only the Π-type constructor, but this requires higher order logic and we do not want to restrict the user to use higher order logic.

Each new type constructor has its own subsection in which we extend the abstract syntax for the set of pseudo-terms and the rules for reduction, substitution and type derivation.

3.3.1 $\rightarrow$-types

$\rightarrow$-types represent function types. For example, $A{\rightarrow}B$ is the set of all functions from A to B. Usually in PTSs $A{\rightarrow}B$ is just an abbreviation for $\Pi x{:}A.\, B$ where x does not appear in B. But in the proof assistant that is developed during this master project we distinguish between these two type constructors, because if we want to disable one type constructor we do not automatically also want to forbid the use of the other one.

The abstract syntax for the set $\mathcal{T}$ of pseudo-terms is extended with one production alternative:

$$\mathcal{T} ::= \ldots \mid \mathcal{T}{\rightarrow}\mathcal{T}$$

The reduction mechanism is extended with the following rules:

$$\begin{array}{ll} A{\rightarrow}B > (\Pi x{:}A.\, B) & \text{with } x \notin B \\ (\Pi x{:}A.\, B) > A{\rightarrow}B & \text{if } x \notin B \end{array}$$

Because the abstract syntax is extended with only one production alternative, the substitution rules are also extended with only one rule:

$$(A{\rightarrow}B)[x := N] \quad = \quad (A[x := N]){\rightarrow}(B[x := N])$$

The type derivation rules are extended with:

$$(\to\text{-form}) \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2}{\Gamma \vdash A \to B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^{\to}$$

$$(\to\text{-intro}) \quad \frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash A \to B : s}{\Gamma \vdash (\lambda x{:}A.\, b) : A \to B}$$

$$(\to\text{-elim}) \qquad \frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B}$$

3.3.2 $\times$-types

$A \times B$ is the Cartesian product of A and B. The pair of $a : A$ and $b : B$ is denoted by $\langle a, b \rangle$ and has type $A \times B$. Taking the first element of $\langle a, b \rangle$ is denoted by $\pi_1(\langle a, b \rangle)$, which reduces to a. Taking the second element goes in a similar manner.

The abstract syntax for the set $\mathcal{T}$ of pseudo-terms is extended in the following way:

$$\mathcal{T} ::= \ldots \mid \langle \mathcal{T}, \mathcal{T} \rangle \mid \mathcal{T} \times \mathcal{T} \mid \pi_1(\mathcal{T}) \mid \pi_2(\mathcal{T})$$

The reduction rules are extended with the following rules:

$$\pi_1(\langle a, b \rangle) > a$$
$$\pi_2(\langle a, b \rangle) > b$$

The substitution rules are extended with the following rules:

$$\begin{array}{rcl} (A \times B)[x := N] & = & (A[x := N]) \times (B[x := N]) \\ \langle A, B \rangle[x := N] & = & \langle A[x := N], B[x := N] \rangle \\ (\pi_1(A))[x := N] & = & \pi_1(A[x := N]) \\ (\pi_2(A))[x := N] & = & \pi_2(A[x := N]) \end{array}$$

The type derivation rules are extended with the following rules:

$$(\times\text{-form}) \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2}{\Gamma \vdash A \times B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^{\times}$$

$$(\times\text{-intro}) \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma \vdash A \times B : s}{\Gamma \vdash \langle a, b \rangle : A \times B}$$

$$(\times\text{-elim}_1) \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1(M) : A}$$

$$(\times\text{-elim}_2) \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2(M) : B}$$

3.3.3 +-types

$A{+}B$ is the sum of A and B. It is also known as the disjoint union. When A and B are sets, then $A{+}B$ is defined as $\{(1,a)\,|\,a \in A\} \cup \{(2,b)\,|\,b \in B\}$. When a has type A and b has type B, then $\mathrm{inj}_1(A{+}B, a)$ and $\mathrm{inj}_2(A{+}B, b)$ are of type $A{+}B$. In $\mathrm{inj}_1(A{+}B, a)$ the type $A{+}B$ must be explicitly stated, because from $\mathrm{inj}_1(a)$ alone it is impossible to decide whether its type is $A{+}B$ or for example $A{+}X$. The same holds for $\mathrm{inj}_2(A{+}B, b)$. When f_1 is a function of type $A{\to}C$ and f_2 is a function of type $B{\to}C$, then $f_1 {\triangledown} f_2$ is the function that has type $(A{+}B){\to}C$. Applying $f_1 {\triangledown} f_2$ on $\mathrm{inj}_1(A{+}B, a)$ reduces to $f_1\, a$ and applying it on $\mathrm{inj}_2(A{+}B, b)$ reduces to $f_2\, b$.

The abstract syntax for the set $\mathcal{T}$ of pseudo-terms is extended in the following way:

$$\mathcal{T} ::= \ldots \mid \mathcal{T} {\triangledown} \mathcal{T} \mid \mathcal{T}{+}\mathcal{T} \mid \mathrm{inj}_1(\mathcal{T}, \mathcal{T}) \mid \mathrm{inj}_2(\mathcal{T}, \mathcal{T})$$

The reduction rules are extended with the following rules:

$$(f_1 {\triangledown} f_2)\, \mathrm{inj}_1(A{+}B, c) > f_1\, c$$
$$(f_1 {\triangledown} f_2)\, \mathrm{inj}_2(A{+}B, c) > f_2\, c$$

The substitution rules are extended with the following rules:

$$\begin{array}{rcl}
(A{+}B)[x := N] & = & (A[x := N]){+}(B[x := N]) \\
\mathrm{inj}_1(A, B)[x := N] & = & \mathrm{inj}_1(A[x := N], B[x := N]) \\
\mathrm{inj}_2(A, B)[x := N] & = & \mathrm{inj}_2(A[x := N], B[x := N]) \\
(A {\triangledown} B)[x := N] & = & (A[x := N]) {\triangledown} (B[x := N])
\end{array}$$

The type derivation rules are extended with the following rules:

(+-form) $$\frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2}{\Gamma \vdash A{+}B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^+$$

(+-intro$_1$) $$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A{+}B : s}{\Gamma \vdash \mathrm{inj}_1(A{+}B, a) : A{+}B}$$

(+-intro$_2$) $$\frac{\Gamma \vdash b : B \quad \Gamma \vdash A{+}B : s}{\Gamma \vdash \mathrm{inj}_2(A{+}B, b) : A{+}B}$$

(+-elim) $$\frac{\Gamma \vdash f_1 : A_1{\to}C \quad \Gamma \vdash f_2 : A_2{\to}C \quad \Gamma \vdash A_1{+}A_2{\to}C : s}{\Gamma \vdash f_1 {\triangledown} f_2 : (A_1{+}A_2){\to}C}$$

(+-elim$'$) $$\frac{\Gamma \vdash f_1 : A_1{\to}C \quad \Gamma \vdash f_2 : A_2{\to}C \quad \Gamma \vdash e : A_1{+}A_2}{\Gamma \vdash f_1 {\triangledown} f_2\, e : C}$$

3.3.4 Σ-types

Dependent sum types, that have the form $(\Sigma x{:}A.\,B)$, can be seen as a generalization of the sum type. For example, $A_1{+}A_2$ is the same as "$\Sigma i \in \{1,2\}.\,A_i$". When a has type A and b has type $B[x := a]$, then $\mathrm{inj}((\Sigma x{:}A.\,B), a, b)$ has type $(\Sigma x{:}A.\,B)$. When F has type $(\Pi x{:}A.\,B{\to}C)$, then $\bigtriangledown F$ has type $(\Sigma x{:}A.\,B){\to}C$. Applying $\bigtriangledown F$ to $\mathrm{inj}((\Sigma x{:}A.\,B), a, b)$ reduces to $F\,a\,b$.

The abstract syntax for the set $\mathcal{T}$ of pseudo-terms is extended in the following way:

$$\mathcal{T} ::= \ldots \mid \bigtriangledown\mathcal{T} \mid \Sigma V{:}\mathcal{T}.\,\mathcal{T} \mid \mathrm{inj}(\mathcal{T}, \mathcal{T}, \mathcal{T})$$

The reduction rules are extended with the following rule:

$$\bigtriangledown F\,\mathrm{inj}((\Sigma x{:}A.\,B), a, b) > F\,a\,b$$

The substitution rules are extended with the following rules:

$$\begin{array}{rcl}
(\Sigma y{:}A.\,B)[x := N] & = & (\Sigma y{:}A[x := N].\,B[x := N]) \\
\mathrm{inj}(A, B, C)[x := N] & = & \mathrm{inj}(A[x := N], B[x := N], C[x := N]) \\
(\bigtriangledown A)[x := N] & = & \bigtriangledown(A[x := N])
\end{array}$$

Note that we still obey Barendregt's variable convention, which has the consequence that in $(\Sigma y{:}A.\,B)[x := N]$ the variable y does not occur in N.

The type derivation rules are extended with the following rules:

(Σ-form) $$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Sigma x{:}A.\,B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^{\Sigma}$$

(Σ-intro) $$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[x := a] \quad \Gamma \vdash (\Sigma x{:}A.\,B) : s}{\Gamma \vdash \mathrm{inj}((\Sigma x{:}A.\,B), a, b) : (\Sigma x{:}A.\,B)}$$

(Σ-elim) $$\frac{\Gamma \vdash F : (\Pi x{:}A.\,B{\to}C) \quad \Gamma \vdash (\Sigma x{:}A.\,B){\to}C : s}{\Gamma \vdash \bigtriangledown F : (\Sigma x{:}A.\,B){\to}C}$$

(Σ-elim′) $$\frac{\Gamma \vdash M : (\Sigma x{:}A.\,B) \quad \Gamma \vdash F : (\Pi x{:}A.\,B{\to}C) \quad \Gamma \vdash C : s}{\Gamma \vdash \bigtriangledown F\,M : C}$$

Chapter 4

Propositions as types

Proof assistants can be based on the theory of pure type systems and the propositions as types principle [13]. In the previous chapter we described the PTS framework, here we describe the propositions as types principle.

The introduction and elimination rules of the type derivation rules, which were introduced in the previous chapter, look similar to the derivation rules in predicate logic. In Table 4.1 the introduction and elimination rules of implications in logic and arrow types in typed lambda calculi are shown. When in the $\rightarrow$-rules the terms, the :-symbols and the premise containing an s are removed, and the $\rightarrow$-symbol is replaced by a $\Rightarrow$-symbol, then the rules are identical. There is a similar correspondence between the $\times, +, \Pi, \Sigma$-type constructors and the $\wedge, \vee, \forall, \exists$-connectives. This correspondence between logic and typed lambda calculus is known as the propositions as types principle.

	$\Rightarrow$	$\rightarrow$
intro	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash A \rightarrow B : s}{\Gamma \vdash (\lambda x{:}A.\, b) : A \rightarrow B}$
elim	$\dfrac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash F : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B}$

Table 4.1: Introduction and elimination rules

In the previous chapter the judgment $\Gamma \vdash M : \tau$ had the interpretation: "In context Γ, term M has type τ". In this chapter it has another interpretation: "In context Γ, term M encodes the proof of proposition τ".

4.1 Term finding example

Suppose we want to find a term that has type $P \rightarrow (Q \rightarrow P)$ in the simply typed lambda calculus [6]. We can find such a term by applying the type derivation rules backwards. $P \rightarrow (Q \rightarrow P)$ has the form $A \rightarrow B$, so we use the rule $\rightarrow$-intro. According to the rule we need to extend the context with a variable that has type P and then we need to find a term that has type $Q \rightarrow P$:

1	$\boxed{x{:}P}$	
	$\vdots$	
m	$?{:}Q{\to}P$	
n	$?{:}P{\to}(Q{\to}P)$	$\to$-intro on (1) and (m)

Again, the type of the term that we want to find is of the form $A{\to}B$. So we take a similar step:

1	$\boxed{x{:}P}$	
2	$\boxed{y{:}Q}$	
	$\vdots$	
l	$?{:}P$	
m	$?{:}Q{\to}P$	$\to$-intro on (2) and (l)
n	$?{:}P{\to}(Q{\to}P)$	$\to$-intro on (1) and (m)

Now we need to find a term that has type P. Fortunately, we have such a term in the context:

1	$\boxed{x{:}P}$	
2	$\boxed{y{:}Q}$	
3	$x{:}P$	Start on (1)
4	$?{:}Q{\to}P$	$\to$-intro on (2) and (3)
5	$?{:}P{\to}(Q{\to}P)$	$\to$-intro on (1) and (4)

The rest of this example follows from the rule $\to$-intro, twice:

1	$\boxed{x{:}P}$	
2	$\boxed{y{:}Q}$	
3	$x{:}P$	Start on (1)
4	$(\lambda y{:}Q.x){:}Q{\to}P$	$\to$-intro on (2) and (3)
5	$(\lambda x{:}P.(\lambda y{:}Q.x)){:}P{\to}(Q{\to}P)$	$\to$-intro on (1) and (4)

When in this last figure we remove the terms and the :-symbol, we change every $\to$-symbol to a $\Rightarrow$-symbol and we change the commentary accordingly, we get:

1	$\boxed{P}$	
2	$\quad\boxed{Q}$	
3	$\quad P$	assumption on (1)
4	$Q \Rightarrow P$	$\Rightarrow$-intro on (2) and (3)
5	$P \Rightarrow (Q \Rightarrow P)$	$\Rightarrow$-intro on (1) and (4)

This derivation is a proof of $P \Rightarrow (Q \Rightarrow P)$ being a tautology. What we basically did is, we encoded the proposition $P \Rightarrow (Q \Rightarrow P)$ as the type $P \to (Q \to P)$ and we tried to find a term that has this type. Hence the name *propositions as types*.

4.2 Type derivation example

Suppose we want to derive the type of the term $(\lambda x{:}P.\,(\lambda f{:}(P \to Q).\,f\ x))$ in the simply typed lambda calculus. The term is a λ-term, so we use the rule $\to$-intro. According to the rule, we need to extend the context with a variable x of type P and derive the type of the body of the λ-term:

1	$\boxed{x{:}P}$	
	$\vdots$	
m	$(\lambda f{:}(P \to Q).\,f\ x){:}?$	
n	$(\lambda x{:}P.\,(\lambda f{:}(P \to Q).\,f\ x)){:}?$	$\to$-intro on (1) and (m)

Now we have again a λ-term, so we repeat the procedure:

1	$\boxed{x{:}P}$	
2	$\quad\boxed{f{:}P \to Q}$	
	$\quad\vdots$	
l	$\quad f\ x{:}?$	
m	$(\lambda f{:}(P \to Q).\,f\ x){:}?$	$\to$-intro on (2) and (l)
n	$(\lambda x{:}P.\,(\lambda f{:}(P \to Q).\,f\ x)){:}?$	$\to$-intro on (1) and (m)

$f\ x$ is an application, so we use the $\to$-elim rule, which means that we need to find the type of f and of x:

1	$\boxed{x{:}P}$	
2	$\quad\boxed{f{:}P{\to}Q}$	
	$\quad\vdots$	
j	$\quad f{:}?$	
	$\quad\vdots$	
k	$\quad x{:}?$	
l	$\quad f\,x{:}?$	$\to$-elim on (j) and (k)
m	$(\lambda f{:}(P{\to}Q).\,f\,x){:}?$	$\to$-intro on (2) and (l)
n	$(\lambda x{:}P.\,(\lambda f{:}(P{\to}Q).\,f\,x)){:}?$	$\to$-intro on (1) and (m)

f and x are both in the context, so we know their types:

1	$\boxed{x{:}P}$	
2	$\quad\boxed{f{:}P{\to}Q}$	
3	$\quad f{:}P{\to}Q$	Start on (2)
4	$\quad x{:}P$	Start on (1)
5	$\quad f\,x{:}?$	$\to$-elim on (3) and (4)
6	$(\lambda f{:}(P{\to}Q).\,f\,x){:}?$	$\to$-intro on (2) and (5)
7	$(\lambda x{:}P.\,(\lambda f{:}(P{\to}Q).\,f\,x)){:}?$	$\to$-intro on (1) and (6)

f has type $P{\to}Q$ and x has type P. The type of x matches with the left hand side of the type of f, so $f\,x$ has type Q:

1	$\boxed{x{:}P}$	
2	$\quad\boxed{f{:}P{\to}Q}$	
3	$\quad f{:}P{\to}Q$	Start on (2)
4	$\quad x{:}P$	Start on (1)
5	$\quad f\,x{:}Q$	$\to$-elim on (3) and (4)
6	$(\lambda f{:}(P{\to}Q).\,f\,x){:}?$	$\to$-intro on (2) and (5)
7	$(\lambda x{:}P.\,(\lambda f{:}(P{\to}Q).\,f\,x)){:}?$	$\to$-intro on (1) and (6)

The rest of this example follows from the $\to$-intro, twice:

1	$\boxed{x{:}P}$	
2	$\quad\boxed{f{:}P{\to}Q}$	
3	$\quad f{:}P{\to}Q$	Start on (2)
4	$\quad x{:}P$	Start on (1)
5	$\quad f\,x{:}Q$	$\to$-elim on (3) and (4)
6	$(\lambda f{:}(P{\to}Q).\,f\,x){:}(P{\to}Q){\to}Q$	$\to$-intro on (2) and (5)
7	$(\lambda x{:}P.\,(\lambda f{:}(P{\to}Q).\,f\,x)){:}P{\to}((P{\to}Q){\to}Q)$	$\to$-intro on (1) and (6)

We have derived that the type of the term $(\lambda x{:}P.\,(\lambda f{:}(P{\to}Q).\,f\,x))$ is $P{\to}((P{\to}Q){\to}Q)$. The term determines the whole structure of this type derivation. Due to the propositions as types principle we have now reconstructed a proof for the tautology $P{\Rightarrow}((P{\Rightarrow}Q){\Rightarrow}Q)$. This is more clear when all terms and :-symbols are removed and all $\to$-symbols are replaced by $\Rightarrow$-symbols. The term is an encoding of the proof. This is known as *proofs as terms*.

Part II

Abstract code model

Part II contains the abstract code of a proof assistant. Chapter 5 describes how we represent terms. A type checker to compute and check the type of a term is given in Chapter 6. Chapter 7 contains tactics to construct a term, given a type. In Part III an implementation of this abstract code model is described.

Chapter 5

Terms, items and contexts

To support convenient manipulation of terms, they are represented as trees, where every node in the tree is annotated with the type of the subterm that the node represents. Incomplete proofs have holes, so we allow that terms have holes too, each annotated with its desired type. An item is the defining occurrence of a variable name and its type. A context is a list of items. Every node n in the tree has a local context, which corresponds to the list of items that is gathered while walking the shortest path that starts in the root node and ends in node n. The global context is just a list of items, separate from the tree.

For example, in Figure 5.1 the tree that represents the term $(\lambda x{:}P.(\lambda y{:}Q.x))$ is shown. The global context consists of the list $P{:}*, Q{:}*$. Every occurrence of P and Q in the figure refers to the global context, but to keep the figure legible the global context and the bindings to the global variables are not shown. The root node is a λ-node and the local context in this node is empty. The local context of the other λ-node consists only of the item $x{:}P$. The local context of the node that is shown as an x without a box around it, is the list $x{:}P, y{:}Q$. This means that the variable x is in scope and the binding is shown as an arrow pointing to the defining occurrence of the variable.

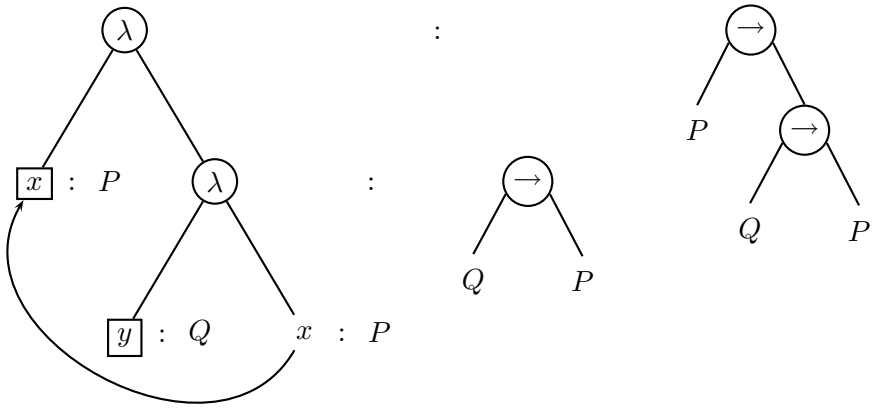


Figure 5.1: Tree representation of $(\lambda x{:}P.(\lambda y{:}Q.x))$

To create such trees we need a set of sorts and a set of variable names, which we will call respectively $Sort$ and $Name$. We will also need some subroutines:

A function that creates an item is needed.

- $createItem(AName : Name,\ AType : Node) : Node$

For every kind of term that is introduced in Chapter 3 we need a function that creates the corresponding node. To represent holes in proofs, we need a function for creating hole-nodes. For every argument in the following functions, we assume there is a selection operator to retrieve its value. For example, when a pair-node has been created with $pair := createPair(A, B)$, the values A and B are retrieved by $pair.FLeft$ and $pair.FRight$. This also holds for items.

- $createSort(ASortName : Sort) : Node$
- $createVariable(AItem : Node) : Node$
- $createLambda(AItem : Node,\ ABody : Node) : Node$
- $createArrow(APremise : Node,\ AConclusion : Node) : Node$
- $createPi(AItem : Node,\ ABody : Node) : Node$
- $createApplication(AFunction : Node,\ AArgument : Node) : Node$
- $createPair(ALeft : Node,\ ARight : Node) : Node$
- $createProduct(ALeft : Node,\ ARight : Node) : Node$
- $createProjectionLeft(APair : Node) : Node$
- $createProjectionRight(APair : Node) : Node$
- $createSplit(ALeft : Node,\ ARight : Node) : Node$
- $createSum(ALeft : Node,\ ARight : Node) : Node$
- $createInjectionLeft(AType : Node,\ ATerm : Node) : Node$
- $createInjectionRight(AType : Node,\ ATerm : Node) : Node$
- $createSplitGeneralization(AFunction : Node) : Node$
- $createSigma(AItem : Node,\ ABody : Node) : Node$
- $createInjectionGeneralization(AType : Node,\ ATerm1 : Node,\ ATerm2 : Node) : Node$
- $createHole() : Node$

We also need a procedure that annotates a node with a type and a function to return the type of a term.

- $setType(ATerm : Node,\ AType : Node)$
- $getType(ATerm : Node) : Node$

For the global context, we need a function that creates an empty context and a procedure to extend a context with an item. To maintain the well-formedness of the context, an item should only be added to the context when its type is typable in the current global context.

- $createEmptyContext() : Context$
- $extendContext(AContext : Context,\ AItem : Node)$

To calculate with terms, subroutines for copying, substitution, reduction to normal form, testing for equality modulo conversion and unification are needed. We require that reducing to normal form is possible in all PTSs that are used in the proof assistant.

- $copy(ATerm : Node) : Node$
- $substitute(ATerm1 : Node,\ AVariableName : Name,\ ATerm2 : Node) : Node$
- $reduce(ATerm : Node) : Node$
- $isConvertible(ATerm1 : Node,\ ATerm2 : Node) : Boolean$
- $unify(ATerm1 : Node,\ ATerm2 : Node) : SubstitutionSet$

With these subroutines the tree in Figure 5.1 can be created in the following way (*rootNode* represents the root of the tree):

$P := createItem(\text{"}P\text{"},\ createSort(\text{"}*\text{"}));$
$Q := createItem(\text{"}Q\text{"},\ createSort(\text{"}*\text{"}));$
$globalContext := createEmptyContext();$
$extendContext(globalContext,\ P);$
$extendContext(globalContext,\ Q);$

$x := createItem(\text{"}x\text{"},\ createVariable(P));$
$y := createItem(\text{"}y\text{"},\ createVariable(Q));$

$xNode := createVariable(x);$
$setType(xNode,\ createVariable(P));$

$lyNode := createLambda(y,\ xNode);$
$setType(lyNode,\ createArrow(createVariable(Q),\ copy(getType(xNode))));$

$rootNode := createLambda(x,\ lyNode);$
$rootType := createArrow(createVariable(P),\ copy(getType(lyNode)));$
$setType(rootNode,\ rootType);$

Chapter 6

Type checker

Now that we have a way to represent terms, items and contexts, we want to compute the type of a specific term in a specific context. Or when a term is annotated with a type, we want to check whether this annotation is correct. For this we use the type derivation rules of the PTS framework. These rules depend on the values of $\mathcal{S}, \mathcal{A}, \mathcal{R}^{\Pi}, \mathcal{R}^{\rightarrow}, \mathcal{R}^{\times}, \mathcal{R}^{+}$ and $\mathcal{R}^{\Sigma}$ that specify a PTS with additional type constructors. We assume we have representations of these sets and we can apply the usual set operations on them, like checking whether an object is an element of a set.

We want an efficient implementation of a type checker and this can be done by using a syntax directed subroutine. Unfortunately, the type derivation rules in Section 3.3 are not syntax directed. This problem is caused by the conversion rule: By looking at the structure of a specific term it cannot be decided when to apply this rule. A solution is to distribute the conversion rule over the other rules. This makes a separate conversion rule obsolete and thus it can be removed. This is done in the following altered type derivation rules.

6.1 Altered type derivation rules

Some of the altered type derivation rules have a side condition that states that a context should be valid. Checking whether a context is valid can be done with the following derivation rules. $\Gamma \vdash ok$ means that context Γ is valid.

$$(ok\text{-axiom}) \qquad <> \vdash ok$$

$$(ok\text{-ext}) \qquad \frac{\Gamma \vdash ok \quad \Gamma \vdash A : s}{\Gamma, x{:}A \vdash ok}$$

With the new axiom-rule, sorts can be typed in every valid context. The start-rule and the weaken-rule have been combined into one rule.

$$(\text{axiom}) \qquad \Gamma \vdash s_1 : s_2 \qquad \text{if } (s_1{:}s_2) \in \mathcal{A} \text{ and } \Gamma \text{ is a valid context}$$

$$(\text{start\&weaken}) \qquad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A, \Delta \vdash x : A} \qquad \text{if } \Gamma, x{:}A, \Delta \text{ is a valid context}$$

The Π-form-rule and the Π-elim-rule have been altered, while the Π-intro-rule is left unchanged.

$$(\Pi\text{-form}) \qquad \frac{\Gamma \vdash A : \sigma_1 \quad \sigma_1 \simeq s_1 \qquad \Gamma, x{:}A \vdash B : \sigma_2 \quad \sigma_2 \simeq s_2}{\Gamma \vdash (\Pi x{:}A.\, B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^{\Pi}$$

$$(\Pi\text{-intro}) \qquad \frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash (\lambda x{:}A.\, b) : (\Pi x{:}A.\, B)}$$

$$(\Pi\text{-elim}) \qquad \frac{\Gamma \vdash F : \tau \quad \tau \simeq (\Pi x{:}A.\, B) \qquad \Gamma \vdash a : A \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash F\, a : B[x := a]}$$

The →-form-rule and the →-elim-rule have been altered, while the →-intro-rule is left unchanged.

$$(\to\text{-form}) \qquad \frac{\Gamma \vdash A : \sigma_1 \quad \sigma_1 \simeq s_1 \qquad \Gamma \vdash B : \sigma_2 \quad \sigma_2 \simeq s_2}{\Gamma \vdash A \to B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^{\to}$$

$$(\to\text{-intro}) \qquad \frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash A \to B : s}{\Gamma \vdash (\lambda x{:}A.\, b) : A \to B}$$

$$(\to\text{-elim}) \qquad \frac{\Gamma \vdash F : \tau \quad \tau \simeq A \to B \qquad \Gamma \vdash a : A \quad \Gamma \vdash A \to B : s}{\Gamma \vdash F\, a : B}$$

The rules ×-form, ×-elim$_1$ and ×-elim$_2$ have been altered, while the ×-intro-rule is left unchanged.

$$(\times\text{-form}) \qquad \frac{\Gamma \vdash A : \sigma_1 \quad \sigma_1 \simeq s_1 \qquad \Gamma \vdash B : \sigma_2 \quad \sigma_2 \simeq s_2}{\Gamma \vdash A \times B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^{\times}$$

$$(\times\text{-intro}) \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma \vdash A \times B : s}{\Gamma \vdash \langle a, b \rangle : A \times B}$$

$$(\times\text{-elim}_1) \qquad \frac{\Gamma \vdash M : \tau \quad \tau \simeq A \times B \quad \Gamma \vdash A \times B : s}{\Gamma \vdash \pi_1(M) : A}$$

$$(\times\text{-elim}_2) \qquad \frac{\Gamma \vdash M : \tau \quad \tau \simeq A \times B \quad \Gamma \vdash A \times B : s}{\Gamma \vdash \pi_2(M) : B}$$

All +-rules are changed. The +-elim′-rule is even removed. It was used for computing the type of terms that have the structure $f_1 \triangledown f_2\, e$. But those terms can also be typed with the +- and →-elim-rules. So, the +-elim′-rule is obsolete and can be removed. This makes the type checker smaller, hence there is less room for errors.

$$\text{(+-form)} \qquad \frac{\Gamma \vdash A : \sigma_1 \quad \sigma_1 \simeq s_1 \qquad \Gamma \vdash B : \sigma_2 \quad \sigma_2 \simeq s_2}{\Gamma \vdash A{+}B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^+$$

$$\text{(+-intro}_1\text{)} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash \tau : s \quad \tau \simeq A{+}B \quad \Gamma \vdash A{+}B : s}{\Gamma \vdash \mathrm{inj}_1(\tau, a) : \tau}$$

$$\text{(+-intro}_2\text{)} \qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash \tau : s \quad \tau \simeq A{+}B \quad \Gamma \vdash A{+}B : s}{\Gamma \vdash \mathrm{inj}_2(\tau, b) : \tau}$$

$$\text{(+-elim)} \qquad \frac{\Gamma \vdash f_1 : \tau_1 \quad \tau_1 \simeq A_1{\to}C \qquad \Gamma \vdash f_2 : \tau_2 \quad \tau_2 \simeq A_2{\to}C \qquad \Gamma \vdash A_1{+}A_2{\to}C : s}{\Gamma \vdash f_1 \triangledown f_2 : A_1{+}A_2{\to}C}$$

All Σ-rules are changed. The Σ-elim′-rule is even removed. It was used for computing the type of terms that have the structure $\triangledown F\, M$. But those terms can also be typed with the Σ- and →-elim-rules. So, the Σ-elim′-rule is obsolete and can be removed. This makes the type checker smaller, hence there is less room for errors.

$$\text{(}\Sigma\text{-form)} \qquad \frac{\Gamma \vdash A : \sigma_1 \quad \sigma_1 \simeq s_1 \qquad \Gamma, x{:}A \vdash B : \sigma_2 \quad \sigma_2 \simeq s_2}{\Gamma \vdash (\Sigma x{:}A.\, B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}^\Sigma$$

$$\text{(}\Sigma\text{-intro)} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[x := a] \qquad \Gamma \vdash \tau : s \quad \tau \simeq (\Sigma x{:}A.\, B) \quad \Gamma \vdash (\Sigma x{:}A.\, B) : s}{\Gamma \vdash \mathrm{inj}(\tau, a, b) : \tau}$$

$$\text{(}\Sigma\text{-elim)} \qquad \frac{\Gamma \vdash F : \tau \quad \tau \simeq (\Pi x{:}A.\, B{\to}C) \quad \Gamma \vdash (\Pi x{:}A.\, B{\to}C) : s \qquad \Gamma \vdash C : s' \quad \Gamma \vdash (\Sigma x{:}A.\, B) : s''}{\Gamma \vdash \triangledown F : (\Sigma x{:}A.\, B){\to}C}$$

The big question with these altered type derivation rules is whether these rules are equivalent with the original rules. In other words, can the same terms be typed with these rules as with the original rules? Fortunately, these rules are sound, which means that everything that can be typed with these altered rules can also be typed with the original rules. Whether these rules are complete, which means that everything that can be typed with the original rules can also be typed with the altered rules, is difficult to decide and further investigation is needed.

These rules cannot be used in a syntax directed type checker for all PTSs. The problem becomes apparent in the following example. Let $\mathcal{S} = \{*, \Box\}$, $\mathcal{A} = \{(*, \Box)\}$, $\mathcal{R}^{\rightarrow} = \{(*,*,*), (*,*,\Box)\}$ and A, B and C are of type $*$. When a syntax directed type checker has to compute the type of $A{\rightarrow}(B{\rightarrow}C)$, it first has to compute the type of subterm $B{\rightarrow}C$. B and C are both of type $*$, so according to the $\rightarrow$-$form$ rule it needs to pick an s_3 such that $(*,*,s_3) \in \mathcal{R}^{\rightarrow}$. The problem is that there are two candidates for s_3: $*$ and $\Box$. When the type checker chooses $s_3 = *$, then $A{\rightarrow}(B{\rightarrow}C)$ is of type $*$. However, when it picks $s_3 = \Box$, then $A{\rightarrow}(B{\rightarrow}C)$ is untypable. This problem can be fixed by restricting $\mathcal{A}$ and all $\mathcal{R}^{TC}$, where $TC \in \{\Pi, \rightarrow, \times, +, \Sigma\}$, such that

$$(\forall s, s', s'' {:} \mathcal{S}. ((s{:}s') \in \mathcal{A} \wedge (s{:}s'') \in \mathcal{A}) \Rightarrow (s' = s''))$$

and

$$(\forall s_1, s_2, s_3, s_3' {:} \mathcal{S}. ((s_1, s_2, s_3) \in \mathcal{R}^{TC} \wedge (s_1, s_2, s_3') \in \mathcal{R}^{TC}) \Rightarrow (s_3 = s_3')).$$

A PTS that complies to these restrictions is called a functional PTS.

6.2 Subroutines for type computing

In an implementation the validity of the context can be kept invariant: The empty context is valid and extending a valid context with an item keeps the context valid when the type of the item is typable in the unextended context. Because the validity of a context is kept invariant, computing whether the context is valid is not necessary.

With the aid of the altered type derivation rules a syntax directed subroutine can be constructed that computes the type of a term and annotates the term with it. When a term cannot be typed, this function returns the error value *True* and it returns *False* when no error occurred. This function is called computeType:

```
computeType(AContext : Context, ATerm : Node) : Boolean
|[
  if ATerm :: Sort → r := computeTypeSort(AContext, ATerm)
  [] ATerm :: Variable → r := computeTypeVariable(AContext, ATerm)
  [] ATerm :: Lambda → r := computeTypeLambda(AContext, ATerm)
  [] ATerm :: Arrow → r := computeTypeArrow(AContext, ATerm)
  [] ATerm :: Pi → r := computeTypePi(AContext, ATerm)
  [] ATerm :: Application → r := computeTypeApplication(AContext, ATerm)
  [] ATerm :: Pair → r := computeTypePair(AContext, ATerm)
  [] ATerm :: Product → r := computeTypeProduct(AContext, ATerm)
  [] ATerm :: ProjectionLeft → r := computeTypeProjectionLeft(AContext, ATerm)
  [] ATerm :: ProjectionRight → r := computeTypeProjectionRight(AContext, ATerm)
  [] ATerm :: Split → r := computeTypeSplit(AContext, ATerm)
  [] ATerm :: Sum → r := computeTypeSum(AContext, ATerm)
  [] ATerm :: InjectionLeft → r := computeTypeInjectionLeft(AContext, ATerm)
  [] ATerm :: InjectionRight → r := computeTypeInjectionRight(AContext, ATerm)
  [] ATerm :: SplitGeneralization → r := computeTypeSplitGen(AContext, ATerm)
```

```
  [] ATerm :: Sigma → r := computeTypeSigma(AContext, ATerm)
  [] ATerm :: InjectionGeneralization → r := computeTypeInjectionGen(AContext, ATerm)
  [] ATerm :: Hole → r := computeTypeHole(AContext, ATerm)
  fi ;
  return r
]|
```

As the reader may notice, a function is needed for every term that can be created using the term creators that were introduced in the previous chapter. This includes the hole-term. All these functions are rather straightforward, so we do not show them all. But as an example we show how the type of a pair term like $\langle a, b\rangle$ is computed, to get an impression of what these functions look like.

```
computeTypePair(AContext : Context, APairTerm : Node) : Boolean
|[
  error1 := computeType(AContext, APairTerm.FLeft);
  error2 := computeType(AContext, APairTerm.FRight);
  error := error1 ∨ error2;
  if ¬error →
     A := getType(APairTerm.FLeft);
     B := getType(APairTerm.FRight);
     pairType := createProduct(copy(A), copy(B));
     error := computeType(AContext, pairType);
     if ¬error → setType(APairTerm, pairType)
     [] error → skip
     fi
  [] error → skip
  fi ;
  return error
]|
```

The term $F\,a$ appears in the Π-elim-rule and in the →-elim-rule. So a syntax directed routine that computes the type of $F\,a$ needs to combine these two rules. It first needs to compute the type of subterm F and depending on the structure of that type, the type of the whole term can be constructed. The term $(\lambda x{:}A.\,b)$ also appears in two type derivation rules, hence it has a similar problem and a similar solution.

Hole-terms do not appear in the type derivation rules, so we need to describe here what a subroutine for computing the type of a hole should do: It checks whether a hole is annotated with a type. When it is not annotated, the error value *True* is returned and *False* otherwise.

```
computeTypeHole(AContext : Context, AHoleTerm : Node) : Boolean
|[
  if getType(AHoleTerm) = nil → error := True
  [] getType(AHoleTerm) ≠ nil → error := False
  fi ;
  return error
]|
```

Chapter 7

Tactics

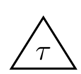
Up to now, we have described how we represent terms and contexts, and how we compute the type of a term in a context. In this chapter we want to construct, given a type τ and a context Γ, a term that has type τ in context Γ. For this we use so-called tactics. A tactic is a way to fill a hole $?_t$ with a term, which might contain new holes. Presumably, these new holes are easier to fill than hole $?_t$. The basic tactics correspond to applying the type derivation rules from Chapter 3, in particular the introduction and elimination rules, backwards. These basic tactics can later be combined to form extended tactics, which fill holes faster because of less user involvement.

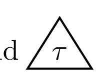
A proof assistant maintains a couple of objects concerning a theorem and its proof:

- A global context.
- A proof tree that is being constructed. Initially, this is a hole that is annotated with a given type.
- A list of all the holes in the proof tree. This list is empty when the proof is complete.
- A focused node, which is a node in the list of holes. It represents the current goal in the proof derivation.
- The local context of the focused node.

When a tactic is applied on the focused node and it finishes successfully, the system replaces the hole with the result of the tactic. When the list of holes is not empty, the system focuses another hole.

In this chapter, for every tactic a graphical representation of what happens to the proof tree when the tactic is applied, is shown. When the focused node is of the general form

$?_t$: 

where $?_t$ is a hole that needs to be filled with a term of type τ, and 

is a tree representation of type τ, then there are multiple ways to fill this hole with a tree representation of a term.

The first section in this chapter describes the subroutines that are used by the tactics. The second section is about general tactics that can be applied to every hole. Each section after

that is based on a type constructor and contains tactics that are based on the introduction and elimination rules of that type constructor. The description of every tactic contains a list of steps that a tactic needs to take. To keep this list readable, we assume that every step finishes without problems. When an error does occur, the tactic returns immediately and the proof tree remains the same as before applying the tactic. Many of the tactics seem similar, of special interest are the following: Π-elimination, →-introduction, ×-elimination, +-elimination, Σ-introduction and Σ-elimination.

7.1 Subroutines

The tactics use the subroutines that are introduced in the previous two chapters and two new subroutines. The first subroutine is a function that generates a name that is not yet used.

- *createFreshName*() : *Name*

The second subroutine is a function that searches the local and global context for an item that has the same type as the function's argument. When such an item is found, a variable that refers to this item is return, otherwise an error occurs.

- *searchContext*(*AType* : *Node*) : *Node*

7.2 General tactics

The tactics in this section can be applied on every hole.

7.2.1 Exact

This tactic corresponds to the conv-rule. The user supplies the exact term, which has a type that is convertible to the goal type τ. The system fills the hole with this term when it has checked that it has a type that is convertible to the goal type.

Focused hole

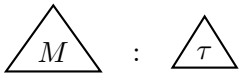
$?_t$: 

Steps the tactic needs to take

exact(M)
Pre : M:τ

1. *setType*(M, τ)
2. return M

Filled hole after applying tactic



7.2.2 Reduce goal

This tactic also corresponds to the conv-rule. It reduces the type of the goal to normal form.

Focused hole

$?_t \ : \ \tau$

Steps the tactic needs to take

reduceGoal()

1. *result* := *createHole*()
2. *setType*(*result*, *reduce*(*getType*(Focused hole)))
3. return *result*

Filled hole after applying tactic

$?_{t'} \ : \ \tau'$

7.2.3 Assumption

This tactic corresponds to the start-rule. It searches the context for a variable that has the same type as the goal.

Focused hole

$?_t \ : \ \tau$

Steps the tactic needs to take

assumption()

1. *result* := *searchContext*(τ)
2. *setType*(*result*, *copy*(*getType*(Focused hole)))
3. return *result*

Filled hole after applying tactic

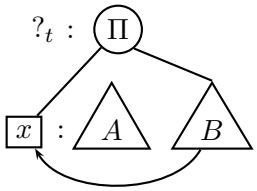
$x \ : \ \tau$

7.3 Π-types

7.3.1 Π-introduction

This tactic corresponds to the Π-intro-rule. To construct a term of type $(\Pi x{:}A.\,B)$, we extend the context with a variable x of type A and construct a term of type B where x may occur in B.

Focused hole

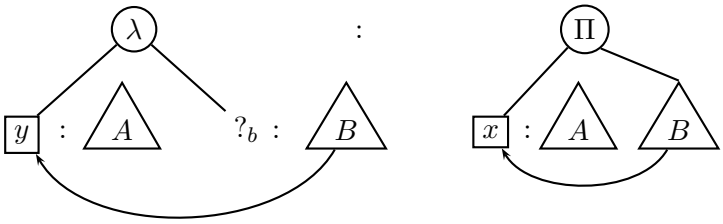


Steps the tactic needs to take

PiIntro()

1. $y := \mathit{createItem}(\mathit{createFreshName}(), \mathit{copy}(A))$
2. $?_b := \mathit{createHole}()$
3. $\mathit{setType}(?_b, \mathit{substitute}(B, x, \mathit{createVariable}(y)))$
4. $\mathit{result} := \mathit{createLambda}(y, ?_b)$
5. $\mathit{setType}(\mathit{result}, \mathit{copy}(\mathit{getType}(\text{Focused hole})))$
6. return *result*

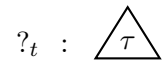
Filled hole after applying tactic



7.3.2 Π-elimination

This tactic corresponds to the Π-elim-rule. When the goal is to construct a term of type τ and the context contains a variable F of type $(\Pi x{:}A.\,B)$, we can try to unify B and τ to get a variable a. When this succeeds, applying F on a leads to a term of type τ.

Focused hole

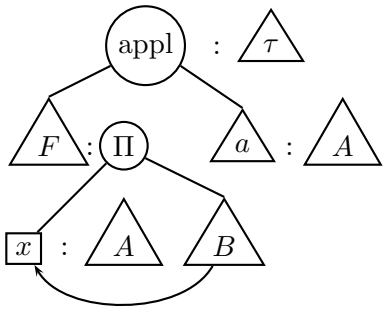


Steps the tactic needs to take

PiElim(F)
Pre : (F:t) $\wedge$ ($t \simeq (\Pi x{:}A.\, B)$)

1. *unify*(B, τ) :: $\{x := a\}$
2. *setType*(a, *copy*(A))
3. *result* := *createApplication*(F, a)
4. *setType*(*result*, *copy*(*getType*(Focused hole)))
5. return *result*

Filled hole after applying tactic

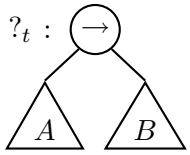


7.4 →-types

7.4.1 →-introduction

This tactic corresponds to the →-intro-rule. To construct a term of type $A \to B$, we extend the context with a variable of type A and construct a term of type B.

Focused hole

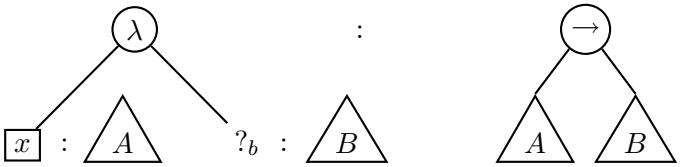


Steps the tactic needs to take

arrowIntro()

1. x := *createItem*(*createFreshName*(), *copy*(A))
2. $?_b$:= *createHole*()
3. *setType*($?_b$, *copy*(B))
4. *result* := *createLambda*($x, ?_b$)
5. *setType*(*result*, *copy*(*getType*(Focused hole)))
6. return *result*

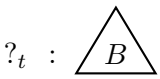
Filled hole after applying tactic



7.4.2 →-elimination

This tactic corresponds to the →-elim-rule. To construct a term of type B one can construct a lambda function f of type $A \rightarrow B$ and a term a of type A and when f is applied to a, a term of type B is constructed.

Focused hole

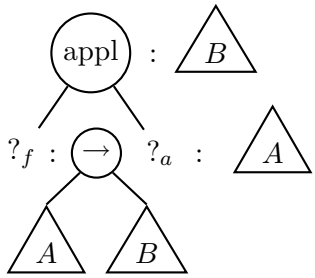


Steps the tactic needs to take

arrowElim(A)

1. $?_f$:= *createHole*()
2. *setType*($?_f$, *createArrow*(*copy*(A), *copy*(B)))
3. $?_a$:= *createHole*()
4. *setType*($?_a$, *copy*(A))
5. *result* := *createApplication*($?_f$, $?_a$)
6. *setType*(*result*, *copy*(*getType*(Focused hole)))
7. return *result*

Filled hole after applying tactic

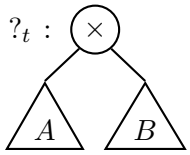


7.5 ×-types

7.5.1 ×-introduction

This tactic corresponds to the ×-intro-rule. To construct a term of type $A \times B$, one constructs a term of type A and a term of type B and then combines them using the pair-operator.

Focused hole

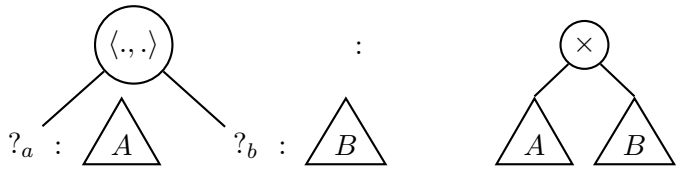


Steps the tactic needs to take

productIntro()

1. $?_a, ?_b :=$ *createHole*(), *createHole*()
2. *setType*($?_a$, *copy*(A))
3. *setType*($?_b$, *copy*(B))
4. *result* := *createPair*($?_a$, $?_b$)
5. *setType*(*result*, *copy*(*getType*(Focused hole)))
6. return *result*

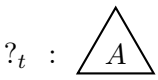
Filled hole after applying tactic



7.5.2 ×-elimination

This tactic corresponds to the ×-elim$_1$-rule. The tactic that corresponds to the ×-elim$_2$-rule is very similar. To construct a term of type A one can construct a term of type $A \times B$ and then apply the left projection function to get a term of type A.

Focused hole

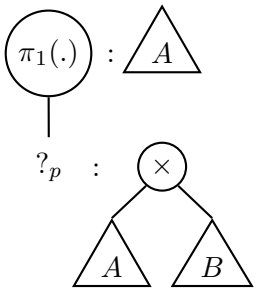


Steps the tactic needs to take

productElim1(B)

1. $?_p :=$ *createHole*()
2. *setType*($?_p$, *createProduct*(*copy*(A), *copy*(B)))
3. *result* := *createProjectionLeft*($?_p$)
4. *setType*(*result*, *copy*(*getType*(Focused hole)))
5. return *result*

Filled hole after applying tactic

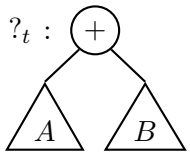


7.6 +-types

7.6.1 +-introduction

This tactics correspond to the +-intro$_1$-rule. The tactic that corresponds to the +-intro$_2$-rule is very similar. To construct a term of type A+B, one can construct a term of type A and then use the injection$_1$-operator to create a term of type A+B. But from the term $\text{inj}_1(a)$ alone one cannot tell whether its type is A+B or for example A+X, so it is essential to append the type to the term.

Focused hole

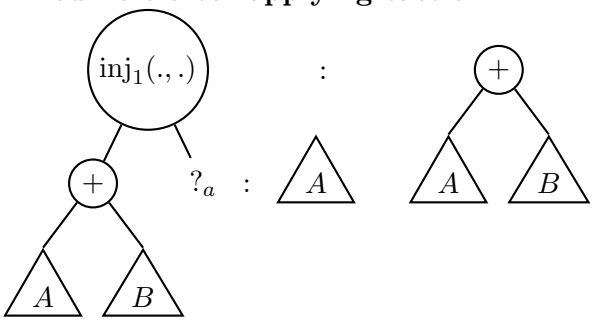


Steps the tactic needs to take

sumIntro1 ()

1. $?_a$:= *createHole*()
2. *setType*($?_a$, *copy*(A))
3. *result* := *createInjectionLeft*(*copy*(*getType*(Focused hole)), $?_a$)
4. *setType*(*result*, *copy*(*getType*(Focused hole)))
5. return *result*

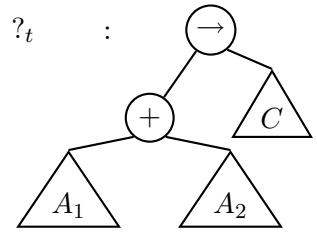
Filled hole after applying tactic



7.6.2 +-elimination

This tactic corresponds to the +-elim-rule. To construct a term of type $(A_1{+}A_2){\rightarrow}C$, we can construct a term of type $A_1{\rightarrow}C$ and another one of type $A_2{\rightarrow}C$, and then combine these two using the split-operator to get a term of the desired type.

Focused hole

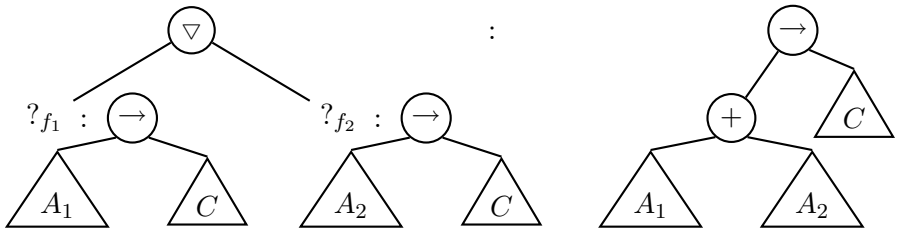


Steps the tactic needs to take

sumElim()

1. $?_{f_1}, ?_{f_2} := \mathit{createHole}(), \mathit{createHole}()$
2. $\mathit{setType}(?_{f_1}, \mathit{createArrow}(\mathit{copy}(A_1), \mathit{copy}(C)))$
3. $\mathit{setType}(?_{f_2}, \mathit{createArrow}(\mathit{copy}(A_2), \mathit{copy}(C)))$
4. $\mathit{result} := \mathit{createSplit}(?_{f_1}, ?_{f_2})$
5. $\mathit{setType}(\mathit{result}, \mathit{copy}(\mathit{getType}(\text{Focused hole})))$
6. return *result*

Filled hole after applying tactic

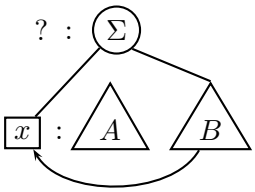


7.7 Σ-type

7.7.1 Σ-introduction

This tactic corresponds to the Σ-intro-rule. To construct a term of type $(\Sigma x{:}A.\,B)$ the following can be done. First a term a of the type A is constructed and then a term b is constructed of type B where every occurrence of variable x is substituted by term a. This solution has the problem that when term a is still unknown, the type that term b should have is also unknown. Thus it is impossible to create a hole for term b and setting its type. To solve it, we ask the user to supply term a. Now the type that term b should have, can be computed. Because from the term $\mathrm{inj}(a, b)$ alone it is not possible to derive its type, the type is appended to the term.

Focused hole

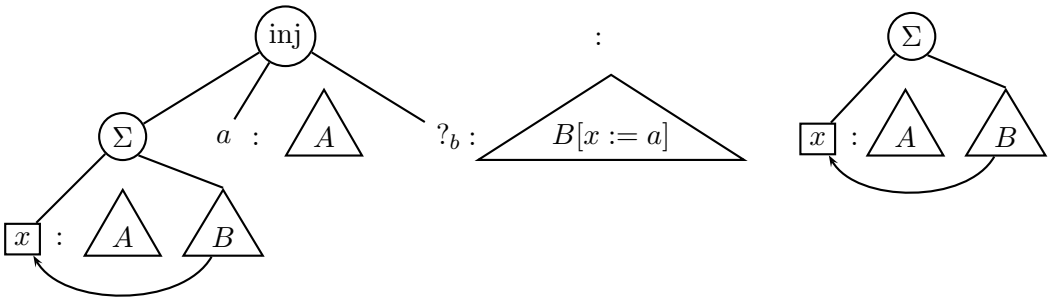


Steps the tactic needs to take

SigmaIntro(a)

1. $?_b :=$ *createHole*()
2. *setType*$(?_b,$ *substitute*$(B, x, a))$
3. *result* := *createInjectionGeneralization*(*copy*(*getType*(Focused hole)), $a, ?_b)$
4. *setType*(*result*, *copy*(*getType*(Focused hole)))
5. return *result*

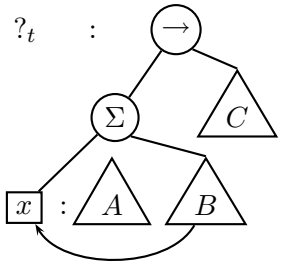
Filled hole after applying tactic



7.7.2 Σ-elimination

This tactic corresponds to the Σ-elim-rule. To construct a term of type $(\Sigma x{:}A.\, B){\rightarrow}C$, we can construct a term of type $(\Pi x{:}A.\, B{\rightarrow}C)$, where x does not occur in C, and then use the bigsplit-operator on it to get a term of the desired type.

Focused hole

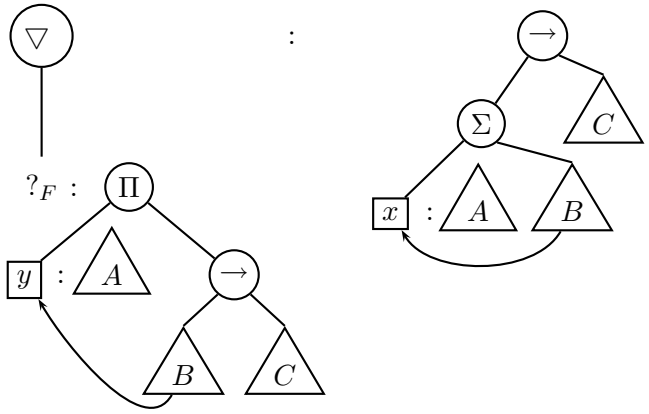


Steps the tactic needs to take

SigmaElim()

1. $?_F := createHole()$
2. $y := copy(x)$
3. $setType(?_F, createPi(y, createArrow(substitute(B, x, createVariable(y)), copy(C))))$
4. $result := createSplitGeneralization(?_F)$
5. $setType(result, copy(getType(\text{Focused hole})))$
6. return *result*

Filled hole after applying tactic



Part III

Implementation

Part III contains implementation details about the proof assistant that has been built during this master project. Chapter 8 contains the overall system design of the proof assistant. Chapter 9 describes how terms, items and contexts are implemented. The type checker that computes the type of terms is described in Chapter 10. The tactics that help to construct a term for a given type is the subject of Chapter 11. The user interface of the proof assistant is described in Chapter 12.

The last chapter in this part is Chapter 13 and describes how the current implementation relates to the objectives that were stated in the introduction, how the proof assistant can be improved and how this master project can be of use for future work.

Chapter 8

System design

This chapter describes the overall system design of the proof assistant that is implemented in this master project. In the first section we state the requirements of the proof assistant. In the second section we explain why it is possible to use the infrastructure of Cocktail for the implementation, instead of the FoolProof components as originally planned. In the last section we divide the system into multiple modules that combined form the proof assistant. Each part of the system is described shortly in this section and more in depth in the chapters to come.

8.1 Requirements

1. The proof assistant allows provable theorems to be proved in an interactive way.
2. The proof assistant is based on PTSs with additional type constructors and the propositions as types principle.
3. Given a context, a term and a type, the proof assistant is able to check whether the term has the given type in the given context.
4. Given a context and a term, the proof assistant can compute the type of the term.
5. Given a context and a type, the proof assistant assists the user in constructing a term that has the given type.
6. The global context can be extended with variable declarations, under the condition that the well-formedness of the global context is maintained.
7. The global context can be extended with definitions, under the condition that the well-formedness of the global context is maintained.
8. Each type constructor can be enabled and disabled independent of the state of the other type constructors.
9. Each reduction rule can be enabled and disabled independent of the state of the other reduction rules.
10. The parameters of the PTS used by the system can be changed.

11. The settings of the type constructors, reduction rules and PTS can be saved to a file.
12. The settings of the type constructors, reduction rules and PTS can be loaded from a file.
13. Theorems can be saved to a file.
14. Theorems can be loaded from a file.
15. The user interacts with the system through text commands.
16. The user interacts with the system by clicking on menus and buttons.
17. Contexts are shown as unstructured text.
18. Contexts are shown as structured text.
19. Contexts are shown graphically
20. Terms and types are shown as unstructured text.
21. Terms and types are shown as structured text.
22. Terms and types are shown graphically
23. The proof assistant is implemented in Borland Delphi 7.
24. The proof assistant uses FoolProof components.
25. The proof assistant has a help function.

8.2 FoolProof and Cocktail

One of FoolProof's main selling points is that it supports manipulating terms with binding structures. For example, copying terms with binding structures such that the bindings in the copy are correct is not straightforward, but FoolProof takes care of this, so the user will not notice that this operation is not trivial.

Cocktail uses terms with binding structures. These bindings are implemented in a way that is similar to how FoolProof handles them. Hence, a proof assistant design that expects that terms with binding structures are implemented in FoolProof's way should only have to be changed a little, if at all, to accept Cocktail's implementation. The FoolProof components were not available in time for this project, but as the Cocktail source code is available, it is used instead.

Cocktail contains a proof assistant that is based on one instance of the PTS framework. This typed lambda calculus does not use the additional type constructors. Cocktail's type checker cannot compute the type of the additional terms. And Cocktail's tactics depend on the internal representation of terms, hence they are not aware of the existence of additional terms. So in this project not everything could be taken from Cocktail's proof assistant: New kinds of terms have been constructed, the type checker is extended so it can type these new terms and new tactics are created that take advantage of the new terms.

8.3 Main modules of the system

The proof assistant implementation that is created during this master project closely follows the structure of Cocktail. At the core of the system there is the symbolic engine that takes care of representing typing judgments and a type checking algorithm, which uses the type derivation rules, to check the correctness of these judgments. To represent a judgment we need representations of terms, items and contexts, and for the type checker we need a representation of sorts, axioms and rules that specify a PTS with additional type constructors. The representations of terms, items and contexts are described in Chapter 9, and the representation of a PTS and the implementation details of the type checker are found in Chapter 10. The symbolic engine is used by the structure editor, which lets the user edit terms by using tactics. This part of the system is described in Chapter 11. The user interacts with the proof assistant through the user interface, which is described in Chapter 12.

Chapter 9

Terms, items and contexts

The proof assistant that is constructed during this master project manipulates terms, items and contexts. Therefore, we need implementations of them. In this chapter we describe how we represent terms, items and contexts, and we define a number of subroutines that operate on them.

9.1 Term and item representation

In Cocktail there are three abstraction levels for the tree representation of terms. The top level defines nodes and the usual tree operators, like getting the number of sons and replacing sons. The second level defines terms, items and bindings on an abstract level. On the third level all the specific terms, which are introduced in Chapter 3 plus hole-terms, are implemented.

The Cocktail source code contains the class Node, which is the superclass of all tree-structures. It provides methods for the usual tree operators and methods for annotating nodes.

Terms are represented as trees, so in Cocktail the class Term is a subclass of Node. Term is the superclass of all PTS terms and hole-terms. It provides methods for calculating with terms, like reducing to normal form, substitution and copying. Node is also the superclass of the class Name, which represents the name of a variable, and the class Item, which represents the name of a variable and its type. Several classes in Cocktail extend Item, but in this project only the subclass Item_Par is used.

BindingTerm is a subclass of Term, which represents the introduction of a typed variable in the context, which can be used within its body. RefTerm is also a subclass of Term and represents a reference to a global or bound variable.

Cocktail has term representations for the ordinary PTS terms: function applications, λ-terms, Π-terms, sorts and variables, called respectively Term_App, Term_Lambda, Term_Pi, Term_Sort and Term_Var. It also has a class Term_Hole for representing holes. These classes are also used in this project. A PTS with additional type constructors also has additional terms. Term representations for these terms are created especially for this project by inheriting from Term and, in the case of Σ-terms, BindingTerm.

Every class that represents a term is built in the same way: It has a couple of fields that contain the representations of the subterms of the term and methods for getting the number of sons, getting a specific son, replacing a son, substituting every reference in the term to a specific item by another term, asking whether the term can be reduced, reducing the term

to normal form, reducing only the term without reducing all subterms, copying the term and getting a string representation of the term. After it was found out that all these methods needed to be implemented, because they are all used and the general implementations by Node and Term are useless in most cases, it was not hard to implement them.

The PTS that Cocktail uses is not extended with the additional type constructors. This means that it does not have an explicit arrow type constructor and as a consequence reducing from a Π-term to an →-term is not supported, because there are no →-terms in Cocktail. To support this feature in our proof assistant implementation without changing the Cocktail source code, a subclass of Term_Pi has been created, which adds this facility. This class is named Term_PiXT, where XT stands for eXtra Types. Term_App has a similar limitation: Cocktail does not support reduction of the application-term when its function-subterm is a split-term or a split-generalization-term. This is fixed, again, by creating a subclass and adding the desired functionality.

A diagram with all the classes for representing terms is shown in Figure 9.1.

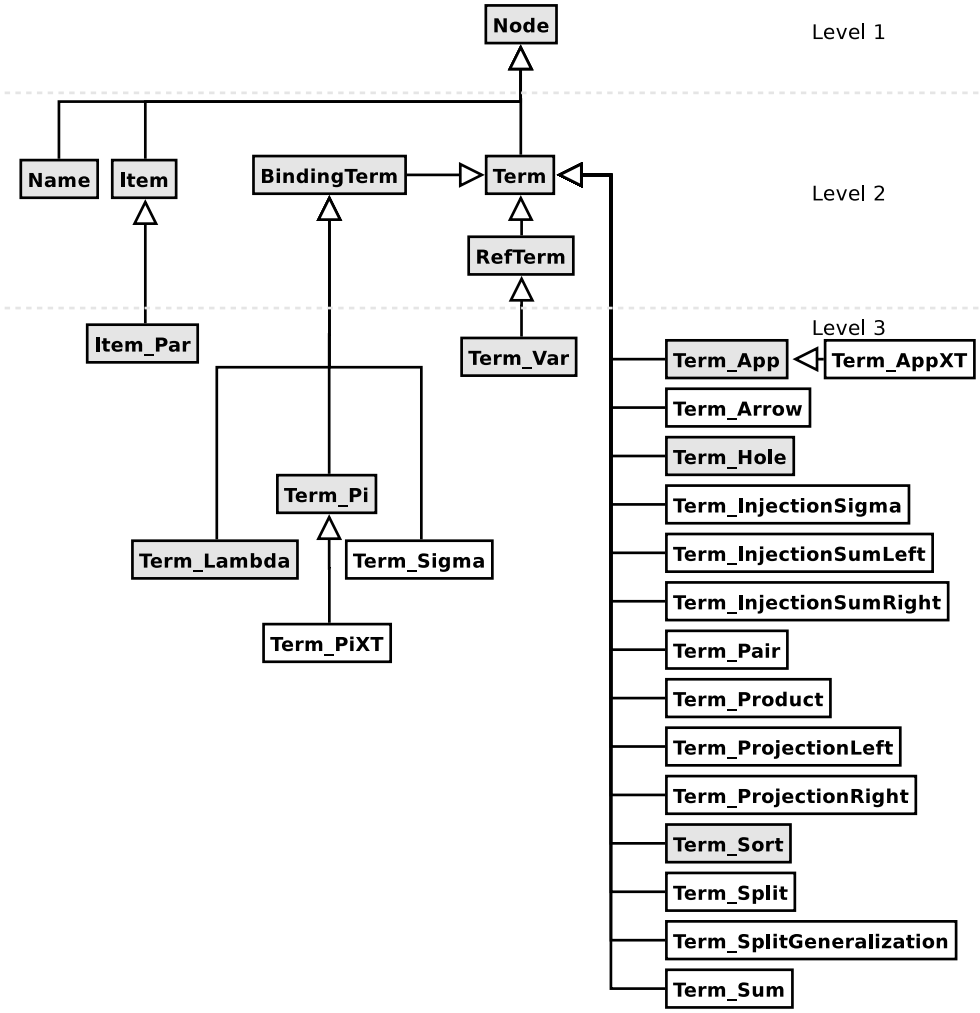


Figure 9.1: The class diagram of the term representation classes. The classes with a gray background are from Cocktail, the other ones are created especially for this project.

9.2 Creating terms and items

In the abstract code model of terms in Chapter 5 we described a number of functions for creating terms. There, we kept it simple by saying every function would return a Node. In the implementation, the methods for creating terms are in the class TermCreator, which came with Cocktail, and a subclass TermCreatorXT that is created for this project. Both classes implement the Abstract Factory pattern. The methods of these classes return an object of a subclass of Term, or when the method creates an item it returns an object of a subclass of Item. For example, in the abstract code we have a function *createProduct*(*ALeft* : *Node*, *ARight* : *Node*) : *Node* and in the Java code we have:

```
public static Term_Product CreateProduct(Term ALeft, Term ARight) {
  Term_Product result = new Term_Product(ALeft, ARight);
  return result;
}
```

The other methods for creating terms are also very similar to the abstract code. However, for creating $\rightarrow$-terms, we have in the abstract code the function *createArrow*(*APremise* : *Node*, *AConclusion* : *Node*) : *Node*. In the implementation this name was already in use by Cocktail for creating implications, which are based on Π-terms. So the method that creates $\rightarrow$-terms does not have the name CreateArrow, but CreateArrowXT instead.

For creating function application-terms a new method called CreateAppXT should be used instead of CreateApp, because the latter one does not return a term that supports reducing $+$-terms and Σ-terms.

To create a Π-term that can reduce to an $\rightarrow$-term, there is a method CreatePi. This method does not have to be called CreatePiXT, because Cocktail does not have a method called CreatePi of its own.

9.3 Contexts

Contexts are implemented by the ItemCollection class, which implements a list of objects of class Item. An empty context is created by calling the constructor of ItemCollection and an item is added to a context by calling the Add method of an ItemCollection object.

The global context can only be extended by the user and the local context is only changed by the system. When the user wants to add an item to the global context, the item is first parsed and then type checked. When this does not cause an error, the system checks whether the name of the item already occurs in the global and local contexts. When this is not the case, the item can safely be added to the global context. This way the validity of the global context is kept invariant.

When the system wants to add an item to the local context and its name already appears in the global or local context, the name of the item is changed by the system to a new name that is not already in use.

9.4 Annotating nodes

In Cocktail, a node can be annotated not only with its type, but also with any other information, which is not used in this project. Therefore, the annotation is implemented by a vector of objects of type Object instead of a single Term object. The Node class has a method SetAnno that, given an index and an object, adds the object to the vector at the specific location. It also has a method GetAnno that given an index, returns the object at the specific location in the vector.

The Term class has a method getType that returns the type of the term. It uses GetAnno to receive the object that is located in the vector at the specific location that is used to store the type annotation. The Term class does not have a method setType. There is a method called SetType in the type checker class, which does more than only annotating a term with its type. This method is discussed in more detail in Chapter 10, which describes the type checker.

9.5 Calculating with terms

Every term has a copy method that returns a copy of the term, including all its annotations. Copying terms such that objects of class RefTerm refer to the correct bound variable is not straightforward. Fortunately, Cocktail takes care of this. A more detailed description of this problem and its solution is discussed by Franssen [10].

Every term also has a substitution method, which is similar to the copy method. Instead of returning an exact copy, it returns a copy in which every reference to a given item is replaced by a copy of a given term.

There are two methods for reducing a term in Cocktail: the method Reduce reduces a term to normal form and the method ReduceThis reduces only a term without reducing its subterms. When the Reduce method is called on a term that does not have a normal form, it will try to reduce it infinitely long. Reduce expects two arguments, a mask and a form, while ReduceThis only expects a mask. The mask lists which reductions are allowed and the form determines to which form the term should be reduced. Initially, in Cocktail the Reduce method was design such that it is also able to reduce the term to head normal form. But the PTS that Cocktail uses has the property that every term is always in normal form [15]. Therefore, terms are never reduced in Cocktail. This had the consequence that the methods for reducing λ-terms and Π-terms were not tested thoroughly until they were needed in this project. Some bugs showed up, which have been fixed. Reducing to head normal form is not used in this project and therefore not implemented.

Equality modulo conversion of two terms is computed by the method Convertible in the class PTSConverter. This class came with Cocktail and is used unaltered. In the abstract model, unification of two terms is only used by the Π-elimination tactic. Cocktail already has a Π-elimination tactic that uses unification. Therefore, unification of two terms is taken from the Cocktail code unaltered.

9.6 Code example

With the classes and methods that are introduced in this chapter, we can construct the tree from Figure 5.1. This Java code is similar to the abstract code example of Chapter 5. anType is an integer constant that holds the value of the index into the annotation vector where the type annotation should be placed.

```
//Create a global context containing items P:* and Q:*.
ItemCollection globalContext = new ItemCollection();
Term s = TermCreatorXT.CreateSort("*");
Item P = TermCreatorXT.CreateItemPar("P", s);
Item Q = TermCreatorXT.CreateItemPar("Q", s.copy());
globalContext.Add(P);
globalContext.Add(Q);

//Create the tree
Term vP = TermCreatorXT.CreateVar(P);
Term vQ = TermCreatorXT.CreateVar(Q);
Item x  = TermCreatorXT.CreateItemPar("x", vP);
Item y  = TermCreatorXT.CreateItemPar("y", vQ);
Term vx = TermCreatorXT.CreateVar(x);
vx.SetAnno(anType, vP.copy());

Term lyTerm = TermCreatorXT.CreateLambda(y, vx);
Term lyType = TermCreatorXT.CreateArrowXT(vQ.copy(), vP.copy());
lyTerm.SetAnno(anType, lyType);

Term rootNode = TermCreatorXT.CreateLambda(x, lyTerm);
Term rootType = TermCreatorXT.CreateArrowXT(vP.copy(), lyType.copy());
rootNode.SetAnno(anType, rootType);
```

Chapter 10

Type checker

10.1 PTS representation

The Cocktail source code contains classes for representing an ordinary PTS by sets of sorts, axioms and Π-rules that specify the PTS. The PTS that Cocktail uses is extended with parametric constants [14]. This extension is implemented in the class CPTS. The constructor of the type checker class in Cocktail expects an object of class CPTS as argument. Because we want to reuse parts of the type checker, the class for representing PTSs with additional type constructors must extend the class CPTS, although parametric constants are not used. This class is called CPTSXT, where XT again stands for eXtra Types. It extends the class CPTS by adding functionality for $\rightarrow$, $\times$, $+$ and Σ-rules.

10.2 Type checker classes

To compute the type of a term, Cocktail constructs the type from scratch. When a term is not typeable an error is raised. When it has computed the type of a node, it uses the method SetType of the type checker class to annotate the node with its type. When the node is already annotated with a type, SetType checks whether the old and the new type are equal modulo conversion. When this is the case, the old type is not overwritten, because the old type might contain additional annotation and when it gets overwritten that annotation is destroyed. When the old and the new type are not equal modulo conversion an error is given.

Cocktail's type checker for ordinary PTS terms is implemented in the class Typer_SD, where SD stands for Syntax Directed. To determine the type of a term t, first the type of every subterm is computed and then these types are combined according to the type derivation rules to construct the type of term t. Typer_SD has a method doTerm that given a term decides which method should be called to type this term depending on the structure of the term. For example, when doTerm is given a λ-term, the method doLambda is called and when doTerm is given a function-application-term, the method doApp is called. Each of these methods is based on a type derivation rule, except for the method that types a hole-term, which just checks if the node is annotated with a type. These typing methods are also used in this master project. [1]

[1] The method for typing a Π-term had a bug that did not appear in Cocktail because of the specific PTS Cocktail uses. This is fixed now.

The class Typer_SD is extended by the class Typer_LPM, where LPM stands for Lambda P Minus or $\lambda P-$, which is the name of the PTS Cocktail uses [15]. Typer_LPM has methods to type terms that only exist in $\lambda P-$, like a term for representing the set of natural numbers and a term which represents the number zero. These terms are not used in the proof assistant that is created during this project.

To compute the type of terms that use the additional type constructors, a class is created that extends Typer_LPM. This class is called Typer_CPTSXT. It could have extended Typer_SD directly, but during the testing of the type checker it was convenient to have built-in terms for the set of natural numbers and an element of this set. Typer_CPTSXT overrides the method for typing a λ-term, because the term might have as type a $\rightarrow$-term but Typer_SD is not aware of the $\rightarrow$-type constructor. The method for typing a function-application-term is also overwritten, because the function-subterm might have as type a $\rightarrow$-term and, again, Typer_SD does not know this type constructor.

Typer_CPTSXT is the superclass of the class Typer_Joystick, which adds the functionality to enable and disable each type constructor. When the user has disabled a type constructor, typing a term that uses this type constructor causes an error. This class is named after Barendregt's joystick [2].

10.3 Code example

In Section 6.2, we showed the abstract code for typing a pair-term. Here, we show how it is implemented. Both are similar, except in the implementation the error value and the context are hold in global variables. And for efficiency reasons, every time an error could have occurred, it is checked to see if it actually did occur and when it did, type checking is aborted.

```
protected void doPair(Term_Pair APair) {
  doTerm(APair.FLeft);
  if (!error) {
    doTerm(APair.FRight);
    if (!error) {
      Term A    = APair.FLeft.getType();
      Term B    = APair.FRight.getType();
      Term type = TermCreatorXT.CreateProduct(A.copy(), B.copy());
      doTerm(type);
      if (!error) {
        SetType(APair, type);
      }
    }
  }
}
```

Chapter 11

Structure editing and tactics

11.1 Structure editor

Cocktail has an inheritance hierarchy of structure editor classes. These classes maintain terms and contexts, and provide all operations on them. Somewhere down the inheritance hierarchy there is a class Editor_Proof that maintains information about a single proof tree. This information includes:

- The focused node, which determines where all actions are targeted.
- The list of all holes in the tree.
- The local context, which is the list of all the items in BindingTerms that are found on the path that starts from the root node and ends with the focused node.

The global context is also maintained by Editor_Proof. Initially, the root node of the proof tree is a hole and its type is an encoding of the theorem that the user wants to prove. Editor_Proof provides access to the tactics, which are used for constructing a proof of the theorem. When a tactic finishes successfully, Editor_Proof replaces the focused node with the result that the tactic returns. The result is a term that might contain typed holes. The fact that Editor_Proof inherits most of its functionality from other classes that are higher in the inheritance hierarchy is not important for this master project.

Editor_Proof has a field of class TacticsManager. The TacticsManager class implements the Factory pattern and it represents a tactics manager that maintains tactics that are added to it. This means that it makes sure that every tactic has access to the same type checker, the same global context and the same local context.

For this project the class Editor_Proof had to be changed. In the constructor of this class the tactics that are used by Cocktail were added to the tactics manager. But in this project other tactics are used and the Cocktail tactics should not get in the way. This has been remedied by letting the Editor_Proof constructor call a newly created method addTactics that adds the tactics to the tactics manager. For this project the class Editor_ProofXTBasic is created that extends Editor_Proof. It overrides the method addTactics so that now only the tactics that have been created for this project are available in the proof assistant.

Editor_ProofXTBasic has a method GetHoles that returns a list of holes. This list is computed by a class in the Cocktail inheritance hierarchy of structure editor classes, but it was not publicly available for the user interface to show the list of holes, until this functionality was added in Editor_ProofXTBasic.

11.2 Tactics

In Cocktail the class Tactic is the superclass of all tactics. It has fields that contain the global context, local context and the type checker. These fields are set by the tactics manager. The Tactic class defines a couple of methods that are overridden by subclasses of Tactic. These methods include the methods valid and apply. The method valid takes a node and returns whether the tactic is applicable on this node. The method apply that takes a node and an object of class Object, which represents zero or more parameters, and returns the result of applying the tactic on the given node. When applying the tactic was successful, the result is a term that might contain typed holes, otherwise the result is null.

For this project the class TacticXT has been created. This class is a subclass of Tactic and the superclass of all the tactics that have been created for this project. Some of the new tactics reduce terms by calling the Reduce method that is defined in the Term class. As already mentioned in Chapter 9, this method needs two arguments that determine which reductions are enabled and to which form should be reduced. The values of these arguments are supplied by TacticXT.

Every type constructor has one or two corresponding introduction rules and one or two corresponding elimination rules. Each of these rules corresponds to a tactic. These tactics are written especially for this project and extend TacticXT, except for the Π-intro and Π-elim tactics. These two tactics are copied from Cocktail and modified a little. The Π-elim tactic is modified, because Cocktail's Π-elim tactic depends on the typed lambda calculus that Cocktail's uses and as a consequence it did not work correctly in the proof assistant that is created during this project. The Π-intro tactic is changed to make it more efficient by removing some case distinctions that were only needed by Cocktail.

11.3 Code example

Implementing the tactics that are described in Chapter 7 was straightforward. This is clear when the abstract code is compared to the implementation code: both are very similar. A difference is that in the abstract code each tactic itself replaces the focused hole-node by the result; in the implementation the tactic returns the result and replacing the hole-node by the result is done by the structure editor. Another difference is that in the abstract code it is assumed every step went without errors; in the implementation error checking is added to the code.

Here is the Java code that implements the ×-intro tactic. Note the similarity between this code and the abstract code.

```
public Node apply(Node focus, Object parms) {
  //focus : (A x B)
  assert valid(focus);

  //?a
  Term holeLeft  = TermCreatorXT.CreateHole();

  //?b
  Term holeRight = TermCreatorXT.CreateHole();
```

```
    //?a : A
    Term typeLeft  = ((Term_Product)((Term)focus).getType()).FLeft.copy();
    holeLeft.SetAnno(anType,  typeLeft);

    //?b : B
    Term typeRight = ((Term_Product)((Term)focus).getType()).FRight.copy();
    holeRight.SetAnno(anType, typeRight);

    //<?a, ?b>
    FResult = TermCreatorXT.CreatePair(holeLeft, holeRight);

    //<?a, ?b> : (A x B)
    FResult.SetAnno(anType, ((Term)focus).getType().copy());

    //focus : (A x B)  <-  <?a, ?b> : (A x B)
    return FResult;
}
```

Chapter 12

User interface

The user interface lets the user interact with the proof assistant. The user interface consists of two windows. The main window, see Figure 12.1, displays the global context, the local context, the proof object and the theorem it proves, the focused goal and the other goals. At the bottom of this window is a text field where the user can enter commands, like entering a theorem, adding global variables and applying tactics. At the top of the window is a menu with a list of all the available commands, a list of all available symbols and how they can be entered with a keyboard, and a menu item that opens the second window.

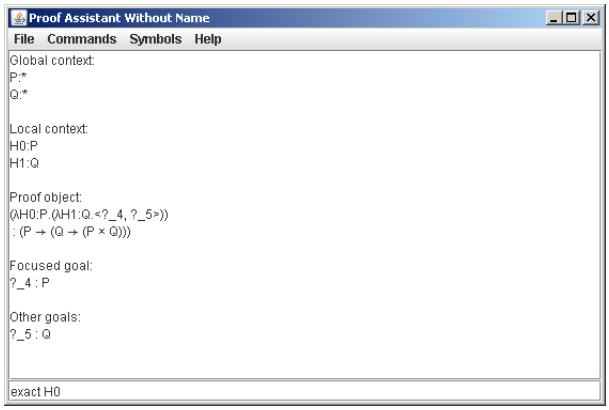


Figure 12.1: Main window

The second window is the settings window, see Figure 12.2. Here the user can set the sorts, axioms and rules that specify the PTS that is used for type checking. Each type constructor can be enabled and disabled. This window also allows the user to set which reduction rules are enabled and which ones are disabled. The settings can be saved to and loaded from a file.

When the user enters a command, the proof assistant parses it and when this succeeds it executes the corresponding action. When an error occurs, a pop-up window with an error message appears. The parser that is being used in this project is not taken from Cocktail, but generated with JavaCC [12] specifically for this project.

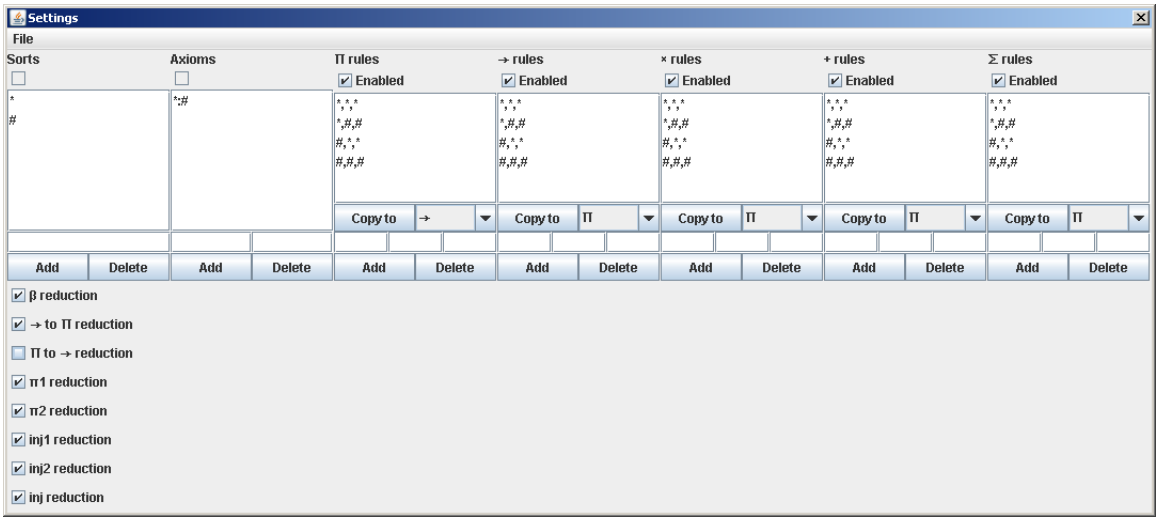


Figure 12.2: Settings window

Chapter 13

Evaluation and future work

At the start of this master project, the main goal was to investigate the usability of the FoolProof components. FoolProof's main selling points include its support for structure editing of terms with binding structures. It was decided that building a proof assistant based on pure type systems and the propositions as types principle would be a good test case for the components, because such a tool makes heavy use of structure editing of terms with binding structures.

After studying the theory of pure type systems and the propositions as types principle, an abstract code model was made of a proof assistant based on that theory. When it turned out the FoolProof components were not going to be available in time and throwing away the abstract code model would be a shame, the decision was made that the proof assistant implementation would use the infrastructure of Cocktail instead. This way, the usefulness of the abstract code model could be tested and when the FoolProof components eventually are ready, the model can be used as the foundation of an implementation that does use FoolProof.

Some problems surfaced while designing the abstract code model, but looking under the hood of Cocktail and seeing how it tackled those problems, surely improved the abstract code model. Mechanisms that have been proven to work in practice by Cocktail, were later incorporated in the abstract code model.

At the end of this project we have an abstract code model of a proof assistant that is based on the pure type system framework extended with additional type constructors and the propositions as types principle, and an implementation that is based on the model. It assists the user in constructing a proof of a theorem that the user has entered. The structure of the proof derivation is captured by a term that is called the proof object. By computing the type of the proof object the whole proof derivation and the theorem are reconstructed. This proof object can be communicated to other type checkers to verify the proof.

At runtime, several settings can be changed by the user that influence which terms can be typed by the type checker. These settings can be saved to and loaded from a file and consist of:

- Which type constructors are enabled.
- Which reductions are enabled.
- Which sorts, axioms and formation rules, which specify the pure type system, are available.

In addition to not using the FoolProof components, some other requirements for the proof assistant were not met. While the user can add variable declarations to the global context, it is not yet possible to add definitions to the global context. Definitions can be seen as shorthand notations for larger terms. However, they do not increase the expressive power of the system.

Theorems and proof objects cannot be saved to and loaded from files in the proof assistant itself. A workaround for saving terms is to select the string representation of a term, copy it to the clipboard and paste it in a text editor that can save it to a file. There is a similar workaround for loading terms that involves using an external text editor, but the term may not contain holes, because for an unstructured term that contains holes it is impossible to compute its type, unless the type of the term is contained as a subterm in the term, like in the case of a $\text{inj}_1(A+B, ?)$-term.

As a consequence of using the Cocktail infrastructure, the proof assistant implementation is written in Java like Cocktail and not in Borland Delphi 7 as was one of the requirements.

The current implementation can be improved by adding support for definitions and saving to and loading from a file of theorems and proofs, as mentioned above. Another improvement could be the way in which terms and contexts are displayed. Currently, terms and contexts are shown as unstructured text, but drawing the proof derivation in flag notation would allow for a more intuitive user interface that supports drag-and-drop operations and it would look nicer as well.

The tactics that are currently implemented in the system are very basic. For example, when the context contains a variable x of type $P \times Q$ and the goal is to construct a term of type P, one way to solve this problem is to first apply the $\times\text{-elim}_1$ tactic with Q as argument. This will change the goal to $P \times Q$. Then, the exact tactic can be used with as x argument and this will solve the goal. There exist other proof assistants that can solve the goal in one step by applying a $\times\text{-elim}_1$ tactic with as argument the value x. The current implementation can easily be extended with such tactics.

It is also important to investigate whether the altered type derivation rules, which form the basis for the syntax directed type checker in the proof assistant, are complete with respect to the original type derivation rules.

Cocktail could be changed such that it uses the term representations that are created during this project, but it would not increase its expressive power.

Of course, the most important thing left to do would be, when the FoolProof components are done, investigate how well the abstract code model translates to an implementation that uses those components. The proof assistant that is created during this project proves that the developed abstract code model can be used as a basis for an implementation.

Bibliography

[1] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volume II*. Oxford University Press, 1992.

[2] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, April 2002.

[3] Marco Brassé. Explorations of a proof-assistant system for λC, July 1992.

[4] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 33(2):346–366, April 1932.

[5] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 34(4):839–864, October 1933. (Second paper).

[6] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[7] The Coq proof assistant. http://coq.inria.fr/.

[8] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[9] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, February/March 1988.

[10] Michael Franssen. *Cocktail: A Tool for Deriving Correct Programs*. PhD thesis, Eindhoven University of Technology, Eindhoven, 2000.

[11] Kees Hemerik. FoolProof: A component toolkit for abstract syntax with variable bindings. Computer Science Report 08/16, Eindhoven University of Technology, Eindhoven, June 2008.

[12] JavaCC - the Java parser generator. https://javacc.dev.java.net/.

[13] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory: From its Origins until Today*, volume 29 of *Applied Logic Series*, chapter 4: Propositions as Types and Pure Type Systems. Springer Netherlands, 2005.

[14] Twan Laan. *The Evolution of Type Theory in Logic and Mathematics*. PhD thesis, Eindhoven University of Technology, Eindhoven, 1997.

[15] Twan Laan and Michael Franssen. Embedding first-order logic in a pure type system with parameters. *Journal of Logic and Computation*, 11(4):545–557, 2001.

[16] Erik Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, Eindhoven, October 1994.

Appendix A

Original problem description

Afstudeeronderwerp: Component-based proof assistant
Student: Frenkel Smeijers
Begeleider: Hemerik

FoolProof is een component library voor het manipuleren van formele talen met bindingsstructuur binnen de Delphi ontwikkelomgeving. FoolProof bevat (of zal bevatten) componenten voor o.a.
- lexical scanning
- syntax highlighting
- parsing
- tree building
- structure editing
- textual views
- structural views
- context management

Een kenmerkend verschil met andere omgevingen zijn de faciliteiten voor het manipuleren van termen met bindingsstructuren, zoals copieren, substitueren, unificeren, etc. Dit maakt de FoolProof omgeving bijzonder geschikt voor formele talen zoals lambda-calculi, logica's, programmeer- en specificatietalen.

Dit afstudeerproject betreft het ontwikkelen van een "proof assistant", die gebaseerd is op typetheorie en het propositions-as-types principe (net als bijvoorbeeld Coq en Cocktail). In essentie werkt dit soort proof assistants op basis van structure editing van getypeerde lambda-termen, al zijn de termen zelf doorgaans niet zichtbaar.
De proof assistant zal waar mogelijk gebruik maken van de beschikbare FoolProof componenten. Het is tevens een studie naar de bruikbaarheid van deze componenten voor de gestelde taak. Het is uitdrukkelijk niet de bedoeling om een automatic prover te ontwikkelen. De proof assistant zal zodanig opgezet worden dat deze zelf als een of meer componenten in het FoolProof systeem opgenomen kan worden.

Als uitgangspunten van dit project dienen:

- De FoolProof componenten;
- Literatuur over typetheorie;
- Een stageverslag van Marco Brass'e:
  "Explorations of a proof assistant system for lambda-C"
- De interactieve prover van het Cocktail systeem