Politecnico di Milano
V Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

A solution of POPLMark Challenge with VCPT

Relatore: Prof. Dino Mandrioli
Correlatore: Ing. Paola Spoletini

Tesi di Laurea Specialistica di :
Diciolla Marco, Matricola: 719633

Anno Accademico 2009-2010

ESTRATTO TESI LINGUA ITALIANA

Al giorno d'oggi siamo completamente immersi in una realtà che ci circonda di oggetti e dispositivi elettronici. L'interazione con tali oggetti è tanto frequente quanto obbligatoria.

Spesso diamo per scontato che tali dispositivi debbano sempre e comunque funzionare correttamente. Ma possiamo veramente avere tale certezza? Possiamo veramente fidarci nel delegare tali congegni elettronici nell'effettuare compiti di importanza vitale? La risposta a tale domande è che solitamente, pur non avendo certezza assoluta riguardo la correttezza di un prodotto software o hardware che sia, lo si delega comunque di responsabilità e/o operazioni critiche.

La conseguenza di una cattiva progettazione nella migliore delle ipotesi, sfocia in una grossa perdita di denaro, ma nella peggiore delle ipotesi oltre al danno economico si affianca anche il disastro umano.Esempi di tragedie causate da dispositivi elettronici sono sfortunatamente ricorrenti nel corso della storia. Il problema principale è che il metodo più comune per verificare la correttezza di programmi è il testing. Il testing, pur essendo un supporto tanto valido quanto utile alla programmazione, non può fornire certezze riguardo il funzionamento di programmi. Infatti come recita una famosa frase di Dijkstra "Il Testing può unicamente mostrare la presenza di errori, no la loro assenza". Per tali motivi, nel corso degli anni, sono state sviluppate rigorose tecniche matematiche/informatiche per garantire maggiore affidabilità e un miglior grado di conoscenza dei sistemi con i quali interagiamo. Tali tecniche prendono il nome di Formal Methods.

I due più comuni approcci nell'ambito dei Formal Methods sono: model checking e theorem proving.

Model Checking: Il model checking è un metodo che consente di effettuare verifiche automatiche di proprietà in sistemi formali. Più precisamente, le tecniche di model checking presuppongono che si abbia a disposizione un modello teorico del sistema che si intende studiare e una rappresentazione matematica (di solito utilizzando logiche) della specifica da verificare. Compito del model checker sarà quindi quello di fornire la risposta "Yes" qualora la specifica sia valida nel modello considerato, oppure "NO" qualora la specifica non sia valida nel modello considerato e in quest'ultimo caso presentare anche un contro esempio.

Theorem Proving: Il theorem proving è un altro metodo che consente di effettuare verifiche formali di specifiche in modelli teorici di sistemi reali. I theorem provers lavorano applicando regole di inferenza a determinate specifiche in modo da derivare alcune proprietà dei sistemi considerati. Si parte quindi da un determinato obiettivo/proprietà da verificare e si applicano regole di inferenza. Le assunzioni delle regole applicate diventeranno quindi i nuovi obiettivi da verificare. Si procede in tale maniera finché non si hanno più ulteriori obiettivi da dimostrare. Tra Model Checking e Theorem Proving non esiste una tecnica prevaricante in assoluto. Per alcuni problemi potrebbe esser adeguato un approccio tramite model checking, per altre un approccio basato su theorem proving. Entrambe le tecniche hanno diversi pro e contro.

Con questa tesi si vuole mostrare che tecniche di theorem proving sono attualmente in grado di aiutare le comunità di sviluppatori software a scrivere codice altamente sicuro e che rispetti specifiche formali. Infatti l'intero progetto si basa su di un meta-linguaggio chiamato System F<: che ingloba molte delle principali caratteristiche dei più comuni linguaggi di programmazione. System F<: è un estensione di ciò che è meglio conosciuto come "polimorphic lamda-calculus" con "records", "record patterns" e "subtyping relation". Riuscire a provare attraverso l'aiuto di un thorem prover alcune proprietà di tale sistema, equivale ad assicurare di poter fare lo stesso per i più ben noti ed utilizzati linguaggi di programmazione. Così un equipe di importanti ricercatori appartenenti principalmente alle Università di Pennsylvania e Cambridge, nel 2005, ha proposto un challenge internazionale che prende il nome di POPLMARK

Challenge. Lo scopo di tale challenge è appunto quello di invitare comunità scientifiche ad utilizzare theorem provers per verificare alcune proprietà critiche di F<: (da loro definite). Così in questa tesi, si è risolto il primo punto del POPLMARK Challenge, "Transitivity of Subtyping Relation", usando come theorem prover VCPT. VCPT è un theorem prover sviluppato dal SICS (Swedish Institute of Computer Science) che utilizza logica del primo ordine con punti fissi e approssimazione per gestire tecniche di ricorsione e "discharge" dei nodi.

Risolvere POPLMARK Challenge con VCPT ha dimostrato quindi due cose essenziali:

1. Che l'utilizzo di theorem provers per verifiche formali è giunto ad un punto maturo; si può pensare quindi di risolvere ed affrontare problemi reali tramite l'utilizzo di tali tools.

2. Che problemi generalmente considerati "complessi" possono essere affrontati e gestiti senza perdere di generalità e senza aumentare il grado di complessità, sia usando logiche espressivamente potenti (come ad esempio HOL) che usando logiche con un potere espressivo ridotto (come appunto nel nostro caso Logica del Primo Ordine).

Abstract

Today theorem proving and program properties verification are gaining more and more importance. Despite encouraging efforts and progresses achieved in the past years, the use of tools for proving properties and verify theories is still not commonplace.

The critical issue is the difficulty to take into account details, rather than conceptual complexity. Furthermore, most proofs are wearisome, long and hard to understand for people lacking of deep knowledge of the problem they are dealing with.

On the other hand, new and more powerful tools have been developed in order to build a bridge between the two communities of automated proof developers and programming language designers. It might be the time to reach a good trade off between efforts to formalize language metatheories and results obtained from that process.

Acknowledgements

This work would not have been possible without the help of many people. First, I would like to express my deep appreciation and sincere gratitude to Professor *Mads Dam*, my supervisor at Royal Institute of Technology (KTH). During the course of this project he was always there to listen and to give advice. He had confidence in me when I doubted myself, and brought out the good ideas in me. He is responsible for involving me in a field of computer science that I had not explored before. He has been constantly helpful and he spent lot of his valuable time to generously guide me with this research project.
A special thank also to Professor *Dilian Gurov*. Even though he was not directly involved with the project, he always cooperated and assisted me throughout this master thesis.
I am grateful to three current Ph.D students of Professor Mads Dam: *Andreas Lundblad*, *Karl Palmskog* and *Musard Balliu*. All of them have been both good friends and valuable colleagues.
Finally, I would like to thank all the Theoretical Computer Science (TCS) group of KTH. They have all contributed in various ways with interesting conversations and great enthusiasm.

Ringraziamenti

Giunto al termine di questo lavoro desidero espriemere la mia gratitudine a tutte le persone che in modi diversi hanno contribuito allo svolgimento di questo progetto. Primo su tutti vorrei ringraziare il Professor *Dino Mandrioli*, mio supervisore al Politecnico di Milano. Professor *Mandrioli* è stato sempre disponibile e pronto ad aiutarmi in ogni circostanza. Seppur molto impegnato ha sempre trovato il tempo per dedicarmi attenzione e fornirmi consigli utili per poter portare a termine questo lavoro di tesi.
Vorrei inoltre ringraziare la Dottoressa *Paola Spoletini* che ha contribuito a questo progetto con numerosi commenti e preziosi suggerimenti.

Questa tesi è dedicata alla mia famiglia, a ciascun membro di essa che ha contribuito supportandomi in ogni decisione sin dalla più tenera età. Li ringrazio non solo per il sostegno economico che mi ha permesso di coronare il sogno di conseguire la laurea, ma per quell'aiuto tacito ed esplicito proveniente dal loro cuore. Tutte le volte che mi hanno visto in difficoltà, tutte le volte che mi hanno trovato preso da un esame o da questa tesi e tutte le volte che il mio umore toccava il fondo, loro sono sempre stati lì pronti ad aiutarmi, hanno sempre avuto parole di conforto e hanno sempre trovato il giusto modo per donarmi quella carica che mi ha permesso di stringere i denti e non mollare.
Un sincero grazie di cuore al primo e per me più grande maestro di vita, mio padre *Raimondo*. Sin da adolescente è stato per me non solo genitore, ma anche amico e confidente più caro. Persona che riesce a tramutare ogni discussione in un valido strumento di arricchimento interiore e che in ogni circostanza, positiva o negativa che sia, ti fornisce quel caloroso supporto necessario per poter andare avanti.
Altra persona speciale a cui voglio dedicare un ringraziamento particolare è mia madre *Maria*. Premurosa, attenta ed amorevole. Piena di preoccupazioni, quelle preoccupazioni che solo una madre può avere nei confronti del proprio figlio. Costantemente presente, sempre disponibile e sempre pronta a farti sentire amato in un mondo che troppo spesso trascura i sentimenti per dar spazio a pensieri materialistici. A lei ringrazio per avermi insegnato ad amare e a prendermi cura delle persone a me care.
Estrema gratitudine anche ai miei due fratelli *Giuseppe* e *Gianluca*. Loro mi hanno guidato ed inspirato durante tutti questi anni. Ho appreso ed imparato molto da ciascuno dei due; entrambi in maniera differente hanno saputo incoraggiarmi ed aiutarmi nei momenti di difficoltà. Spero con questa tesi di poter ripagare, almeno in parte, tutti i profondi sacrifici che la mia famiglia ha dovuto sostenere dall'inizio alla fine di questo percorso di laurea.

Infine vorrei ringraziare tutte quelle persone che hanno condiviso con me questi cinque anni di studio; persone con le quali ho trascorso momenti stupendi, discussioni animate e serate incredibilmente pazze. Tutti in un modo o nell'altro hanno contribuito al raggiungimento del mio obiettivo, la laurea, e per questo vi sonoe vi sarò per sempre debitore.

Contents

List of Tables

Chapter 1

Introduction

In modern society people are surrounded by computers and electronic devices. They are everywhere and we trust them for a lot of critical tasks in our daily life. Over the years, examples of incorrectly functioning software have shown the risks connected to faulty programs. For example, in 1962, the NASA shuttle Mariner 1 was intended to perform a Venus flyby. The vehicle veered off course and it was destroyed 293 seconds after the lunch due to the risk of crashing the shuttle in the North Atlantic shipping lanes or in an inhabited area. According to the NASA report a missing hyphen in coded computer instructions caused the change of direction. The cost of the operation was about 18,500,500 dollars.

It is not enough to write software that is correct and works as expected in 95% of times when dealing with critical-safety system such as on board of Mariner 1. The system must function correctly all the time. But how is it possible to achieve the ambitious result of proving the complete correctness of software? How can it be proven that programs will always function as expected?

Probably there will never be answers for these questions, but some mathematical techniques such as Formal Methods can at least increase the understanding of systems and reveal inconsistencies, ambiguities and incompletenesses that might otherwise be overlooked. More specifically, in computer science, Formal Methods are usually referred as mathematically-based techniques for specification and verification of software and hardware systems. The system is modelled by mathematical objects and its functions become properties of the model. With the help of mathematics, deciding whether the system is correct or faulty is translated into deciding whether some properties hold or not in the model of the system. There are many variants and styles that conform to this basic scheme. For example, the model of the system might be large or small, the properties to verify might be vague or detailed and the verification might be manual or automatic. Deciding which is the best approach for all problems is impossible. Handling things in a specific way could be appropriate for one system and its related properties but not for another. Since the importance of verifying and demonstrating correctness of systems has been discussed so far , why are formal method techniques not commonplace everywhere yet? The simplest answer is that formal methods are not easy and simple; verifying that a certain property holds in a system might be intrinsically hard. Besides, industries are often interested in producing cheap software, but not completely secure, rather than software that is reliable but more expensive.

The two most common approaches for verifying systems properties are: *theorem proving*, and *model checking*.

Model Checking: Model checking was proposed in the 1980s independently by Clarke and Emerson [CE82] and by Quielle and Sifakis [QS82]. In a broad view, model checking is seen as an automatic or semiautomatic process to verify systems. Model checking assumes an available

mathematical model of a system, which is represented by computer memory, and a given formal specification in some logical formalism and then it compares the given model with the given specification. The answer of the checker is either 'YES the specification is valid in the model' or 'NO the specification is not valid in the model' and in the latter case it gives a counterexample. The major problem of model checking is related with the space explosion in modelling systems. The state space explosion problem in model checking is a result of the exponential growth of the number of states in a system. The hardware on which the model checker runs should allow to store the necessary number of states. Unfortunately a lot of times this is not achievable. Over the years, some techniques have been proposed to reduce the number of states.
Theorem Proving: Theorem proving is another method for performing verification on formal specifications of system models. Theorem provers apply inference rules to specifications in order to derive system properties. The theorem proving tools consist of a powerful collection of inference steps that can be used to reduce a proof goal to simpler sub-goals that can be automatically discharged by the primitive proof rules of the prover. The major disadvantages of theorem proving is that if the proof fails, it does not mean that the property is not valid in the specification. It might happen that the user has not been good enough on finding the right proof strategy to reach and prove the intended goal. On the other side, problems that are not solvable using model checking (for example due to the model space explosion) might be provable with theorem proving (or vice versa).

However, it is impossible to give an answer to which approach between theorem proving and model checking is the best for verifying properties. There exists problems where theorem proving approaches best fit in demonstrating properties and problems for which model checking techniques are more indicated.

1.1 Aim

The aim of this thesis is to solve the first point of what is so-called *POPLMark Challenge*. The challenge is based on an extension of the typed λ-calculus with subtyping relation and records. Basically the challenge wants to show that theorem proving can assist researchers and software developers in writing safe code.

The theorem prover VCPT (VeriCode Proof Tool) has been used to solve the challenge. VCPT was developed at the Swedish Institute of Computer Science (SICS), and has powerful features for proving inductive assertions.

1.2 Scope

The scope of the thesis is strictly related to the point of the *POPLMark Challenge* addressed. The whole project has been developed focusing on the relevant aspects of the part that has been solved.

However, this project can be extended further. In fact, the thesis can be viewed as a starting point, in order to cover also the other parts of the challenge, thanks to the code reuse.

1.3 Overview

This thesis starts off with a presentation of the challenge in Chapter 2. It is discussed why it is important dealing with issues that the challenge concerns, and there is a detailed presentation of the challenge. Basic concepts of background materials are introduced in Chapter 3. There are presented theoretical arguments like first-order logic, μ-calculus and fixed points. Chapter

4 gives an accurate description of the tool used (VCPT). There are shown tactics and tacticals that users can handle while proving properties with VCPT. Chapter 5 is the core of this thesis. It contains our solution of the challenge in terms of theory and lemma used. Some examples are discussed in order to better understand the strategy applied to solve the challenge. Chapter 6 draws lines of possible future works.

Chapter 2

The *POPLMark Challenge*

In this chapter we describe in detail the *POPLMark Challenge* and the reason why it is important to make efforts in solving the issues that it concerns.

The basic goal of the challenge is to show that we are at the point in which theorem proving can really be helpful to assist researchers and software developers in writing safe code.

We refer to paper [ABF$^+$05] for descriptions of the most common approaches used to solve the challenge. In [ABF$^+$05] there are also guidelines and paper-proofs that help users in solving the *POPLMark Challenge* .

Since the challenge is designed to cover some critical aspects of system $F_{<:}$ [Pie02], we need to introduce what system $F_{<:}$ is, and what are the basic issues that arise in formalizing key properties of programming languages in this particular case.

System $F_{<:}$ is basically an enrichment of polymorphic λ-calculus (simple λ-calculus with universal quantifier over types) with subtypes, records and record patterns.

In the next sections we give first a short introduction to the λ-calculus (Section 2.1) and after that, we formally present the challenge (Section 2.2).

2.1 λ-Calculus

Functional abstraction is a key feature of most programming languages. The λ-calculus [Bar07] embodies this concept in a pure form in order to make the study of functional computation easier.

In untyped λ-calculus everything is a function, and the syntax is composed by the grammar in Table 2.1:

$t :=$		terms
	x	variable
	$\lambda x.t_1$	abstraction
	$t_1\ t_2$	application

Table 2.1: Syntax of λ-calculus

In the above syntax, a simple variable x (identified by a name) is a term, if t_1 is a term and x a variable then $\lambda x.t$ is a term (an abstraction), if t_1 and t_2 are both terms then $t_1\ t_2$ is also a term (an application). Nothing else is a λ-term.

Intuitively, a λ-abstraction $\lambda x.t_1$ represents a function that takes an input and applies the function denoted by t_1 on that input. We say that λ binds x in t_1. An application $t_1\ t_2$,

instead, is seen as the application of the function t_1 to the function t_2. We can think of t_2 as the parameter given to t_1.

Using these three simple concepts it is possible to formalize a lot of the most important features of programming languages and mathematics.
For example, suppose we want to represent in λ-calculus the identity function id that takes a parameter x as input and always returns x: $id(x) = x$. With the syntax previously introduced we could write $id = \lambda x.x$.

According to Table 2.1, λ-abstractions are unary functions, i.e. functions on one parameter. Nevertheless, in mathematics, functions can receive any number of inputs. To overcome this mismatch, there exists a mathematical technique which transforms a multi arguments function, in a chain of functions, each with one single parameter. This technique is called *currying*.

In order to clarify what currying is, we present a short example. Let f be the binary function: $f(x, y) = x * x + y * y$. Function f can be expressed in λ-calculus using two abstractions instead of one: $\lambda x.\lambda y.(x * x + y * y)$.

A variable x that is not in the scope of any binder λ, is called a *free variable*. The free variables of a generic term are defined recursively as in Table 2.2:

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda x.t_1) & = & FV(t_1)\backslash\{x\} \\ FV(t_1\ t_2) & = & FV(t_1) \cup FV(t_2) \end{array}$$

Table 2.2: Free variables of a λ-term

The only way in which terms compute is the application of a function to arguments. More specifically, an application $t_1\ t_2$ can take a computational step, if the left side term is a λ-abstraction $t_1 = \lambda x.t$. We may substitute the right hand term for the bound variable in the abstraction body. We write that: $(\lambda x.t)t_2 \rightarrow t[t_2/x]$. The term $t[t_2/x]$ is the function obtained by substituting all the free occurrences of x in t by t_2.

It is clear that the free variables of terms play a central role in substitutions. Each time a substitution takes place capture of variables should be avoided. Thus, for example, after the substitution of t_2 in the term t, written as $t[t_2/x]$, all free occurrences of a variable in t_2 must remain free and all the bound occurrences of a variable in t must remain bound.

Let x, y, s be variables, and let t_1, t_2 be terms, as in Table 2.3:

$$\begin{array}{lcll} x[s/x] & = & s & \\ y[s/x] & = & y & \text{if } x \neq y \\ (\lambda y.t_1)[s/x] & = & \begin{cases} \lambda y.t_1 & \text{if } x = y \\ \lambda y.(t_1[s/x]) & \text{if } x \neq y \text{ and } y \notin FV(s) \end{cases} & \\ (t_1\ t_2)[s/x] & = & (t_1[s/x])(t_2[s/x]) & \end{array}$$

Table 2.3: Partial substitution function in λ-calculus

Notice here, that Table 2.3 shows a partial substitution function in λ-calculus. In fact, the case in which $x \neq y$ and $y \in FV(s)$ is not covered. In order to make the substitution function total, it is possible to work with terms up to renaming of bound variables.
The process of renaming bound variables is called α-conversion. Let x, y be λ-variables, and let $t = \lambda x.t_1$ be a λ-term. An α-conversion of term t consists in the substitution in t_1 of all the free occurrences of x with y. The result of the α-conversion is a new λ-term t' such that $t' = \lambda y.(t_1[y/x])$. The terms t and t' are α-congruent and we write $t \equiv_\alpha t'$. More specifically, two terms t and t' are α-congruent ($t \equiv_\alpha t'$), if there exists a chain of α-conversions from t to t'

Intuitively, the name of bound variables can change whenever it is convenient. For example, if we want to calculate $(\lambda y.xy)[yz/x]$, we first rewrite $(\lambda y.xy)$ as $(\lambda w.xw)$ and after we calculate $(\lambda w.xw)[yz/x]$ giving as result $(\lambda w.yzw)$.

Unfortunately some problems arise from renaming variables. Suppose that an α-conversion of the term $t_1 = \lambda x.\lambda x.x$ returns the term $t_2 = \lambda y.\lambda x.y$ renaming with y the first and third occurrences of variable x in the abstraction t_1. The terms t_1 and t_2 have not the same meaning, so they are not interchangeable. In the same fashion, renaming the variable x with variable y in the term $t_1 = \lambda x.\lambda y.x$, would produce the term $t_2 = \lambda y.\lambda y.y$, that again has different meaning from t_1. The examples above suggest that the choice of the new variable to substitute must be careful. More specifically, let x,y be variables, let t_1 be a λ-term and let t be a λ-abstraction such that $t = \lambda x.t_1$. To substitute the bound variable x with the variable y in t it must be the case that:

- y does not appear freely in t_1.
- y is not bound by a λ in t_1

Convention 2.1.1 α-congruent terms are interchangeable in all contexts.

Thanks to Convention 2.1.1, the substitution function can be rewritten as total function, as in Table 2.4:

$$\begin{array}{lcll} x[s/x] & = & s & \\ y[s/x] & = & y & \text{if } x \neq y \\ (\lambda y.t_1)[s/x] & = & \lambda y.(t_1[s/x]) & \\ (t_1\ t_2)[s/x] & = & (t_1[s/x])(t_2[s/x]) & \end{array}$$

Table 2.4: Substitution function in λ-calculus

2.1.1 De Bruijn Indices Representation

In order to make λ-calculus more machine oriented, de Bruijn in 1972 proposed a new technique to formalize the concepts of the λ-calculus [dB72]. The technique is known as *de Bruijn Indices Representation*.

The idea is to represent bound variables in a canonical way up to α-congruences. Although the two λ-terms $\lambda x.\lambda y.(yx)$ and $\lambda z.\lambda y.(yz)$ are different, they can be considered as the same term, with respect to an α-conversion $\lambda x.\lambda y.(yx) \equiv_\alpha \lambda z.\lambda y.(yz)$. The idea of de Bruijn, then, was to find an unique representation for both terms $\lambda x.\lambda y.(yx)$ and $\lambda z.\lambda y.(yz)$.

Variables will be represented by natural numbers and not by names. If there is a bound occurrence of variable x in a term t, for example in $\lambda x.x$, then the bound occurrence of x is replaced by the natural number k. The number k indicates to which binder λ the bound occurrence of x refers to. Thus, the term $\lambda x.x$ becomes $\lambda.0$ in de Bruijn representation.

In the same fashion, a natural number has to be assigned to each free variable of terms. The idea is to introduce a *naming context* Γ. The naming context Γ assigns once and for all a number to each free variable. When we need to represent a term with free variables, we choose a number that is consistent with that assignment. Each free variable x in a term t, will be represented by a number z such that: $z =$ *number of variable* x *in the context* $\Gamma +$ *number of* λ *binders over* x. Some examples will explain the concept better.

Suppose that the naming context Γ is the same of Table 2.5.

$$\begin{array}{rcl} \Gamma & = & x \rightarrow 2 \\ & & y \rightarrow 1 \\ & & z \rightarrow 0 \end{array}$$

Table 2.5: Example of context

The term $x(yz)$ will be represented as 2(10) because all the variables are free and there are no binders, so variables take the number that they have in the naming context Γ. On the other hand, the term $\lambda x.zx$, in which variable x is bound and variable z is free, will be represented as $\lambda.10$. The number 0 (for x) refers then to the first λ binder, and the number 1 (for z) is the number of the variable z in the context Γ, plus the number of binders over z (so just one in the example). Following this approach it is possible to represent any λ-term [dB72].

Shifting and Substitution

In section 2.1, it has been showed how dealing with λ-calculus needs substitution mechanisms over terms. The same concept has to be defined now, using de Bruijn representation. One auxiliary operation called *shifting* is required. Shifting gives new numbers to free variables within terms, after substitutions.

The problem is that when a substitution takes place, for example in $(\lambda y.x)[s/x]$, the term s, after the substitution, will be enclosed under one more binder. According to the de Bruijn representation, in order that variables keep referring to the same names after substitutions, all the free variables of s have to be shifted by one, because we are adding one more binder to the context of s.

Finding the free variables inside a generic term t expressed using de Bruijn representation, is not a trivial operation. We need a parameter called *cutoff*, c, to keep track of how many binders we cross from the outermost term t, to subterms of t. In this way it is possible to determine whether a variable is bound or free in any subterm of t. We need also another parameter d representing the increment to give to the variable that we are shifting.

The notation $\uparrow_c^d (t)$ is used to indicate the d-place shift of term t from cutoff c.

Formally a d-place shift of term t from cutoff c is defined recursively as in Table 2.6:

$$\begin{array}{lcl} \uparrow_c^d (k) & = & \begin{cases} k & \text{if } k < c \\ k + d & \text{if } k \geq c \end{cases} \\ \uparrow_c^d (\lambda.t_1) & = & \lambda.\uparrow_{c+1}^d (t_1) \\ \uparrow_c^d (t_1\ t_2) & = & \uparrow_c^d (t_2)\ \uparrow_c^d (t_2) \end{array}$$

Table 2.6: Shifting function

Once we have defined the shifting function, we are ready to define the substitution.

The substitution of a term s for variable number j in a term t, written $t[s/j]$ is defined as in Table 2.7:

$$\begin{array}{lcl} k[s/j] & = & \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases} \\ (\lambda.t_1)[s/j] & = & \lambda.(t_1[\uparrow^1 (s)/j + 1]) \\ (t_1\ t_2)[s/j] & = & (t_1[s/j])(t_2[s/j]) \end{array}$$

Table 2.7: Substitution function with de Bruijn representation

Let c, d, x be natural numbers, let t, s be generic nameless λ-terms, then Appendix A shows a pen and paper proof of Lemma 2.1.2:

Lemma 2.1.2 $\uparrow_c^d ([s/x]t) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (t)$

Intuitively, Lemma 2.1.2 states that the result of the d-place shift from cutoff c of the term obtained by the substitution of x with s in t, is the same result obtained by first d-shifting from cutoff c all the terms x, s, t and after computing the substitution.

Let t be a λ-term, let Γ be an environment and let $rm_\Gamma(t)$ be the function that takes a λ-term, t, and returns the nameless representation of t. Appendix A shows also a more general lemma (Lemma 2.1.3).

Lemma 2.1.3 $rm_\Gamma([s/x]t) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]rm_\Gamma(t)$

2.2 The Description of the Challenge

The goal of *POPLMark Challenge*, as we described in the beginning of this chapter, is to show that theorem proving can finally become commonplace and help researchers and software developers in writing safe code.

The challenge covers some critical features of most programming languages such as:

- **Binders**: Binders are a centerpiece of programming languages. In order to be type-safe, each programming language should at least ensure in its semantics mechanisms to handle α-conversions and to avoid to capture variables after substitutions.
- **Complex induction**: Programming languages supports recursive definition's mechanisms. One of the key feature of the meta-theory proposed, is to stress this aspect.
- **Component reuse**: Another important aspect of the challenge, is to show how the reuse of components can be simple and convenient at the same time. The theory built should be extendable for future projects.

All the above characteristics of programming languages are taken into account in the challenge, and a good solution covering all the points of the challenge, would be applicable across a wide range of programming language theory.

POPLMark is based on system $F_{<:}$. System $F_{<:}$ extends the polymorphic λ-calculus (simple λ-calculus with universal quantifier over types), with subtypes, records, record patterns and subtyping relation for records. Since the challenge is widely based on Pierce's book *Types and Programming Languages* more information can be retrieved from there, or from the papers of Cardelli et al [CMMS94] and Curien and Ghelli [CG94].

The challenge has been divided into three different parts:

1. The first deals with the type language of system $F_{<:}$.
2. The second deals with terms, evaluation and type soundness.
3. The third concerns semantics issues and uses results from part 1 and part 2 for *testing* and *animating*.

Parts 1, 2 and 3 above, are successively divided into two parts: the first that considers the pure system $F_{<:}$, and the second that considers the system $F_{<:}$, enriched with records.

The strategy that should be used for solving the challenge, is to start small, and then thanks to the reuse, extend the work.

It is possible to use any kind of formalization for solving the challenge. The choice should be wise, according the theorem prover used. A good formalization is the one which is intuitive and convenient respect to the tool adopted.

There are solutions developed using de Bruijn indices representation, solutions with locally nameless representation [Cha09] and solutions that addressed the challenge using names for variables.

Evaluating which formalization is more adequate for the challenge is really hard and probably there is no answer. Nevertheless, practical experience has shown that the locally nameless representation is the one that best fits for the challenge, representing a good trade off between the machine-oriented approach followed using de Bruijn indices and the user-friendly solution with names.

No matter which formalization is used, we must always operate on types that are well formed in the environment considered. Intuitively, a type T is well formed in the environment Γ, if all the free variables of T are in the domain of Γ. The symbol $\Gamma \vdash T$ will be used to express the well-formedness judgment of a type T with respect to the environment Γ.

Since we addressed only the first part of the challenge (called part 1.a) considering the pure system $F_{<:}$, point 2 and 3 of the above listing will not be described anymore.

2.2.1 Challenge 1.a: The Transitivity of Subtyping

The part 1.a of the *POPLMark Challenge*, deals with system $F_{<:}$ without introducing records. The basic elements of system $F_{<:}$ are types and environments. The syntax of types and environments is expressed respectively in Table 2.8 and in Table 2.9.

T	::=		types
		X	type variable
		Top	maximum type
		$T_1 \to T_2$	type of functions
		$\forall X <: T_1.T_2$	universal type

Table 2.8: Syntax of the types

In $\forall X <: T_1.T_2$ of Table 2.8, the variable X (subtype of variable T_1) is a binding occurrence with scope T_2 (X is not bound in T_1). The type $\forall X <: T_1.T_2$ is essential to express polymorphic types. In fact, if a function f has type $\forall X <: T_1.T_2$, it means that for any type X given to f, f will always returns an object that is of type T_2.

Γ	::=		type environments
		$\emptyset$	empty type environments
		$\Gamma, X <: T$	type variable binding

Table 2.9: Syntax of the environments

Although the syntactical rules are intuitive, showing some of their properties could require sophisticated reasoning.

The first part of the challenge concerns the transitivity of subtyping. Informally speaking, the subtyping relation states that if two types T and S are in subtyping relation, we write $T <: S$ (T is subtype of S), then it is possible to use an instance of T anytime an instance of S is required.

Subtyping Relation

$$\Gamma \vdash S <: Top \qquad (SA\text{–}Top)$$

$$\Gamma \vdash X <: X \qquad (SA\text{–}Refl\text{–}TVar)$$

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T} \qquad (SA\text{–}Trans\text{–}TVar)$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \qquad (SA\text{–}Arrow)$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1.S_2 <: \forall X <: T_1.T_2} \qquad (SA\text{–}All)$$

Table 2.10: Subtyping Relation

The subtyping relation is defined as the least relation closed under the rules in Table 2.10.

Let X be a type variable, U, T, T_1, T_2, S_1, S_2 be generic types and let Γ be an environment. Rule $(SA\text{–}Trans\text{–}TVar)$ states that if X is subtype of U in Γ, and U is subtype T in Γ, then it must be also the case that X is subtype of type T in Γ. Rule $(SA\text{–}Arrow)$ asserts that if in Γ, T_1 is subtype of S_1 and S_2 is subtype of T_2, then it must be valid that the type function $S_1 \to S_2$ is subtype of the type function $T_1 \to T_2$ in Γ. Rule $(SA\text{–}All)$ states that if T_1 is subtype of S_1 in Γ and if S_2 is subtype of T_2 in $\Gamma, X <: T_1$ (environment Γ enriched with one more variable type binding $X <: T_1$), then it must be the case that $\forall X <: S_1.S_2$ is subtype of $\forall X <: T_1.T_2$ in Γ.

These rules are meant to help users in the formalization. In fact they are the translation of the declarative form of the subtyping relation into an algorithmic representation.

The lemma to show is the following:

Lemma 2.2.1 (Transitivity of Algorithmic Subtyping) If $\Gamma \vdash S <: Q$ and $\Gamma \vdash Q <: T$, then $\Gamma \vdash S <: T$

In order to consider part 1.a of the challenge solved, another property called *narrowing*, has to be proved simultaneously with transitivity.

If two environments Γ, Γ' are in narrowing relation, then for all types S, T such that $\Gamma \vdash S <: T$, it is the case that $\Gamma' \vdash S <: T$. In other words, if for any two environments Γ, Γ' the narrowing relation holds, then if a type S is subtype of another type T in Γ, it is also valid that S is subtype of T in Γ'.

Lemma 2.2.2 (Narrowing) If $\Gamma, X <: Q, \Delta \vdash M <: N$ and $\Gamma \vdash P <: Q$, then $\Gamma, X <: P, \Delta \vdash M <: N$

Appendix A of [ABF+05] presents a pen and paper proof that shows one of the proof-strategies that can be adopted for solving the challenge. The approach that has been used in this thesis will be presented in next chapters.

Chapter 3

Background

The aim of this chapter is to give a short introduction to the background material necessary for this thesis.

Studies of theorem proving are concentrated into two different themes : *Automated Theorem Proving* and *Interactive Theorem Proving*.

The differences between the two approaches are suggested directly from words *automated* and *interactive*, in fact:

- The focus of automated theorem proving is to automatically prove the validity of a wide range of mathematical theorems, with help of computer programs.
- On the other hand, interactive theorem proving aims to reach the same goal, concentrating the efforts of humans and tools together.

Theorem provers use logics to prove formulas and depending from the logic adopted, the problem of deciding whether a formula is valid or not varies from trivial to impossible. It is clear then that logics have a central role in theorem proving. Since VCPT uses first order logic with explicit fixed point, we focus on that.

Next sections introduce the concepts of *First Order Logic* (Section 3.1), *First Order μ-calculus* (Section 3.2) and *Gentzen-Type System for first order μ-calculus* (Section 3.3).

3.1 Introduction to First-Order Logic

First order logic is a formal logic widely used in computer science and mathematical fields. Basically first order logic differs from propositional logic for the introduction of quantifiers and domains over quantifiers range.

The use of first order logic is attractive due to the number of existing correct deductive systems (set of *axioms* and *inference rules*).

Next subsections describe two important features of all logics: *syntax* and *semantics*.

3.1.1 Syntax of First Order Logic

The syntax of a logic, consists in all the symbols that can be used within the logic.

Regarding first order logic it is possible to express the syntax as combination of *Alphabet* and *Formation Rules*.

Alphabet

The alphabet of first order logic is composed by:

- **Constant Symbols**: Which represent individual elements in the formalization.
- **Variable Symbols**: Entities ranging over a specific domain.
- **Logical Connectives**: $\wedge$ for conjunction, $\vee$ for disjunction, $\rightarrow$ for implication, $\leftrightarrow$ for biconditional and $\neg$ for negation.
- **Quantifiers**: $\forall$ for universal quantifier and $\exists$ for existential quantifier.
- **Auxiliary Symbols**: Parentheses, brackets and other punctuation symbols.
- **Predicates**: Usually represented through uppercase letters. They express truth values over individual elements.
- **Terms**: Usually represented through lowercase letters. They represent relations among individual elements.

Not all symbols are required and it is possible find some slight differences representations in literature.

For convenience, precedence rules for logical connectivities have been developed (although they are not necessary). Hence, the following order for connectives will be considered: $\neg,\wedge,\vee,\forall,\exists,\rightarrow,\leftrightarrow$.

Formation Rules

Formation rules define how to construct *terms* and *formulas* using the first order logic alphabet.

Inductively the set of *terms* are defined by the following rules:

- **Constants**: Any constant is a term.
- **Variables**: Any variable is a term.
- **Functions**: If $t_1,t_2,..,t_n$ are terms and f is a function of arity n, then also $f(t_1,t_2,...t_n)$ is a term.

Nothing else is a term.

In the same fashion we define inductively the set of *formulas* by the following rules:

- **Predicate Symbols**: If P is a predicate of arity n and $t_1,t_2,...,t_n$ are terms, then $P(t_1,t_2,...t_n)$ is a formula.
- **Negation**: If ϕ is a formula, then $\neg\phi$ is a formula.
- **Binary Connectivities**: If ψ and ϕ are formulas then also : $\psi \wedge \phi$, $\psi \vee \phi$, $\psi \rightarrow \phi$, $\psi \leftrightarrow \phi$ are formulas.
- **Quantifiers**: if ϕ is a formula and x is a variable, then $\forall x.\phi$ and $\exists x.\phi$ are formulas.

Nothing else is a formula.

Once formulas and terms are defined, it is possible to inductively define the set of *atomic formulas*, as follows:

- If t_1 and t_2 are terms, then $t_1 = t_2$ is an atomic formula.

- If $t_1, \ldots, t_n$ are terms, and P is an n-ary predicate, then $P(t_1, \ldots, t_n)$ is an atomic formula.

We introduce now the concepts of *free* and *bound* variables for any first-order formula ϕ. Intuitively we say that a variable is free if it is not quantified, so for example the variable y in $\exists x.\phi(x, y)$ is free. More specifically, we can inductively define free and bound variables in this way:

1. **Atomic Formula**: There are no bound variable in any atomic formula ϕ and for each variable x in ϕ, x is free.

2. **Negation**: If $\neg\phi$ is a formula and x a variable, then x is bound/free in $\neg\phi$ if and only if x is bound/free in ϕ.

3. **Binary Connectives**: If $\phi \vee \psi$ is a formula and x a variable, then x is bound/free in $\phi \vee \psi$ if and only if x is bound/free in either ϕ and ψ. The same definition holds for all other binary operators.

4. **Quantifiers**: If $\forall y.\phi$ is a formula and x a variable, then x is free in $\forall y.\phi$ if and only if x is free in ϕ and different from y. On the other hand, x is bound in $\forall y.\phi$ if and only if x is y or x is bound in ϕ. The same definition holds for the existential operator.

For example in $\forall x.\forall y.(T(x) \wedge (Q(y) \vee (P(z))))$, the variables x and y are bound while z is not.

3.1.2 Semantics of First Order Logic

The use of term semantics usually refers to the study of meanings. The syntax introduces the symbols but without the semantics that gives a meaning to that symbols they are actually futile.

The interpretation I of first-order logic determines the domain (indicated with D) over quantifiers range and gives a denotation to all non logical constants. Interpretation assigns to each constant c an element of D, to each n-ary function f an operation of arity n among elements of D and for each n-ary predicate P a relation of arity n on D.

In this way each formula evaluates in true or false according to a given interpretation I and to an assignment μ to variables. To better understand why an assignment μ is needed, notice that the only way to evaluate the truth value of a formula F, is to assign an element of D to each free variable x of F. The truth value of F changes depending on the values given to its free variables.

The following rules are used to make assignments:

- **Variables**: Each variable x evaluates to $\mu(x)$.

- **Functions**: Given terms $t_1,t_2,\ldots,t_n$ that have been evaluated to elements $d_1,d_2,\ldots,d_n$ of the domain D, and a n-ary function symbol f, the term $f(t_1,t_2,\ldots,t_n)$ evaluates to $(I(f))(d_1,d_2,\ldots,d_n)$.

Next each formula, is assigned a truth value:

- **Atomic Formulas 1**: A formula $P(t_1,t_2,\ldots,t_n)$ is associated the value true or false depending on whether $<v_1,v_2,\ldots,v_n> \in I(P)$, where $<v_1,v_2,\ldots,v_n>$ are the evaluation of terms $t_1,t_2,\ldots,t_n$ and $I(P)$ is the evaluation of P.

- **Atomic Formulas 2**: A formula $t_1=t_2$ is assigned true if t_1 and t_2 evaluate to the same object.

- **Logical Connectives**: A formula in the form $\neg\phi$, $\phi \wedge \varphi$, $\phi \vee \varphi$, $\phi \rightarrow \varphi$ and $\phi \leftrightarrow \varphi$ is evaluated according to the truth table for the connective in question.
- **Existential Quantifiers**: A formula φ in the form $\exists x.\phi(x)$ is true according to I and μ if there exists an evaluation μ' of the variables that only differs from μ regarding the evaluation of x and such that φ is true according to the interpretation I and the variable assignement μ'.
- **Universal Quantifiers**: A formula φ in the form $\forall x.\phi(x)$ is true according to I and μ if $\varphi(x)$ is true for every pair composed by the interpretation I and some variable assignement μ' that differs from μ only on the value of x.

Finally we introduce the concepts of *validity, satisfiability* and *logical consequence.*

- **Satisfiable**: A formula F is satisfiable if there exists some interpretations under which F is true.
- **Valid**: A formula F is valid if F is true in every interpretations.
- **Logical Consequence**: A formula ϕ is a logical consequence of the formula φ if every interpretations that make φ true, make ϕ also true

3.2 First-Order μ-calculus

The *μ-calculus* is a formal system which can express some important behaviors of programming languages. Propositional logic and first-order logic are inadequate in expressing some features of programming languages due to their weak expressive power.

Over the years studies have been conducted exploring how to make logics more powerful in order to deal with more advanced problems in computer science and mathematics. One good solution is to increase the power of basic logics (propositional logic, first-order logic, modal logic) with general inductive and co-inductive definitions. The logics obtained with such inclusion will be respectively called propositional *μ-calculus*, first-order *μ-calculus* and modal *μ-calculus.*

Thanks to the fixed point operator μ many behaviors of programs can be formalized in a nice and elegant way. However, such increase in expressive power is not totally effortless. Fixed point logics are traditionally considered hard to understand. Furthermore, their semantics requires familiarity with material that, although not difficult, is often omitted from undergraduate studies. Not least, the logics become substantially more expressive but decision problems become harder and in some cases impossible (for instance from first order logic to first order logic μ-calculus). Moreover, proof systems become more complex.

Next sections present first an introduction to fixed point theory and then the syntax and semantics of first-order logic *μ-calculus* is introduced.

Fixed Points

The description in this section is based on Stirling's book *Modal and Temporal Properties of Processes* [Sti01].

Let P be a generic set of entities that are reachable starting from an initial state and following an arbitrary length sequence of transitions. Then, to give a semantic to our state based logic we could take an arbitrary formula ϕ, and map ϕ to the powerset of P, 2^P (we write $\wp(P)$). The purpose of such representation is to know in which states a formula holds, or differently speaking, in which states a formula is valid. A formula ϕ may also contain free variables Z ranging over $\wp(P)$. The semantic of a formula $\phi(Z)$ can be viewed as a function $f{:}\wp(P) \rightarrow \wp(P)$.

If $f(S)=S$, then S is called fixed point. The application of the function f on the set S such that $S \subseteq P$, has left S unchanged. We are again at the point we started. This illustrates the similarity of fixed points with recursive behaviors. Furthermore if $f(S) \subseteq f(S')$ whenever $S \subseteq S' \subseteq P$ then f is *monotonic*.

For monotone functions on complete lattices, there is *Knaster* and *Tarski* theorem [Win93], that ensures the existence of both least and greatest fixed point.

Theorem 3.2.1 (Knaster-Tarski) If $(L, \leq)$ is a complete lattice and
$f{:}L \to L$ is ordering-preserving (monotone function), then f has a fixed point. In fact, the set of fixed points of f is a complete lattice.

The important result of theorem 3.2.1 is the guarantee of existence of least and greatest fixed points for f, since every complete lattice always has a least upper bound and a greatest lower bound. Greatest and least fixed points are defined as follows:

- *Greatest* : Union of *post fixed point* : $\bigcup\{S \subset P \mid S \subseteq f(S)\}$
- *Least* : Intersection of *pre fixed point* : $\bigcap\{S \subset P \mid f(S) \subseteq S\}$

Hence, it is possible to introduce two more operators to represent least and greatest fixed points: μ, so that $\mu Z.\phi(Z)$ is a formula whose semantics is least fixed point of f, and similarly ν, so that $\nu Z.\phi(Z)$ is a formula whose semantics is greatest fixed point of f (when the semantics $\phi(Z)$ is monotonic).

Approximation

There is a more mechanical method, due to Tarski and others, for finding the greatest and the least fixed point of monotonic functions. Let f be a monotonic function, let P be a set of processes, and let S be a subset of P, then from Knaster and Tarski theorem, there always exists greatest and least fixed points.

Supposing now that our goal is to determine the greatest fixed point νf, we might define iteratively $\nu^i f$ as follows:

$$\nu^0 f = P \text{ and } \nu^{i+1} f = f(\nu^i f)$$

Due to the monotonicity of f we have that $\nu^1 f \subseteq \nu^0 f$ and $f(\nu^1 f) \subseteq f(\nu^0 f)$ that is $\nu^2 f \subseteq \nu^1 f$. The iterative application of f will then imply that for all i, $\nu^{i+1} f \subseteq \nu^i f$. Notice now that the greatest fixed point νf is subset of $\nu^0 f$, since $\nu^0 f$ is the set of all processes. Due to the monotonicity of f, again, for all i, $\nu f \subseteq \nu^i f$. We have built a chain of inclusions such that:

$$\begin{array}{ccccccccc} \nu^0 f & \supseteq & \nu^1 f & \supseteq & \ldots & \supseteq & \nu^i f & \supseteq & \ldots \\ \cup & & \cup & & & & \cup & & \\ \nu f & & \nu f & & \ldots & & \nu f & & \ldots \end{array}$$

Thus, when $\nu^{i+1} f = \nu^i f$ we have found the greatest fixed point νf. The situation for the least fixed point is exactly the dual of the latter. More specifically, let μf be the least fixed point, $\mu^0 = \emptyset$ (empty set) and let $\mu^{\alpha+1} f = f(\mu^\alpha f)$, then:

$$\begin{array}{ccccccccc} \mu f & & \mu f & & \ldots & & \mu f & & \ldots \\ \cup & & \cup & & & & \cup & & \\ \mu^0 f & \subseteq & \mu^1 f & \subseteq & \ldots & \subseteq & \mu^\alpha f & \subseteq & \ldots \end{array}$$

This time the least fixed point μf will be a superset of each of the iterates $\mu^{\alpha} f$ and it will appear at the first time when $\mu^{\alpha} f = \mu^{\alpha+1} f$. Each $\nu^{\alpha} f$ approximates νf from above, whereas each $\mu^{\alpha} f$ approximates μf from below.

If P is a finite set containing n processes, then the iterative process terminates in at most n steps.

If P is an infinite set, it is still possible to guarantee that the fixed point νf is reachable iteratively by invoking ordinals as indices. In 1883, Cantor introduced the concept of ordinals [Can82]. Recall that ordinals are ordered as follows.

$$0, 1, \ldots, w, w+1, \ldots, w+w, w+w+1, \ldots$$

The ordinal w is the initial limit ordinal (which has no immediate predecessor), whereas $w+1$ is its successors. Let α and λ be two ordinals. Let $\nu^0 f$ be the set P ($\nu^0 f$=P) and $\nu^{\alpha+1} f = f(\nu^{\alpha} f)$. The case when λ is a limit ordinal is defined as follows.

$$\nu^{\lambda} f = \bigcap \{\nu^{\alpha} f : \alpha < \lambda\}$$

Due to the monotonicity of f, like previous cases:

$$\begin{array}{ccccccccccc} \nu^0 f & \supseteq & \nu^1 f & \supseteq & \ldots & \supseteq & \nu^w f & \supseteq & \nu^{w+1} f & \supseteq & \ldots \\ \cup & & \cup & & & & \cup & & \cup & & \\ \nu f & & \nu f & & \ldots & & \nu f & & \nu f & \supseteq & \ldots \end{array}$$

The fixed point νf appears somewhere in the sequence, at the first point when $\nu^{\alpha} f = \nu^{\alpha+1} f$.

The situation for the least fixed point μf is dual. Let $\mu^0 f$ be the empty set $\emptyset$, and $\mu^{\alpha+1} f = f(\mu^{\alpha} f)$. The limit ordinal λ is defined as follows:

$$\mu^{\lambda} f = \bigcup \{\mu^{\alpha} f : \alpha < \lambda\}$$

There is the following possibly increasing sequence of sets.

$$\begin{array}{ccccccccccc} \mu f & & \mu f & & \ldots & & \mu f & & \mu f & & \ldots \\ \cup & & \cup & & & & \cup & & \cup & & \\ \mu^0 f & \subseteq & \mu^1 f & \subseteq & \ldots & \subseteq & \mu^w f & \subseteq & \mu^{w+1} f & \subseteq & \ldots \end{array}$$

The fixed point μf is a superset of each iterates $\mu^{\alpha} f$. First, $\mu^0 f \subseteq \mu f$, and by monotonicity of f $\mu^{\alpha} f \subseteq \mu f$ for any α. As for the latter case, the first point in which $\mu^{\alpha} f = \mu^{\alpha+1} f$ determinates the fixed point μf.

3.2.1 Syntax of First-Order μ-calculus

The presentation in this section is based on Dam and Sprenger paper [SD03].

Let $x,y,z... \in V_i$ be an infinite set of individual variables, let $X,Y,Z... \in V_p$ be predicate variables with arity $n \geq 0$ and let $\iota,\kappa,\lambda... \in V_o$ be ordinal variables. Let t ranges over the terms of some signature Σ. We write $\overline{t}$ for a vector $t1,...,tn$ of terms. According to the above notation the syntax of the first-order *μ-calculus* is expressed as follows:

Definition 3.2.2 (Syntax) The syntax of *μ-calculus* formulas ϕ and predicates Φ over Σ is inductively defined by:

$$\phi ::= \mathrm{t}{=}\mathrm{t}' \mid \kappa{<}\kappa' \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists \mathrm{x}.\phi \mid \exists \kappa.\phi \mid \Phi(\overline{t})$$

$$\Phi ::= \mathrm{X} \mid \mu \mathrm{X}(\overline{x}).\phi \mid \mu^{\kappa} \mathrm{X}(\overline{x}).\phi$$

$$
\begin{array}{lcl}
fv(\exists\kappa\text{'} < \kappa.\phi) & = & (fv(\phi)\text{-}\{\kappa\text{'}\})\cup\{\kappa\} \\
fv(\Phi(\bar{t})) & = & fv(\Phi)\cup fv(\bar{t}) \\
fv(\mu X(\overline{x}).\phi) & = & fv(\phi)\text{-}\{\mathrm{X},\overline{x}\} \\
fv(\mu^{\kappa}\mathrm{X}(\overline{x}).\phi) & = & (fv(\phi)\text{-}\{\mathrm{X},\overline{x}\})\cup\{\kappa\}
\end{array}
$$

Table 3.1: Free variables of first-order *μ-calculus* formulas

with the restriction that the arities of Φ and $\bar{t}$ are the same in $\Phi(\bar{t})$ and both $\mu\mathrm{X}(\overline{x}).\phi$ and $\mu^{\mathrm{k}}\mathrm{X}(\overline{x}).\phi$ are subject to the conditions that (i) the arities of X and $\overline{x}$ are the same, and (ii) all occurrences of X in ϕ appear under an even number of negations (formal monotonicity).

The sets of free variables of formulas are defined as expected. In particular:

This can be extended to any set of formulas Δ by defining $\mathrm{fv}(\Delta) = \bigcup\{\mathrm{fv}(\phi) \mid \phi \in \Delta\}$. In Table 3.2, the derivation of greatest fixed point and greatest fixed point approximation is highlighted:

$$
\begin{array}{lcl}
\nu\mathrm{X}(\overline{x}).\phi & = & \neg\mu\ \mathrm{X}(\overline{x}).\neg\phi[\neg\ \mathrm{X}/\mathrm{X}] \\
\nu^{\kappa}\mathrm{X}(\overline{x}).\phi & = & \neg\mu^{\kappa}\mathrm{X}(\overline{x}).\neg\phi[\neg\ \mathrm{X}/\mathrm{X}]
\end{array}
$$

Table 3.2: Greatest fixed point operators

3.2.2 Semantics of First-Order μ-calculus

Let Σ be a first-order signature. A Σ-model $M = (A, \rho)$ consists in a Σ-structure A and an A-environment ρ, that maps each variable to its respectively domain. Let $\mathbf{2}$=$\{0,1\}$ be the two point lattice and let $\mathbf{Pred}(S) = \mathbf{2}^S$ be the lattice of predicates over S ordered pointwise. The semantics interprets a *μ-calculus* formula ϕ to an element $||\phi||_M \in \{true, false\}$ and a m-ary abstraction Φ as an element $||\Phi_M|| \in \mathbf{Pred}(|A|^m)$.

The semantics $||t_\rho|| \in |A|$ of a term t is defined as usual. Thus, given a signature Σ and a Σ-model (A, ρ) the semantics of *μ-calculus* formulas ϕ and abstractions Φ over Σ is inductively defined as:

$$
\begin{array}{lcl}
||t = t'||_\rho & = & \text{if } ||t||_\rho = ||t'||_\rho \text{ then 1 else 0} \\
||\neg\phi||_\rho & = & 1 - ||\phi||_\rho \\
||\phi_1 \vee \phi_2|| & = & \max\{||\phi_1||_\rho, ||\phi_2||_\rho\} \\
||\exists x.\phi||_\rho & = & \bigvee_{a\in|A|} ||\phi||_{\rho[a/x]} \\
|||\exists k.\phi|_\rho & = & \bigvee_{\beta} ||\phi||_{\rho[\beta/k]} \\
||\exists k' < k.\phi||_\rho & = & \bigvee_{\beta<\rho(k)} ||\phi||_{\rho[\beta/k']} \\
||\Phi(\bar{t})||_\rho & = & ||\Phi||_\rho(||\bar{t}||_\rho) \\
||X||_\rho & = & \rho(X) \\
||\mu X(\overline{x}).\phi||_\rho & = & \mu\Psi \\
||\mu^k X(\overline{x}).\phi||_\rho & = & \mu^{\rho(k)}\Psi
\end{array}
$$

Table 3.3: Semantic of first-order *μ-calculus*

where $\Psi = \lambda P.\lambda\overline{a}.||\phi||_{\rho[P/X,\overline{a}/\overline{x}]}$.

A model $M = (A, \rho)$ *satisfies* a formula ϕ, and we write $M \models \phi$, if $||\phi||_\rho$=1. The formula ϕ is called *valid*, and we write $\models \phi$, if ϕ is satisfied in all Σ-models.

3.3 Gentzen-Type System for first order μ-calculus

This section introduces a Gentzen-Type System for proving properties with respect to first-order *μ-calculus* formulas. The contents are strongly based on papers of Dam, Gurov and Fredlund [Dam98, DG00, Fre01]. The crucial idea is to use explicit approximations of fixed points. The system is based on a global condition on derivation, which uses ordinal approximation to apply a well-founded induction on ordinals.

3.3.1 Ordinal Constraints

The proof system uses explicit ordinal approximation of fixed point formulas.Let $O = (|O|, <_O)$ be a partial order, where $|O|$ is a finite set of ordinal variables and $<_O$ is a binary, irreflexive and transitive relation on $|O|$. We call O set of ordinal constraints.

3.3.2 Sequent

The sequent of the proof system has the following form : $\Gamma \vdash_O \Delta$. Both Γ, Δ are sets of formulas and O is the set of ordinal constraints. The sequent $\Gamma \vdash_O \Delta$ is *well-formed* if all the free ordinal variables in Γ and Δ are in $|O|$.

The set of free variables of a sequent $\Gamma \vdash_O \Delta$ is defined as follows:

$$fv(\Gamma \vdash_O \Delta) = fv(\Gamma \cup \Delta) \cup |O|.$$

Given a Σ-model $M = (A, \rho)$ we say that M *satisfies* a sequent $\Gamma \vdash_O \Delta$ whenever ρ respects O and $M \models \phi$ for all $\phi \in \Gamma$ implies that $M \models \psi$ for some $\psi \in \Delta$. If the model M does not satisfy the sequent $\Gamma \vdash_O \Delta$, then M *falsifies* $\Gamma \vdash_O \Delta$.

A sequent $\Gamma \vdash_O \Delta$ is *valid*, if it is satisfied in all models and *invalid* otherwise.

3.3.3 Proof Rules

The proof rules are presented according to the tableaux style with a conclusion above the line and the premises below. Informally, the conclusion is valid, if all the premises are valid.

$$(\text{Name})\frac{\Gamma \vdash_O \Delta}{\Gamma_1 \vdash_{O_1} \Delta_1 \ldots \Gamma_n \vdash_{O_n} \Delta_n}$$

For sake of simplicity rules of derived operators are not presented.
In Table 3.4 and 3.5 the standard *Structural* and *Logical/Equality* rules are introduced.

Structural Rules

$(Id)\ \dfrac{\Gamma, \phi \vdash_O \phi, \Delta}{\cdot}$ $\qquad$ $(Cut)\ \dfrac{\Gamma \vdash_O \Delta}{\Gamma, \phi \vdash_O \Delta \quad \Gamma \vdash_O \phi, \Delta}$

$(Weak\text{–}L)\ \dfrac{\Gamma, \phi \vdash_O \Delta}{\Gamma \vdash_O \Delta}$ $\qquad$ $(Weak\text{–}R)\ \dfrac{\Gamma \vdash_O \Delta}{\Gamma, \phi \vdash_O \Delta}$

Table 3.4: Structural Rules

Logical and Equality Rules

$$(\neg\text{–}L)\ \frac{\Gamma, \neg\phi \vdash_O \Delta}{\Gamma \vdash_O \phi, \Delta} \qquad (\neg\text{–}R)\ \frac{\Gamma \vdash_O \neg\phi, \Delta}{\Gamma, \phi \vdash_O, \Delta}$$

$$(\vee\text{–}L)\ \frac{\Gamma, \phi_1 \vee \phi_2 \vdash_O \Delta}{\Gamma, \phi_1 \vdash_O, \Delta \quad \Gamma, \phi_1 \vdash_O, \Delta} \qquad (\vee\text{–}R)\ \frac{\Gamma \vdash_O \phi_1 \vee \phi_2, \Delta}{\Gamma \vdash_O \phi_1\phi_2, \Delta}$$

$$(\exists_I\text{–}L)\ \frac{\Gamma, \exists x.\phi \vdash_O \Delta}{\Gamma, \phi \vdash_O \Delta}\ x \notin fv(\Gamma \cup \Delta) \qquad (\exists_I\text{–}R)\ \frac{\Gamma \vdash_O \exists x.\phi, \Delta}{\Gamma \vdash_O \phi[t/x], \Delta}$$

$$(=\text{–}L)\ \frac{\Gamma[t_2/x], t1 = t2 \vdash_O \Delta[t_2/x]}{\Gamma[t_1/x] \vdash_O \Delta[t_1/x]} \qquad (=\text{–}R)\ \frac{\Gamma \vdash_O t = t, \Delta}{\cdot}$$

Table 3.5: First Order Rules

Fixed Point Rules

$$(\mu_1\text{–}L)\ \frac{\Gamma, (\mu X(\overline{x}).\phi)(\bar{t}) \vdash_O \Delta}{\Gamma, \exists k.(\mu^k X(\overline{x}).\phi)(\bar{t}) \vdash_O \Delta} \qquad (\mu_0\text{–}R)\ \frac{\Gamma \vdash_O (\mu X(\overline{x}).\phi)(\bar{t})}{\Gamma \vdash_O \phi[\mu X(\overline{x}).\phi/X, \bar{t}/\overline{x}], \Delta}$$

$$(\mu_k\text{–}L)\ \frac{\Gamma, (\mu^k X(\overline{x}).\phi)(\bar{t}) \vdash_O \Delta}{\Gamma, \exists k' < k.\phi[\mu^{k'} X(\overline{x}).\phi/X, \bar{t}/\overline{x}] \vdash_O \Delta}$$

$$(\mu_k\text{–}R)\ \frac{\Gamma \vdash_O (\mu^k X(\overline{x}).\phi)(\bar{t}), \Delta}{\Gamma \vdash_O \exists k' < k.\phi[\mu^{k'} X(\overline{x}).\phi/X, \bar{t}/\overline{x}], \Delta}$$

Ordinal Rules

$$(\exists_O\text{–}L)\frac{\Gamma, \exists k.\phi \vdash_O \Delta}{\Gamma, \phi \vdash_{O,k} \Delta}\ k \notin |O| \qquad (\exists_O\text{–}R)\frac{\Gamma \vdash_O \exists k.\phi, \Delta}{\Gamma \vdash_O \phi[\iota/k]\Delta}\ \iota \in |O|$$

$$(\exists_O^<\text{–}L)\frac{\Gamma, \exists k' < k.\phi \vdash_O \Delta}{\Gamma, \phi \vdash_{O,k'<k} \Delta}\ k' \notin |O| \qquad (\exists_O^<\text{–}R)\frac{\Gamma \vdash_O \exists k' < k.\phi, \Delta}{\Gamma \vdash_O \phi[\iota/k']\Delta}\ \iota <_O k$$

$$(OrdSrt)\ \frac{\Gamma \vdash_O \Delta}{\Gamma \vdash_{O'} \Delta}\ O' \subseteq O \text{ and } \iota <_{O'} k,\ k \in |O| \Rightarrow \iota \in |O|$$

Table 3.6: Fixed Points and Ordinal Rules

In Table 3.6 there are rules for *fixed points* and *ordinals*. Rule (μ_1–L) is the crucial rule for fixed points because it introduces a new ordinal approximation k in the sequent. If there is a least approximated fixed point, $\mu^{\kappa}\mathrm{X}(\overline{x}).\phi$, on the left or right side of the turnstile, with rules (μ_k–L) and (μ_k–R), it is possible to unfold the definition of $\mu^{\kappa}\mathrm{X}(\overline{x}).\phi$ (left or right side). It will be possible to introduce a new ordinal $k' \notin O$ that is strictly less than k to the left hand side or to use an ordinal $k' \in O$ that is strictly less than k to the right hand side.

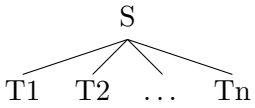
3.3.4 Derivation Tree

Formally a *derivation tree* is a three-tuple (N, E, L) composed by a tree (N, E) with nodes N and edges E, and a labeling function L which associates each node of the tree with a sequent. The labeling function L labels nodes and sequents consistently with respect to the proof rules.

As example consider a proof rule:

$$\frac{S}{S1 \quad S2 \ \ldots \ Sn}$$

and the derivation trees *T1,T2, . . . ,Tn* having the roots *S1,S2, . . . , Sn*, then



is a derivation tree with root S and leaves *T1, T2,. . . , Tn*.

3.3.5 The Proof

Fixed point formulas have no bound on the number of possible unfoldings. Generally fixed point formulas can grow infinitely so often it is not possible to decide validity of properties using proof rules introduced in Section 3.3.3. We need a process that allows to consider only a finite part of an infinite derivation tree as proof for properties. To reach that goal, the well-founded induction on ordinal presented by Dam and Sprenger in [SD03] is used. We need to find a special derivation tree representing an infinite derivation chain. Intuitively, in that derivation tree, the root and the leaf have to be a *repeat* of the same node. In order to understand the concept, it must be precisely defined what a repeat is. Let $D = (E, N, L)$ be a derivation tree (see Section 3.3.4) then:

Definition 3.3.1 (Repeat) Let $M = (\Gamma \vdash_O \Delta)$ and $N = (\Gamma' \vdash_{O'} \Delta')$ be two nodes of D. A repeat of D is a three-tuple (M, N, σ), where N is a leaf node of D and σ a substitution function such that:

1. $\phi \in \Gamma$ implies $\sigma(\phi) \in \Gamma'$.
2. $\psi \in \Delta$ implies $\sigma(\psi) \in \Delta'$.
3. $k \in |O|$ implies $\sigma(k) \in |O'|$.
4. $\iota <_O k$ implies $\sigma(\iota) <_{O'} \sigma(k)$

The node N is called repeat node while M is called companion node.

Definition 3.3.2 (Pre-Proof) A pre-proof $P = (D, R)$ is a derivation tree D with a set R of repeats such that R contains exactly one repeat (M, N, σ) for each leaf node N of D.

Definition 3.3.3 (Pre-Proof Graph) Let $P = (D, R)$ be a pre-proof. The graph obtained from D by adding $\{(N, M)|(M, N, \sigma) \in R\}$ is called pre-proof graph for P.

Definition 3.3.4 (Discharge Order) A discharge order on a pre-proof $P = (D, R)$ is a partial ordering $\prec$ on the repeats such that, if $R = (M, N, \sigma)$ and $R' = (M', N', \sigma')$ are any two repeats for which $R \not\preceq R'$ and $R' \not\preceq R$, then every path $M \ldots N'$,$M' \ldots N$ through the pre-proof graph contains a repeat R'' with $R \prec R''$ and $R' \prec R''$.

Definition 3.3.5 (Progress, Preservation) Let $R = (M, N, \sigma)$ be a repeat and M, N be respectively labelled with sequents $\Gamma_M \vdash_P \Delta_M$ and $\Gamma_N \vdash_O \Delta_N$. Let $\alpha \in |P|$ be an ordinal variable. We say that R progresses on α if $\sigma(\alpha) <_O \alpha$, and R preserves α if $\sigma(\alpha) \leq_O \alpha$.

Definition 3.3.6 (Discharge) A pre-proof $P = (D, R)$ is dischargeable with respect to a discharge order $\prec$, if for every repeat R there is an ordinal variable α_R such that R progresses on α_R and R preserves $\alpha_{R'}$ whenever $R \prec R'$.

Definition 3.3.7 (Proof) A pre-proof $P = (D, R)$ is a proof of the root sequent of D if it is dischargeable with respect to some discharge order $\prec$.

We now cite the local soundness lemma and the soundness theorem without proving them. Further information can be found in [SD03].

Lemma 3.3.8 (Local Soundness) For each proof rule:

$$\frac{\Gamma \vdash_O \Delta}{\Gamma_1 \vdash_{O_1} \Delta_1 \ldots \Gamma_n \vdash_{O_n} \Delta_n}$$

and any falsifying interpretation E of its conclusion sequent $\Gamma \vdash_O \Delta$, there exists a falsifying interpretation E' of one of its premises $\Gamma_i \vdash_{O_i} \Delta_i$ such that E and E' agree on all common free variables of $\Gamma \vdash_O \Delta$ and $\Gamma_i \vdash_{O_i} \Delta_i$.

Theorem 3.3.9 Let $P = (D, R)$ be a proof of $\Gamma \vdash_O \Delta$, then $\Gamma \vdash_O \Delta$ is valid.

3.4 A concrete Example

The section shows a concrete example of all the concepts introduced in the previous sections.

More specifically, it is proved that a natural number x, is either even or odd.

Before presenting the complete proof, some derived rules and abbreviations must be introduced. The derivation uses the rules $(\wedge\text{–}L)$, $(\wedge\text{–}R)$ for conjunctions left and right hand side. More specifically:

$$(\wedge\text{–}L)\frac{\Gamma, \phi_1 \wedge \phi_2 \vdash_O \Delta}{\Gamma, \phi_1, \phi_2 \vdash_O \Delta} \quad (\wedge\text{–}R)\frac{\Gamma \vdash_O \phi_1 \wedge \phi_2, \Delta}{\Gamma \vdash_O \phi_1 \Delta \quad \Gamma \vdash_O \phi_2 \Delta}$$

Table 3.7: Some derived rules

The partial orders O_1, O_2, O_3 used in the derivation are the smallest partial orders such that:
Three least fixed point predicates are defined as follows:

$$Nat : \mu Nat(x).(x = 0 \vee \exists y(Nat(y) \wedge x = succ(y)))$$
$$Even : \mu Even(x).(x = 0 \vee \exists y(Nat(y) \wedge Even(y) \wedge x = succ(succ(y))))$$
$$Odd : \mu Odd(x).(x = succ(0) \vee \exists y(Nat(y) \wedge Odd(y) \wedge x = succ(succ(y))))$$

$$
\begin{array}{lll}
|O_1| = \{\alpha\} & & \\
|O_2| = \{\alpha, \alpha^{'}\} & & \alpha^{'} <_{O_2} \alpha \\
|O_3| = \{\alpha, \alpha^{'}, \alpha^{''}, t, p\} & \alpha^{''} <_{O_3} \alpha^{'} & \alpha^{'} <_{O_3} \alpha
\end{array}
$$

Table 3.8: Partial order of the variable used in the example

The predicates *Nat, Even, Odd* are intended to represent respectively the straightforward concepts of natural number, even natural number and odd natural number.

Furthermore the following abbreviations will be used in the rest of the example:

$$
\begin{array}{l}
Nat^{\alpha} : \mu^{\alpha} Nat(x).(x = 0 \vee \exists y(Nat(y) \wedge x = succ(y))) \\
Even^{\alpha} : \mu^{\alpha} Even(x).(x = 0 \vee \exists y(Nat(y) \wedge Even(y) \wedge x = succ(succ(y)))) \\
Odd^{\alpha} : \mu^{\alpha} Odd(x).(x = 0 \vee \exists y(Nat(y) \wedge Odd(y) \wedge x = succ(succ(y))))
\end{array}
$$

The property to prove is the following:

$$Nat(z) \vdash Even(z), Odd(z)$$

The derivation starts in this way:

$$
\dfrac{\dfrac{\dfrac{\dfrac{(\mu Nat(x).(x = 0 \vee \exists y(Nat(y) \wedge x = succ(y))))(z) \vdash_{\emptyset} Even(z), Odd(z)}{(\exists k.\mu^{k} Nat(x).(x = 0 \vee \exists y(Nat(y) \wedge x = succ(y))))(z) \vdash_{\emptyset} Even(z), Odd(z)}\ (\mu_1\text{-}L)}{(\mu^{\alpha} Nat(x).(x = 0 \vee \exists y(Nat(y) \wedge x = succ(y))))(z) \vdash_{O_1} Even(z), Odd(z)}\ (\exists_O\text{-}L)}{Nat^{\alpha}(z) \vdash_{O_1} Even(z), Odd(z) \quad \blacksquare}\ (\mu^{k}\text{-}L)}{\dfrac{(z = 0 \vee \exists y(Nat^{\alpha^{'}}(y) \wedge z = succ(y)))(z) \vdash_{O_2} Even(z), Odd(z)}{(A) \qquad (B)}\ (\vee\text{-}L)}
$$

The two subgoals **(A)** and **(B)** generated by the last rule are the following:

- **(A)**: $z = 0 \vdash_{O_2} Even(z), Odd(z)$
- **(B)**: $(\exists y(Nat^{\alpha^{'}}(y) \wedge z = succ(y))) \vdash_{O_2} Even(z), Odd(z)$

We focus for the moment on goal **(A)**.

$$
\dfrac{\dfrac{\dfrac{\dfrac{z = 0 \vdash_{O_2} Even(z), Odd(z)}{\vdash_{O_2} Even(0), Odd(0)}\ (=\text{-}L)}{\vdash_{O_2} (\mu Even(x).(x = 0 \vee \exists y(Nat(y) \wedge Even(y) \wedge x = succ(succ(y)))))(0), Odd(0)}}{\vdash_{O_2} ((0 = 0 \vee \exists y(Nat(y) \wedge Even(y) \wedge 0 = succ(succ(y)))), Odd(0)}\ (\mu\text{-}R)}\ (\vee\text{-}R)}{\vdash_{O_2} ((0 = 0, \exists y(Nat(y) \wedge Even(y) \wedge 0 = succ(succ(y)))), Odd(0)}\ (=\text{-}R)}{Qed}
$$

Now that the goal **(A)** is solved, we can prove the validity of goal **(B)**.

$$
\dfrac{\dfrac{\dfrac{\dfrac{(\exists y(Nat^{\alpha^{'}}(y) \wedge z = succ(y))) \vdash_{O_2} Even(z), Odd(z)}{Nat^{\alpha^{'}}(t) \wedge z = succ(t) \vdash_{O_2} Even(z), Odd(z)}\ (\exists_I\text{-}L)}{Nat^{\alpha^{'}}(t), z = succ(t) \vdash_{O_2} Even(z), Odd(z)}\ (\wedge\text{-}L)}{(\mu^{\alpha^{'}} Nat(x).(x = 0 \vee \exists y(Nat(y) \wedge x = succ(y))))(t), z = succ(t)) \vdash_{O_2} Even(z), Odd(z)}\ (\mu^{k}\text{-}L)}{\dfrac{(t = 0 \vee \exists y(Nat^{\alpha^{''}}(y) \wedge t = succ(y))), z = succ(t) \vdash_{O_3} Even(z), Odd(z)}{(B1) \qquad (B2)}\ (\vee\text{-}L)}
$$

The two subgoals **(B1)** and **(B2)**, obtained from the last rule are:

- **(B1)**: $t = 0, z = succ(t) \vdash_{O_3} Even(z), Odd(z)$

- **(B2)**: $\exists y(Nat^{\alpha}{}''(y) \wedge t = succ(y)), z = succ(t)) \vdash_{O_3} Even(z), Odd(z)$

We proceed solving first the goal **(B1)**:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{t = 0, z = succ(t) \vdash_{O_3} Even(z), Odd(z)}{\vdash_{O_3} Even(succ(0)), Odd(succ(0))}\,(= -L)}{\vdash_{O_3} Even(succ(0)), (\mu Odd(x).(x = succ(0) \vee \exists y(Nat(y) \wedge Odd(y) \wedge x = succ(succ(y)))))(succ(0))}}{\vdash_{O_3} Even(succ(0)), (succ(0) = succ(0) \vee \exists y(Nat(y) \wedge Odd(y) \wedge succ(0) = succ(succ(y))))}\,(\mu{-}R)}{\vdash_{O_3} Even(succ(0)), succ(0) = succ(0), \exists y(Nat(y) \wedge Odd(y) \wedge succ(0) = succ(succ(y)))}\,(\vee{-}R)}{Qed}\,(= -R)$$

Finally we solve goal **(B2)**.

$$\dfrac{\dfrac{\dfrac{\exists y(Nat^{\alpha}{}''(y) \wedge t = succ(y)), z = succ(t) \vdash_{O_3} Even(z), Odd(z)}{(Nat^{\alpha}{}''(p) \wedge t = succ(p)), z = succ(t) \vdash_{O_3} Even(z), Odd(z)}\,(\exists_I{-}L)}{Nat^{\alpha}{}''(p), t = succ(p), z = succ(t) \vdash_{O_3} Even(z), Odd(z)}\,(\wedge{-}L)}{Nat^{\alpha}{}''(p) \vdash_{O_3} Even(succ(succ(p))), Odd(succ(succ(p)))}\,(= -L)$$

In order to be more concise we consider the application of rule $(\mu{-}R)$ for both formulas on the right hand side, *Even(succ(succ(p)))* and *Odd(succ(succ(p)))*. After such application the following goal is obtained:

$$Nat^{\alpha}{}''(p) \quad \vdash_{O_3} \quad \begin{array}{l} succ(succ(p)) = 0, \exists y(Nat(y) \wedge Even(y) \wedge succ(succ(p)) = succ(succ(y))), \\ succ(succ(p)) = succ(0), \exists y(Nat(y) \wedge Odd(y) \wedge succ(succ(p)) = succ(succ(y))) \end{array}$$

For sake of simplicity the first and third formula on the right hand side are weaken out (they are trivially false), and it is considered only the remainder.

$$Nat^{\alpha}{}''(p) \quad \vdash_{O_3} \quad \begin{array}{l} \exists y(Nat(y) \wedge Even(y) \wedge succ(succ(p)) = succ(succ(y))), \\ \exists y(Nat(y) \wedge Odd(y) \wedge succ(succ(p)) = succ(succ(y))) \end{array}$$

Applying twice rule $(\exists_I{-}R)$ and choosing the variable p as new variable to introduce, the sequent becomes:

$$Nat^{\alpha}{}''(p) \quad \vdash_{O_3} \quad \begin{array}{l} Nat(p) \wedge Even(p) \wedge succ(succ(p)) = succ(succ(p)), \\ Nat(p) \wedge Odd(p) \wedge succ(succ(p)) = succ(succ(p)) \end{array}$$

Notice now that applying more times rule $(\wedge{-}R)$ we obtain 9 subgoals in which:

- If there is the formula $Nat(p)$ right hand side, the sequent is true due to the rule (id).
- If there is the formula $succ(succ(p)) = succ(succ(p))$ right hand side, the sequent is true due to rule $(= -R)$.

It remains to analyze the following subgoal:

$$Nat^{\alpha}{}''(p) \vdash_{O_3} Even(p), Odd(p)$$

The node above, can be discharged due to the well-founded discharge condition against the node labelled with ■, concluding the proof.

Chapter 4

The Tool: VCPT

The term VCPT is an acronym for *VeriCode Proof Tool*.

Concisely VCPT is a proof assistant for *μ-calculus* based on first-order logic with explicit fixed points.

VCPT is implemented using *Standard ML* (SML), a programming language derived from *ML*. ML was originally a set of meta-languages created by *Robin Miller* at the University of Edinburgh, with the intention of helping users and software developers in theorem proving. Later these meta-languages have been standardized in what was called SML. It is not uncommon to refer to SML for theorem proving issues and lot of other theorem provers, like *HOL* [GM93] and *Isabelle* [Pau94b], are written with the same language. For further information about SML see the book *Programming with standard ML* [MCP93]. For an introduction to the proof assistant theory and how to implement a proof assistant in SML see [Pau94a].

The user that aims to prove properties with VCPT has to reason backward starting from the main goal. He shows the validity of the goal by applying proof rules that match a conclusion. If these rules have assumptions, they become new goals of the proof.

The major difference between VCPT and other theorem provers lies in the way VCPT manages the whole proof tree. In fact, the discharge rule works on the entire proof tree and needs to check all the ancestors of the current node. For this reason it is impossible to save in the memory just the node that we are proving, but it must be stored somewhere the entire proof tree of the property.

VCPT is not an automated theorem prover, so it does not provide sophisticated methods and tactics to handle logical formulas without interaction with user. However, being a proof assistant, VCPT helps users in proving properties by means of collection of available definitions, delaying the choice of witness and incremental/automatic checks for the discharge rule. The choice of a witness is a crucial step in lot of proofs. VCPT allows delaying the choice of the witness, introducing what it is usually called meta-variable. A meta-variable can be viewed as an abstract variable that can assume any value of its domain. In this way users can introduce meta-variables sooner, and later when more information about which value may be suitable for that variable are available, assign the value to the variable in order to make the proof sound.

There is also a support for graphical display of proofs called *DaVinci graph visualizer*. The DaVinci tool gives a graphical interface to visualize how the proof is going and to simply navigate the proof tree. Thus, with the command `display_proof()` from the VCPT prompt an instance of DaVinci tool will automatically start displaying the current proof. Nevertheless the DaVinci tool will not be explained anymore because it has not been used in this thesis.

4.1 Syntax and Theories

Proving properties with VCPT requires to collect theorems in a so-called *theory*. A theory is a set of *datatypes, predicates* and *theorems* to prove. During the implementation of the theory it is possible to augment the power of the tool by defining some ad-hoc tactics and helping functions in order to make the reasoning simpler. In fact the tool allows to embed SML-Code in the theory, or just call it through an external file. Thus, when a theory file is read, the SML-external code is loaded and since then tactics declared in that SML file can be used.

Another important characteristic of the tool is that it allows to build theories incrementally. A theory can use other theories and build up data and functions that are available in them.

All the elements that compose a VCPT theory will be discussed in the next sections.

Since VCPT uses many-sorted first-order structures for constructing datatypes, it is presented a brief description of those structures before introducing VCPT datatypes.

4.1.1 Many-Sorted First-Order Structures

In mathematics and logic, there is often a need for a formal way to define datatypes that will be used later.

Datatypes are constructed in VCPT as terms of what is a so-called many-sorted first-order structure. In order to generate datatypes, we need to specify *sorts* and *functions* ranging over those sorts. Function symbols are applied to a number of terms of a specific sort. For more information about signatures and structures in many-sorted contexts see [MT92].

Definition 4.1.1 (Many-Sorted First-Order Structure) A many-sorted first-order structure is given by a tuple $\Sigma = (\langle S, F \rangle)$ where S is a set of sorts and F is a set of function-symbols. For each f in F there is also an annotation for the type of the function $s_1 \times s_2 \times \ldots \times s_n \rightarrow s$, with $s_1, s_2, \ldots, s_n, s$ from S. The s_i, for $1 \leq i \leq n$, are the sorts of the arguments, and s is the sort of the result.

Given a structure Σ we write $f(t_1, t_2, \ldots, t_n)$ for the application of f to the subterms $t_1, t_2, \ldots, t_n$. Each subterm $t_1, t_2, \ldots, t_n$ is a term of a specific sort s_i. If a term t receives no argument as input, then we simply write t instead of $t()$.

Definition 4.1.2 (Terms of signature Σ) The set $T = (\Sigma, V)$ represents the set of all terms in our signature Σ, with variables from the set V. The variables themselves belong to a specific sort s_i. We write $T = (\Sigma, V)_s$ for the set of $\{x \mid x \in T = (\Sigma, V), x \text{ of sort } s\}$. Thus, $T = (\Sigma, V)_s$ is the set of all the terms of sort s in Σ. When V is empty, it is possible to omit the variables and consider only the ground terms $T = (\Sigma)$.

$T = (\Sigma, V)$ is defined inductively using the following two rules:

- $v \in T = (\Sigma, V)$ iff $v \in V$
- $f(x_1, x_2, \ldots, x_n) \in T = (\Sigma, V)_s$ where f is a function from sorts $s_1, s_2, \ldots, s_n$ to sort s, iff forall $1 \leq i \leq n$, $x_i \in T = (\Sigma, V)_{s_i}$.

Example 4.1.3 shows a many-sorted first-order structure with the intent of clarifying the previous concepts.

Example 4.1.3 Suppose we want to declare a new datatype, *nat*, that represents natural numbers. A natural number can be either zero or a successor of another natural number. Thus, we can declare a new many sorted first-order structure *nat* with one constant *zero* and a function *succ* that takes an element of the sort *nat* and returns another element of the same sort. Our signature Σ is $(nat, \{zero, succ\})$ with the annotation:

Function	**Type**
zero	nat
succ	nat $\rightarrow$ nat

Table 4.1: Example of many-sorted structure

4.1.2 DataTypes

In the rest of this thesis, the words type and sort are used as synonyms.

In VCPT, it is possible to specify a new sort s with constructors $f_1,\ldots,\ f_n$, and with annotations $t_1,\ldots,\ t_n$, as presented in Table 4.2:

datatype	s	=	f_1	**of**	t_1
		|	f_2	**of**	t_2
				$\vdots$	
		|	f_n	**of**	t_n

Table 4.2: Syntax of datatypes in VCPT

The annotation t_i could eventually be empty and in that cases the keyword *of* is omitted.

Declaring a sort s as in Table 4.2 creates a new freely generated sort(see Section 4.1.1) unless it is differently specified inserting the keyword *nonflat* before the word datatype in the declaration.

The tool provides a simple way to build lists of sorts. A list of elements of sort s can be declared using the notation *s list*.

VCPT allows to declare lists in two different ways:

1. Listing all the elements of type s of the list: $[s_1, \ldots, s_n]$.
2. Listing some head elements (at least one) of type s of the list, and then a tail S of type *s list*: $[s_1, \ldots, s_n \mid S]$.

There are also more sophisticated features to specify sorts, like using sub-sorts or predicates, but we refrain to discuss them here because these features have not been used in this thesis.

4.1.3 More constructors

With the basic simple constructors it is possible to define and declare a lot of formulas. However, using only the simple operators that logic provides, makes formulas grow substantially. VCPT allows users to use some derived constructors, like *if-then-else* or *case*, to make the definition of formulas more user friendly.

For example with VCPT it is possible to write a formula as in Table 4.3:

if	ϕ	**then**
	ψ	
else		
	γ	

Table 4.3: If-then-else constructor in VCPT

The if-then-else constructor does not introduce something new, it is simply an abbreviation of the formula : $(\phi \rightarrow \psi) \wedge (\neg\phi \rightarrow \gamma)$.

Keeping in mind we are dealing with algebraic datatypes, it is useful to have a constructor for matching an element of a certain datatype with some alternatives. The operator is called *case* and it is defined by the syntax in Table 4.4:

case	f	**of**	
	f_1	$\Rightarrow$	ϕ_1
|	f_2	$\Rightarrow$	ϕ_2
	$\vdots$		
|	f_n	$\Rightarrow$	ϕ_n

Table 4.4: Case constructor in VCPT

The formulas f_i in Table 4.4 might have any number of free variables. For this reason the case constructor might return more than one alternative for the formula *f*. More specifically, let $FV_1, \ldots, FV_n$ be the free variables of $f_1, \ldots, f_n$ we can say that if there exists an assignment to those free variables such that $f=f_i$, then ϕ_i.

Suppose to have the case constructor in Table 4.5, with variable *Y,X* ranging over natural numbers:

case	Y	**of**	
	2	$\Rightarrow$	ϕ_1
|	5	$\Rightarrow$	ϕ_2
|	X	$\Rightarrow$	ϕ_3

Table 4.5: Example of case constructor in VCPT

For any value of *Y*, the case will always match at least the last rule, so that it will return ϕ_3. Eventually, if the value of *Y* is either 2 or 5, it will return also ϕ_1 or ϕ_2.

4.1.4 Predicates

Predicates have a center role in theories. They allow users to define relations among datatypes.

A predicate can be declared in three different ways:

1. Simple predicate.
2. Greatest Fixed Point.
3. Least Fixed Point.

The simple predicate, *pred*, that takes in input the variables $V_1, \ldots, V_n$ of sort $s_1, \ldots, s_n$ and returns ϕ, can be declared as follows:

```
pred : s1 → ...  sn→ prop =
  λ V1:s1.  → ... λ Vn:sn.
  φ
```

The simbol = changes to <= or => if the predicate is respectively a least fixed point or a greatest fixed point.

The variables $V_1, \ldots, V_n$ have to be bound in ϕ. Similarly, if the predicate *pred* is declared as greatest/least fixed point, *pred* must be bound in ϕ.

There is also the possibility of using the constructor:

```
LET: def WHERE def-list END
```

The *let-where* constructor exports only one definition, *def*, but allows to call all the definitions in *def-list*, in *def*, and vice-versa. This notation is particularly interesting and useful to declare nested fixed points, or on the other hand, to make definitions more clear and concise.

4.1.5 Notation

Due to the restriction of VCPT in accepting only ASCII characters it is necessary to find a way for defining non-ASCII symbols like $\wedge$ or $\vee$.

In the Table 4.6 is given the notation used for non-ASCII symbols.

Symbol	Equivalent	Symbol	Equivalent
$\Leftarrow$	<=	$\wedge$	/\ or **and**
$\Rightarrow$	=>	$\vee$	\/ or **or**
$\top$	**tt**	$\rightarrow$	−− >
$\bot$	**ff**	$\neg$	**not**
μ	**lfp**	$\exists$	**exists**
ν	**gfp**	$\forall$	**forall**
λ	\	$\vdash$	|−
$\times$	*	|	|

Table 4.6: Table for non-Ascii symbols used in VCPT

4.2 Rules and Tacticals

The rules to handle the proof goal are implemented in SML. Rules are standard functions from a sequent (the current goal, conclusion) to a list of sequents (the premises) and a list of assignments to the meta-variables caused by the rule. Usually rules are applied to a particular formula in the sequent, so often they need a parameter to identify the position of such formula in the sequent considered.

4.2.1 Rules

Rules are grouped according to the kind of formulas for which they are applicable. Thus, we have:

Structural Rules

These rules do not care how formulas are and what is their internal structure.

- **id**: The axiom rule, succeeds if there is both side of turnstile the same formula ϕ. It raises an exception otherwise.

- **idmon int int**: It succeeds if there is the same approximated fixed point both side of turnstile. The two indices indicate where the approximated fixed point formulas are located. It raises an exception otherwise.
- **exitid**: It succeeds if there is the same formula both side of turnstile with respect of an assignment to their meta-variables.
- **weak_l int**: The indexed formula on the left is weakened out.
- **weak_r int**: The indexed formula on the right is weakened out.
- **contract_l int**: A new formula like the one indexed by *int* is created on the left.
- **contract_r int**: A new formula like the one indexed by *int* is created on the right.
- **cut formula**: Two sequents are created. One with the *formula* on the right and the other with the *formula* on the left.
- **term_abtract_r termabstractor int**: With the help of *termabstractor* the indexed formula on the right is converted to an quivalent using abstraction and application.

VCPT has lot of more complex structural rules that have not been used in this thesis. For this reason, we skip their introduction in this section. For more information about all the structural rules that can be used with VCPT see the user manual of the tool.

Logical Rules

Rules handling logical operations of *μ-calculus*.

- **false_l int**: This rule succeeds if there is a false formula on the left side of the turnstile at index *int*. It raises an exception otherwise.
- **false_r int**: The indexed formula to the right side of the turnstile is deleted. An exception is raised otherwise.
- **true_l int**: The indexed formula to the left is deleted. An exception is raised otherwise.
- **true_r int**: The rule succeeds if there is a true formula on the left side at index *int*. It raises an exception otherwise.
- **not_l int**: The indexed formula to the left is deleted and the same formula without the negation connective is placed on the right. An exception is raised otherwise.
- **not_r int**: The indexed formula to the right is deleted and the same formula without the negation connective is placed on the left. An exception is raised otherwise.
- **and_l int**: If the indexed formula on the left is a conjunction, it is replaced by the two conjuncts. An exception is raised otherwise.
- **and_r int**: If the indexed formula on the right is a conjunction then two sequents are created. In one sequent the formula is replaced by the right conjunct and in the other sequent the formula is replaced by the left conjunct. An exception is raised otherwise.
- **or_l int**: If the indexed formula on the left is a disjunction then two sequents are created. In one sequent the formula is replaced by the right disjunct and in the other sequent the formula is replaced by the left disjunct. An exception is raised otherwise.

- **or_r int**: If the indexed formula on the right is a disjunction, it is replaced by the two disjuncts. An exception is raised otherwise.
- **implication_l int**: If the indexed formula to the left is an implication, two sequents are created. In the first sequent the formula is replaced by the consequent of the implication while in the other sequent the antecedent of the implication is moved to the right. An exception is raised otherwise.
- **implication_r int**: If the indexed formula to the right is an implication, the antecedent is moved to the left and the consequent remains to the right. An exception is raised otherwise.
- **unfold_l int**: The indexed formula to the left is unfolded if it is a fixed point. An exception is raised otherwise.
- **unfold_r int**: The indexed formula to the right is unfolded if it is a fixed point. An exception is raised otherwise.
- **approx_l int**: If the indexed formula to the left is a least fixed point, it is approximated. An exception is raised otherwise.
- **approx_r int**: If the indexed formula to the right is a greatest fixed point, it is approximated. An exception is raised otherwise.
- **unfold_app_l int**: If the indexed formula to the left is an approximated least fixed point, it is unfolded. An exception is raised otherwise.
- **unfold_app_r int**: If the indexed formula to the right is an approximated greatest fixed point, it is unfolded. An exception is raised otherwise.
- **apply_l int**: If the indexed formula to the left is an application, it is β-reduced. An exception is raised otherwise.
- **apply_r int**: If the indexed formula to the right is an application, it is β-reduced. An exception is raised otherwise.

Transition Rules

Since VCPT consents declaring formulas with the modal operators *box* and *diamond*, there is a need for rules to handle modal formulas.

In this project it has been used first-order μ-*calculus*. Formulas, then, never contained modal operators. For this reason, we do not discuss the transition rules. More information can be retrieved in the user manual of the tool.

Quantifier Rules

Rules to handle formulas with quantifiers.

- **exists_r (term opt) int**: If the indexed formula to the right is an existential quantifier, the existential variable is replaced by the *opt* term. If the *opt* term is *NONE* a new fresh variable is created. An exception is raised otherwise.
- **exists_l int**: If the indexed formula to the left is an existential quantifier, the existential variable is replaced by a new fresh variable. An exception is raised otherwise.

- **forall_l (term opt) int**: If the indexed formula to the left is an universal quantifier, the universal variable is replaced by the *opt* term. If the *opt* term is *NONE* a new fresh variable is created. An exception is raised otherwise.
- **forall_r int**: If the indexed formula to the right is an universal quantifier, the universal variable is replaced by a new fresh variable. An exception is raised otherwise.

Equality Rules

Rules to handle equalities.

- **eq_r int**: The rule succeeds if the indexed formula to the right is an equality. An exception is raised otherwise.
- **eq_l int**: If the indexed formula to the left is an equality it is weakened out. An exception is raised otherwise.
- **eq_subst int**: If the indexed formula to the left is an equality in which one term is a variable, that variable is substituted throughout the whole sequent and the equality is weakened out. An exception is raised otherwise.
- **eq_flat_decomp_l int**: If the indexed formula to the left is an equality in which the compared terms have the same head constructor applied to the same number of arguments, then the equality is replaced by a list of equalities comparing component-wise the respective arguments. An exception is raised otherwise.
- **eq_flat_decomp_r int**: If the indexed formula to the right is an equality in which the compared terms have the same head constructor applied to the same number of arguments, then this number of copies of the sequent is created, and in each of these the equality is replaced by an equality comparing the respective components. An exception is raised otherwise.
- **eq_flat_elim_l int**: If the indexed formula to the left is an equality in which the compared terms are from a flat datatype, and if either the head constructor or their number of arguments are different, then the rule results in an axiom. An exception is raised otherwise.
- **eq_flat_elim_r int**: If the indexed formula to the right is an equality in which the compared terms are from a flat datatype, and if either the head constructor or their number of arguments are different, then this equality is weakened out. An exception is raised otherwise.

Lemma Rules

Rules to handle and introduce lemmas.

- **lemmadischarge string**: Discharge the current node if it is an instance of the lemma indicated by the *string*.
- **lemma string**: Introduce the lemma indicated by the *string* to the left side of the turnstile.
- **app_lemma_lhs string**: It tries to find formulas on the left side of turnstile that match the assumptions of the lemma indicated by the *string*, then it introduces the conclusion to the left.

- **app_lemma_rhs string**: It tries to find formulas on the left side of turnstile that match the conclusion of the lemma indicated by the *string*, then it introduces the assumptions to the right.

Other Rules

Some other general and useful rules to handle formulas in sequents.

- **insert_at_l int∗int**: It moves the left formula indicated by the left index to the left position indicated by the right index.
- **insert_at_r int∗int**: It moves the right formula indicated by the left index to the right position indicated by the right index.
- **assign_var term term'**: It assigns the term *term* to the meta variable *term'*.
- **add_type_info term**: It adds a new formula on the left side of the turnstile that describes the type of the term *term*.

As usual we skip the description of all the rules we did not use in this thesis.

4.2.2 Tacticals

Tacticals are a way to produce new tactics. More specifically, tacticals combine basic tactics in order to create more complex and powerful tactics.

In this way the user can build ad-hoc strategies to better deal with his specific problem.

- **t_skip**: Tactical that always succeeds. It simply returns the current sequent.
- **t_fail**: Tactical that always fails. It raises always an exception.
- **t_compose tactic (tactic list)**: As first the tactical applies the *tactic* to the current sequent. Then for each sequent returned, it applies the corresponding tactic in the *tactic list*. Thus, the first element of the *tactic list* will be applied to the first sequent returned and so on. The tactical fails if either any tactics fail or the number of sequent returned from the first tactic differs from the length of the *tactic list*.

Example 4.2.1 (t_compose) Simple example of tactical t_compose.

```
prove `` (5 = 5) ⇒ (4 = 5) ⊢ '';
by(t_compose (implication_l 1) [eq_r 1, eq_flat_elim_l 1]);
```

The tactical will succeed. First the application of rule *implication_l 1* on the left hand side will produce two subgoals. After, rule *eq_r 1* will solve the first subgoal and rule *eq_flat_elim_l 1* will solve the second subgoal concluding the proof.

- **t_compose2 tactic1 tactic2**: The tactical applies first *tactic1* to the current sequent, then it applies *tactic2* to all the resulting sequents.
- **t_compose_l (tactic list)**: The tactical is a generalization of t_compose2. It applies all the tactics in the list to all the successive sequents.
- **t_orelse tactic1 tactic2**: The tactical tries to apply *tactic1* and if it fails, then it applies *tactic2*.

- **t_orelse_l (tactic list)**: The tactic is a generalization of t_orelse, it tries to apply the first tactic in the *tactic list*, if it fails then applies the second. If also the second tactic fails, it applies the third and so on.

- **t_if tactic1 tactic2 tactic3**: The tactical applies *tactic1* and in case it succeeds it applies *tactic2* otherwise it applies *tactic3*.

- **t_while tactic**: The tactical continue to apply *tactic* everywhere until it fails somewhere.

- **t_pretactic (sequent→'a option) ('a→tactic)**: This tactical takes as first argument a function from the current sequent to some return value. If there is no return value an exception is raised. The second argument is a function from the return value to a tactic.

Example 4.2.2 (t_pretactic) Simple example of t_pretactic tactical.

```
prove `` ⊢ 4 = 5, 3 = 2, tt ``;
val apply_pretactic_to_last_r =
t_pretactic(fn seq ⇒ SOME(length(Sequent.get_rhs seq)));
by(apply_pretactic_to_last_r true_r);
```

The tactical will succeed. In fact the variable *apply_pretactic_last_r* first takes the current sequent and returns the length of its right hand side. After, it applies the tactic *true_r* to the return value (the sequent length, 3), succeeding.

- **t_try tactic**: The tactical returns true if the *tactic* succeeds and false otherwise.

- **t_fix 'a('a→('a→tactic)→tactic)**: This tactical is intended to express recursive behaviors. The tactical takes two arguments: one arbitrary value and a function from that value to the recursive function. An application of t_fix tactical is showed in the Example 4.2.3.

- **t_bool (sequent→bool) tactic1 tactic2**: The tactical applies *tactic1* if the first argument function returns true, otherwise it applies *tactic2*.

- **t_lhs (int→tactic)**: The tactical attempts to apply the *tactic* to the formula at position *int* of the left side of the turnstile.

- **t_rhs (int→tactic)**: The tactical attempts to apply the *tactic* to the formula at position *int* of the right side of the turnstile.

- **apply_to_last_l (int→tactic)**: The tactical attempts to apply the *tactic* to the last formula on the left side of the turnstile.

- **apply_to_last_r (int→tactic)**: The tactical attempts to apply the *tactic* to the last formula on the right side of the turnstile.

Now that we have a clear picture of all the tacticals we can use to make VCPT more powerful, we show a complete example used in our project to create a more sophisticated tactic.

The goal is to create an automatic tactic which simplifies all the connectives situated to the left or right side of the turnstile that do not generate branches.

Example 4.2.3 (unfold_all_non_branches) First we declare two functions which attempt to apply a list of tactics that do not create branches to a formula situated in a specific position (pos) of the left or right side of the turnstile.

```
fun something_to_do_l pos =
                         t_orelse_l [and_l pos,
                                     not_l pos,
                                     exists_l pos,
                                     t_fail
                                    ]
fun something_to_do_r pos =
                          t_orelse_l [ or_r pos,
                                       not_r pos,
                                       forall_r pos,
                                       implication_r pos,
                                       t_fail
                                     ]
```

The *t_fix* tactical will be used to apply the previous functions (`something_to_do_l` and `something_to_do_r`) to all formulas to the left and right side of the turnstile.

```
val unfold_all_non_branches_left =
    t_fix
     1
    (fn pos =>
         (fn recurse =>
           t_pretactic
           (fn seq =>
            if pos>length(Sequent.get_lhs seq) then NONE
            else SOME ())
           (fn _ => t_orelse (something_to_do_l pos) (recurse (pos+1))
                            )))

val unfold_all_non_branches_right =
    t_fix
     1
    (fn pos =>
         (fn recurse =>
           t_pretactic
           (fn seq =>
            if pos>length(Sequent.get_rhs seq) then NONE
            else SOME ())
           (fn _ => t_orelse (something_to_do_r pos) (recurse (pos+1))
                            )))
```

In the end, function `unfold_all_non_branches_left` and function `unfold_all_non_branches_right` are merged into one single tactical that operates both sides of the turnstile.

```
val unfold_all_non_branches =
   t_while(
           t_orelse_l [unfold_all_non_branches_left,
                       unfold_all_non_branches_right]
          )
```

4.3 A complete Example

A complete example of a VCPT theory and its related proof is presented here in order to clarify all the concepts introduced in the last sections.The example is exactly the same introduced in Section 3.4.

The theory models the even and odd natural numbers and shows that each natural number is either even or odd.

As first, a datatype *nat* which represents the natural numbers is declared. Furthermore, two predicates *even* and *odd* which respectively define when a natural number is either even or odd, are introduced. The predicate even simply states that a natural number is even if it is *zero* or a double successor of another even number. The predicate odd is dual of the predicate even.

```
THEORY EvenOdd
DEFINITIONS

DATATYPES

datatype nat =    zero
                | succ of nat

END

PREDICATES

even : nat --> prop <=
\N : nat.
N = zero() \/ (exists N':nat. even N' /\ N = succ(succ(N')))
end;

odd : nat --> prop <=
\N : nat.
N = succ(zero()) \/ (exists N':nat. odd N' /\ N = succ(succ(N')))
end

END
```

Once the theory is defined, it is possible to introduce the property that must be proven.

```
/* Proof Goal */
Forall X:nat. (nat X => (even X \/ odd X))
```

The proof is showed in detail highlighting all the rules applied at each step.
As first, the following rules are applied:

```
Forall_r 1, Implication_r 1, Or_r 1.
```

The new goal will be:

```
1) nat X0 |- even X0, odd X0
```

Next rules approximate and unfold the least fixed point to the left.

```
RULES :
Approx_l 1, Unfold_app_l 1, Apply_l 1, Or_l 1.
```

Producing two goals:

```
2.1) X0 = 0 |- even X0 , odd X0
2.2) Exists Z:nat. X0=succ(Z) /\ nat Z |- even X0, odd X1
```

In order to solve point 2.1 it is enough to substitute the value of X0 left side, to the right, and after to unfold the definition of even number to the right.

```
RULES :
Eq_subst 1, Unfold_r 1, Apply_r 1, Or_r 1, Eq_r 1.

2.1.1) |- even 0, odd 0
```

Node 2.1.1 is trivially true due to the definition of predicate even.
We focus now to point 2.2 . First we apply the And rule to the right of the turnstile, and after we substitute the definition of X0 on the left, to the right. Unfolding again the definition of natural number it will produce two branches.

```
RULES :
Exists_l 1, And_l 1, Eq_subst 2

2.2) nat X1 |- even succ (X1), odd succ(X1)

RULES :
Unfold_app_l 1, Apply_l 1, Or_l 1.

2.2.1) X1 = 0 |- even succ (X1), odd succ(X1)
2.2.2) Exists Z:nat. X1=succ(Z) /\ nat(Z) |- even succ(X1), odd succ (X1)
```

The goal 2.2.1 is trivial to solve. The only thing to do is to substitute the variable X1 = 0 left side, to the right, obtaining a true formula (odd succ 0).

```
RULES :
Eq_subst 1, Unfold_r 2, Apply_r 2, Or_r 2, Eq_r 2.

2.2.1) |- even succ 0, odd succ 0
```

To show the validity of the goal 2.2.2 we save in a fresh variable the new variable that will be introduced through the existential rule left hand side. After, we apply the And rule to the left hand side.

```
RULES :
val X2 = Exists_l 1, And_l 1.

2.2.2) X1=succ(X2), nat(X2) |- even succ (X1), odd succ (X1)
```

Substituting the variable X1 right hand side with its definition on the left and unfolding the definitions of even and odd numbers to the right, we have:

```
RULES :
Unfold_r 1, Apply_r 1, Unfold_r 2, Apply_r 2, Or_r 1, Or_r 3, Eq_subst 2.

2.2.2) nat (X2) |- succ succ X2 = 0,
                   Exists Z:nat. even Z /\ succ succ X2 = succ succ Z,
                   succ succ X2 = succ 0,
                   Exists Z:nat. odd Z /\ succ succ X2 = succ succ Z
```

Notice that instead of choosing a fresh variable for Z it is possible to substitute Z with X2, obtaining:

```
RULES :
Exists_r (SOME X5) 2, Exists_r (SOME X5) 4

2.2.2) nat (X2) |- succ succ X2 = 0,
                   even X2 /\ succ succ X2 = succ succ X2,
                   succ succ X2 = succ 0,
                   odd X2 /\ succ succ X2 = succ succ X2
```

The application of the And rule to the right (2nd formula) produces two branches:

```
RULE :
And_r 2

2.2.2.1)  nat (X2) |- succ succ X2 = 0,
                      even X2
                      succ succ X2 = succ 0,
                      odd X2 /\ succ succ X2 = succ succ X2

2.2.2.2)  nat (X2) |- succ succ X2 = 0,
                      succ succ X2 = succ succ X2,
                      succ succ X2 = succ 0,
                      odd X2 /\ succ succ X2 = succ succ X2
```

Using again the And rule to the right (4th formula):

```
RULE :
And_r 4

2.2.2.1.1)  nat (X2) |- succ succ X2 = 0,
                        even X2
                        succ succ X2 = succ 0,
                        odd X2

2.2.2.1.2)  nat (X2) |- succ succ X2 = 0,
                        even X2
                        succ succ X2 = succ 0,
                        succ succ X2 = succ succ X2
```

```
2.2.2.2)  nat (X2) |- succ succ X2 = 0,
                      succ succ X2 = succ succ X2,
                      succ succ X2 = succ 0,
                      odd X2 /\ succ succ X2 = succ succ X2
```

The proof is almost finished.
The goal 2.2.2.1.1 can be discharged against node 1 due to the VCPT discharge condition. Finally for goals 2.2.2.1.2 and 2.2.2.2 it will be enough to apply the equality rule on the right (respectively on the 4th and 2nd formula) to complete the proof.

```
RULES:
discharge1(), Eq_r 4, Eq_r 2.
```

Chapter 5

A solution of Poplmark with VCPT

It is presented here a solution for part 1a of *POPLMark Challenge* using the *de Bruijn indices* representation previously discussed in Chapter 3. The first point of the challenge deals uniquely with the type language of *System* $F_{<:}$. Further information about the problem description can be found in [ABF+05] and Chapter 2.

In the following sections we show the formalization of the basic calculus and as main result we prove the *transitivity of subtyping*.

5.1 The Poplmark Theory

The VCPT file *Poplmark.thy* is the core of the project. It contains *datatypes*, *predicates* and *lemmas* needed, in order to prove the main property.

5.1.1 Datatypes

Since the grammar of *System* $F_{<:}$ is quite simple, we do not need to introduce any sophisticated datatype. Furthermore dealing exclusively with the first point of the challenge, does not require to implement and encode anything concerning terms and possible operations on them.

Thanks to the above simplifications only three different kind of datatypes are introduced:

- Type
- Binding
- Environment

Using the de Bruijn representation, a type can then be the *maximum type*, a *type variable* (simply a natural number), a *type function* or the *universal type*. The above specification can straightforwardly be encoded in VCPT declaring the following datatype:

```
/* Datatype Type */
datatype ty =   tyvar of nat
              | top
              | tyfun of ty * ty
              | tyall of ty * ty;
```

Similarly we introduce the concepts of *binding* and *environment*, where an environment simply is a list of bindings, while a binding binds a term or a type to a specific type inside an environment. In the following notation the keywords *varb* and *tvarb* are used to refer respectively to bindings for terms and types.

```
/* Datatype Binding */
datatype binding =   varb of ty
                   | tvarb of ty;

/* Datatype Environment */
         env = binding list;
```

5.1.2 Predicates

Once datatypes are defined it is possible to introduce predicates required to prove the intended goal of the project. Since a predicate takes as input a certain number of parameters (*datatypes*) and returns a proposition that can be either true or false, it is natural to classify predicates into categories according to the datatypes they receive as input. In this thesis there are four different kinds of predicates reflecting the parameter's type received:

- Type Predicates
- Environment Predicates
- Subtyping and Narrowing Predicates
- Transitivity and Narrowing Propositions

Type Predicates

The main operation performed on types is shifting (See Chapter 3 for further information). The *typeshift_general* predicate takes four parameters: a type to be shifted, two indices such that the first indicates the increment that each free variable has to receive and the second called *cutoff* that keeps track of how many binders we have crossed, and another type intended to be the result of the shifting operation.

```
/* TypeShift Predicate */
typeshift_general : ty --> nat --> nat --> ty --> prop <=
                    \ T:ty. \D:nat. \C:nat. \Tcurry:ty.
  cases T of
         tyvar(X) => if(natLess X C)
                     then (Tcurry = tyvar(X))
                     else (exists Z:nat.(natPlus X D Z ∧ Tcurry = tyvar(Z)))
       | top() => Tcurry = top()
       | tyfun(T1,T2) =>
         (exists T3:ty. exists T4:ty.
          typeshift_general T1 D C T3 ∧
          typeshift_general T2 D C T4 ∧ Tcurry = tyfun(T3 , T4)
         )
       | tyall(T1,T2) =>
         (exists Tx:ty. exists Z:nat. exists T3:ty.
          typeshift_general T1 D C Tx ∧ natPlus 1 C Z ∧
          typeshift_general T2 D Z T3 ∧ Tcurry = tyall(Tx , T3)
         )
  end
end;
```

The following listing gives two examples showing how the *typeshift_general* predicate works on two different kinds of types, a simple type variable and an universal type, supposing to shift all the free variables in them.

Example 5.1.1 `typeshift_general tyvar(1) 1 0 Tcurry:`
In this case the predicate matches the first inference rule so it will check if natLess 1 0 and consequently it will produce that `exists Z:nat.(natPlus 1 1 Z ∧ Tcurry = tyvar(Z))`. Basically the function just increment by one the index of our type variable.

Example 5.1.2 `typeshift_general tyall(tyvar(2),tyvar(1)) 1 0 Tcurry:`
In this second case, the predicate will match the last inference rule so it will produce two new shiftings on the types that compose the universal type. In this process we need to increase the *cutoff* of the second type because a binder has been crossed. Thus, we obtain: `(exists Tx:ty. exists Z:nat. exists T3:ty. typeshift_general tyvar(1) 1 0 Tx ∧ natPlus 1 0 Z ∧ typeshift_general tyvar(1) 1 Z T3 ∧Tcurry = tyall(Tx,T3))`. Now we are again in the situation in which we have to shift two simple type variables(So like example 5.1.1).
Notice that later, the second type of `tyall(tyvar(2),tyvar(1))` will not be shifted. In fact that type variable was bound in the original type from the first binder.

Proposition *size* determines the size of a generic type.

```
/* Function that returns the size of a type */
size : ty --> nat --> prop <=
       \T:ty. \R:nat.
       cases T of
             top()        => R=0
            |tyvar(X)     => R=0
            |tyfun(T1,T2) => exists Z:nat. exists R1:nat. exists R2:nat.
             (natPlus R1 R2 Z ∧ size T1 R1 ∧ size T2 R2 ∧ R=Z+1 )
            |tyall(T1,T2) => exists Z:nat. exists R1:nat. exists R2:nat.
             (natPlus R1 R2 Z ∧ size T1 R1 ∧ size T2 R2 ∧ R=Z+1)
       end
end;
```

Environment Predicates

Predicates to check properties and perform operations on environments are required.

The first predicate introduced is called *get_tvar* and its role is to take the type of a *type variable binding* from a generic environment. The crucial aspect of this operation is that we have to shift the final type as many times as the number of binders crossed during the search.

```
/* Get_tvar Predicate */
get_tvar : env --> nat --> ty --> prop <=
           \E:env. \X:nat. \T1:ty.
 cases E of
         []:env => ff
       | [varb(T)|E'] => get_tvar E' X T1
       | [tvarb(T)|E']=>
                     cases X of
                             0    => typeshift_general T 1 0 T1
                            |X'+1 => (exists T':ty.
```

```
                                    typeshift_general T' 1 0 T1 ∧
                                    get_tvar E' X' T'
                                   )
                    end
 end
end;
```

Predicates (named *remove_var* and *insert_bound*) respectively remove a variable binding and insert a type variable binding in an environment. In the same fashion of the predicate *get_tvar* after introducing a new type variable binding, all the free variables within the type have to be shifted each time a type variable binding is crossed. This is to ensure that the environment will remain well formed after the insertion.

Here it is the definition of the two predicates:

```
/* Remove_var Predicate */
remove_var : env --> nat --> env --> prop <=
            \E:env.\X:nat.\E':env.
 cases E of
         []:env => E' = []:env
        |[varb(T)|TLE]=> cases X of
                            0    => E' = TLE
                           |X'+1 =>(exists TLE':env.
                                      remove_var TLE X' TLE' ∧
                                      E'=[varb(T)|TLE']
                                     )
                       end
        |[tvarb(T)|TLE]=> (exists TLE':env.
                           E' = [tvarb(T)|TLE'] ∧
                           remove_var TLE X TLE'
                          )
 end

end;


/* Insert_bound Predicate */
insert_bound: nat --> env --> env --> prop <=
              \X:nat. \E:env. \E':env.
 cases X of
         0=>cases E' of
                [tvarb(T)|E]=>wf_type E T
            end
       |X=>cases E' of
              [varb(Z)|E2]=>cases E of
                                [varb(T)|E1]=>typeshift_general T 1 X Z ∧
                                              insert_bound X E1 E2
                            end
           end
       |X'+1=>cases E' of
                [tvarb(Z)|E2]=>cases E of
                                [tvarb(T)|E1]=>typeshift_general T 1 X' Z ∧
                                               insert_bound X' E1 E2
                              end
             end
```

```
 end
end;
```

Predicate *wf_type* and predicate *wf_env* check respectively if a type is well formed in an environment and if an environment is well formed. An environment E is well-formed, if all types occurring in E only refer to type variables declared further to the right. Thus for example, if we have a `tyvar(2)` in our environment E and E is well-formed, it means that we can count at least two type variable binding starting from the right side of E, otherwise E would not be well-formed.

```
/* Well Formed Type Predicate */
wf_type : env --> ty --> prop <=
  \E:env. \T:ty.
           cases T of
                tyvar(X)     => exists T:ty. get_tvar E X T
                  | top()        => tt
                  | tyfun(T1,T2) => wf_type E T1 ∧ wf_type E T2
                  | tyall(T1,T2) => wf_type E T1 ∧
                                    wf_type [tvarb(T1)|E] T2
           end
end;


/* Well Formed Environment Predicate */
wf_env : env --> prop <=
         \E:env.
          cases E of
                   []:env => tt
                 | [varb(T)|E']  => wf_type E' T ∧ wf_env E'
                 | [tvarb(T)|E'] => wf_type E' T ∧ wf_env E'
          end
end;
```

Subtyping and Narrowing Predicates

Here the subtyping and narrowing judgments are declared as inductive predicates. Each constructor of the predicates corresponds of an inference rule of the subtyping and narrowing relation presented in the paper [ABF+05].

```
/* Subtyping Predicate */
subt_rel : env --> ty --> ty --> prop <=
\E:env. \T1:ty. \T2:ty.
  cases T2 of
           top() => wf_env E ∧ wf_type E T1
          |tyvar(X) =>
                      cases T1 of
                               tyvar(X) => wf_env E ∧ wf_type E T1
                              |tyvar(Y) => exists T:ty.
                                           (get_tvar E Y T ∧ subt_rel E T T2)
                      end
          |tyfun(T21,T22) =>
                      cases T1 of
                                tyfun(T11,T12) => subt_rel E T21 T11 ∧
```

```
                                          subt_rel E T12 T22
                              |tyvar(Y) => exists T:ty.
                                          (get_tvar E Y T ∧ subt_rel E T T2)
                      end
          |tyall(T21,T22) =>
                       cases T1 of
                               tyall(T11,T12) => subt_rel E T21 T11 ∧
                                          subt_rel [tvarb(T21)|E] T12 T22
                              |tyvar(Y) => exists T:ty.
                                          (get_tvar E Y T ∧ subt_rel E T T2)
                       end
  end
end;


/* Narrowing Predicate */
narrow_rel : nat--> env--> env--> prop <=
 \X:nat. \E:env. \E':env.
   cases E' of
         [tvarb(T')|TLE] => cases E of
                              [tvarb(T)|TLE] =>
                                          cases X of
                                                 0    => subt_rel TLE T' T
                                          end
                             |[tvarb(T')|TLE'] =>
                                          cases X of
                                                 X'+1 => narrow_rel X' TLE' TLE ∧
                                                         wf_type TLE T'
                                          end
                             end
        |[varb(T')|TLE] => cases E of
                              [varb(T')|TLE'] => narrow_rel X TLE' TLE ∧
                                          wf_type TLE T'
                           end
  end
end;
```

Transitivity and Narrowing Propositions

As the last basic step the two propositions of *transitivity* and *narrowing* are introduced. They represent the main goal of our challenge.

The transitivity proposition states that in any environment E, if a type variable S is subtype of a type variable Q and in the same environment E the variable Q is subtype of T, then it must be the case that in E, the variable S is subtype of T.

The narrowing proposition is much less intuitive. We can informally give a hint of what it states saying that: if two environments E and E', are in narrowing relation, then if one type T is subtype of another type S in E, T is also subtype of S in E'.

In order to consider the first part of *POPLMark Challenge* solved, transitivity and narrowing propositions must hold for any kind of *types*. A description of the proof and how the proof evolves is presented in the next section.

```
/* Transitivity Proposition */
transitivity_prop : ty --> prop =
                     \Q:ty.
```

```
                    forall E:env. forall S:ty. forall T:ty.
                    subt_rel E S Q => (subt_rel E Q T => subt_rel E S T)
end;


/* Narrowing Proposition */
narrowing_prop : ty --> prop =
 \Q:ty.
 forall E:env. forall E':env. forall S:ty. forall T:ty. forall X:nat.
 narrow_rel X E E'=>(get_tvar E X Q=>(subt_rel E S T=>subt_rel E' S T))
end;
```

5.1.3 Lemmas

Lemmas are the centerpiece of the whole project. In order to achieve and prove the final goal, we have to build an infrastructure that allows us to deal with the concepts of the challenge. Taking a bottom-up approach, it is possible to start by proving simple and trivial theorems concerning natural numbers, that are the basis of the whole theory, and after reasoning on more complex properties.

However, in this section we maintain a high level of abstraction giving just an idea how lemmas are divided according to properties they state and deferring a more detailed elucidation in the next section.

Appendix B shows a listing (with no explanation) of all the lemmas that have been proved in this thesis.

Similarly as predicates, lemmas can be divided into five different categories:

1. Natural number lemmas
2. Shifting lemmas
3. Well-formedness lemmas
4. Subtyping and narrowing lemmas
5. Main proof lemmas

Once it has been shown that a certain number of lemmas are valid, we are ready to prove the main goal of the challenge.

5.2 The solution of the challenge

In this section guidelines of how the main proof evolves and which is the proof-tactic used to reach the intended goal, are given.

Due to the huge number of theorems needed it is impossible to present all of them. Instead it is given an introduction to the crucial and critical used in the solution of the challenge.

The section is divided into four subsections reflecting the kind of lemmas concerned.

5.2.1 Natural Number Lemmas

VCPT do not provide general lemmas for natural numbers like other theorem provers usually do, so it was required a theory to cover some well-known aspects and properties typical of naturals. Natural number lemmas will show properties like the *commutativity of sum*, or ordering properties.

The proofs of these lemmas are usually short and straightforward and since the natural numbers are the simplest entity of our theory, they can be proven with the basic command rules provided by the tool.

To give an idea of what these theorems talk about, a listing of some of them is presented here:

```
/* Commutativity of Naturals */
DECLARE X : nat, Y : nat, Z : nat IN
natPlus X Y Z |- natPlus Y X Z
END
```

```
/* Equality NatPlus */
/* If the sum of the two number A and D, returns two different numbers
   B and C, then B is equal to C */
DECLARE A : nat, B : nat, C : nat, D : nat IN
natPlus D A B, natPlus D A C|- B = C
END
```

```
/* Lemma lemma_trans_less_adding_positive1 */
/* If a natural X0 is strictly less then X1, then X0 is also
   strictly less then X1+1 */
DECLARE X0 : nat, X1 : nat  IN
natLess X0 X1 |- natLess X0 X1+1
END
```

The number of lemmas regarding naturals is quite high and usually they are proven in similar fashion, first showing the base case holds and then proving that exists a way to discharge the unproved nodes using the VCPT discharge condition discussed in chapter 4.

We highlight how the tool helps in proving these simple lemmas showing in details the proof of *Equality NatPlus* previously introduced.

Example 5.2.1 `Equality NatPlus:`
1)`natPlus D A B, natPlus D A C |- C=B`
2)Two cases: `D=0` and `D=E+1`.
3)The base case, when `D=0` is trivial because it implies that both `B` and `C` are equal to `A`.
4)If `D=E+1` then: `exists T:nat.  B=T+1 ∧ natPlus E A T , natPlus E+1 A C |- C=B`
5)`natPlus E A T, natPlus E+1 A C |- C=T+1`
6)With the same operations on the second natPlus we obtain:
7)`natPlus E A T, natPlus E A Z |- Z+1=T+1`
8)`natPlus E A T, natPlus E A Z |- Z=T`
9)Finally the VCPT discharge condition can be helpful to discharge the node because it has the same shape of the first one and the global well-founded condition holds. Thus, the proof is finished.

Example 5.2.2 shows the help received by the tool in proving the commutativity of the sum of natural numbers. The proof is quite intricate and long, so parts trivial and not essential will be skipped in order to highlight just the interesting points of the proof.

Before to start with details, lemma *lemma_natplus_sub* is introduced. The lemma *lemma_natplus_sub* states that if the sum of two natural numbers X, Y is equal to another natural number Z, then it has to hold that the sum of X and $Y + 1$ has to be equal to $Z + 1$. Using the VCPT notation, we can write:

α) `NatPlus X Y Z ⊢ NatPlus X Y+1 Z+1`

The proof of the *lemma_natplus_sub* is not shown. It works with induction on the natural number X.

The property to prove is the following:

Example 5.2.2 `Commutativity of Sum of Natural Numbers:`
1) `NatPlus X Y Z ⊢ Natplus Y X Z.`
Two possible cases with induction on `X` (`X=0` or `X=X'+1`):
1.1) `Y=Z ⊢ NatPlus Y 0 Z.`
1.2) `∃ Z':nat. Z=Z'+1,NatPlus X' Y Z' ⊢ NatPlus Y X'+1 Z.`
We focus our attention on node 1.1.
With induction on `Y` (`Y=0` or `Y=Y'+1`):
1.1.1) `Z=0 ⊢ NatPlus 0 0 Z.`(Trivial)
1.1.2) `Z=Y'+1, Nat Y' ⊢ NatPlus Y'+1 0 Z.`
Unfolding the definition of NatPlus on the right:
1.1.2) `Z=Y'+1, Nat Y' ⊢ ∃ Z':nat. Z=Z'+1∧NatPlus Y' 0 Z'.`
Choosing for the variable `Z'` the value `Y'` we have:
1.1.2.1) `Z=Y'+1, Nat Y' ⊢ Z=Y'+1.`(trivial)
1.1.2.2) `Nat Y' ⊢ NatPlus Y' 0 Y'.`
Last step of induction on `Y'` (`Y'=0` or `Y'=Y''+1`).
1.1.2.2.1) `Nat 0 ⊢ NatPlus 0 0 0.`(Trivial)
1.1.2.2.2) `Nat Y'' ⊢ NatPlus Y''+1 0 Y''+1.`
Unfolding again the definition of NatPlus on the right:
1.1.2.2.2.1) `Nat Y'' ⊢ Y''=Y''.`(Trivial)
1.1.2.2.2.2) `Nat Y'' ⊢ NatPlus Y'' 0 Y''.`
Notice now that the last node can be discharged against node
labelled with 1.1.2.2 completing this part of the proof.
It remains to prove node 1.2.
1.2) `∃ Z':nat. Z=Z'+1,NatPlus X' Y Z' ⊢ NatPlus Y X'+1 Z.`
Introducing a fresh variable for `Z'` we obtain:
1.2) `NatPlus X' Y Z' ⊢ NatPlus Y X'+1 Z'+1.`
Using lemma *lemma_natplus_sub* (labelled as α above),
to add a new formula on the right:
1.2) `NatPlus X' Y Z' ⊢ NatPlus Y X'+1 Z'+1, NatPlus Y X' Z'`
Finally node 1.2 is discharged against node 1 terminating the proof.

We refrain from discussing the natural number lemmas in details as they are quite simple and independent from the theory considered.

5.2.2 Shifting Lemmas

A more interesting role in our theory is covered by the *Shifting lemmas*. Using the compact notation of $\uparrow n\ k\ T$ to indicate the increment of all the free variables of T with index $\geq k$ by n, it can be shown that some simple arithmetic properties of shifting hold.

Examples of such properties are:

$$i \leq j \Rightarrow j \leq i + m \Rightarrow \uparrow n\ j\ (\uparrow m\ i\ T)\ =\ \uparrow (m+n)\ i\ T$$

$$i + m \leq j \Rightarrow \uparrow n\ j(\uparrow n\ i\ T)\ =\ \uparrow m\ i\ (\uparrow n\ (j-m)\ T)$$

Properties of that kind can be found in papers by Barras and Werner [BW] and Nipkow [Nip01] and they can usually be proven by structural induction on terms and types.

Keeping in mind we are dealing only with the first part of the challenge we can forget about substitution properties focusing our attention uniquely on the shifting properties.

Some simple lemmas direct consequences of the definition of the shifting predicate are introduced. Shifting by one a simple type variable `tyvar(X)` (`typeshift_general tyvar(X) 1 Y T`) can lead to two situations:

1. X is less than Y, so the variable is bound and we do not have to shift.
2. X is not less than Y, so we have to shift. In this case, the result type variable T is equal to *X+1*.

```
/* Lemma typeshift_shift_or_natless_indx_tyvar */
DECLARE X:nat, Y:nat, T:ty IN
typeshift_general tyvar(X) 1 Y T
|-
(typeshift_general tyvar(X) 1 Y tyvar(X+1) \/  natLess X Y)
END
```

```
/* Lemma typeshift_no_shift_tyvar */
DECLARE X:nat, Y:nat, T:ty IN
typeshift_general tyvar(X) 1 Y T, natLess X Y
|-
T=tyvar(X)
END
```

These lemmas come straight from the first introduction rule of the shifting predicate and they do not need any further comments.

Then we show that typeshift_general is a partial function.

```
/* Lemma lemma_same_typeshift_general */
DECLARE N:nat, C:nat, T:ty, T':ty, T'':ty IN
typeshift_general T N C T', typeshift_general T N C T''
|-
T' = T''
END
```

In this case the validity of the property is shown by induction on the type T. The two basic cases (when `T=Top` or `T=tyvar(Y)`) are trivial so we focus our attention on how the induction principle works when `T=tyfun(T1,T2)`(the principle is exactly the same when `T=tyall(T1,T2)`.

The example is particularly interesting because it shows how the VCPT discharge condition can help users in proving properties.

Starting from:

```
1)  typeshift_general T N C T',
    typeshift_general T N C T''
    |-
    T' = T''
```

When T=tyfun(T1,T2) happens that:

```
2)  typeshift_general tyfun(T1,T2) N C T',
    typeshift_general tyfun(T1,T2) N C T''
    |-
    T' = T''
```
If we unfold the two predicates:
```
3)  (exists T3:ty. exists T4:ty.
     typeshift_general T1 N C T3 ∧
     typeshift_general T2 N C T4
    ),
    (exists T5:ty. exists T6:ty.
     typeshift_general T1 N C T5 ∧
     typeshift_general T2 N C T6
    )
    |-
    tyfun(T3,T4)=tyfun(T5,T6)
```
The point 3 can be split into two subgoals:
```
3.1) typeshift_general T1 N C T3,
     typeshift_general T2 N C T4,
     typeshift_general T1 N C T5,
     typeshift_general T2 N C T6,
     |-
     T3=T5
3.2) typeshift_general T1 N C T3,
     typeshift_general T2 N C T4,
     typeshift_general T1 N C T5,
     typeshift_general T2 N C T6,
     |-
     T4=T6
```
4) Both nodes 3.1 and 3.2 can be discharged against node 1.

We are ready now to prove the first more sophisticated lemma (called *tshift_tshift_prop*) regarding shifting. For sake of simplicity we first write the lemma in an informal representation to give a clue of what it is intended to state and after that, we show how it is encoded with VCPT.

In the following representation *tshift* is a function that receives a cutoff and a type, and returns the shift of all the variable not bound from the cutoff n in that type.

The *tshift_tshift_prop* is an arithmetic property of shifting that shows how it is possible to obtain the same type from a double-shift, changing the order of the cutoffs between shiftings.

```
/* Lemma tshift_tshift_prop */

tshift n (tshift (n + n') T) = tshift (1 + (n + n')) (tshift n T)
```

Here it is the encoding of the lemma with VCPT:

```
/* Lemma tshift_tshift_prop with VCPT */
DECLARE N:nat, N':nat, T:ty, Z:nat, Z':nat, Tcurry:ty,
        Tcurry':ty, Tcurry'':ty, Tcurry''':ty IN
natPlus N N' Z,typeshift_general T 1 Z Tcurry,
typeshift_general Tcurry 1 N Tcurry',
typeshift_general T 1 N Tcurry'', natPlus 1 Z Z',
typeshift_general Tcurry'' 1 Z' Tcurry'''
```

```
|-
Tcurry'=Tcurry'''
END
```

Induction on types can be used to prove the validity of the lemma above.

5.2.3 Well-Formedness Lemmas

The subtyping judgments defined in 5.1.2 may only operate on types and contexts that are well formed. Intuitively, a type T is well-formed with respect to an environment E if all variables occurring in T are defined in E. More precisely, if T contains a type variable tyvar(i) (Section 5.1.1), then the i th element of E, starting counting from the right side, must exist and have the form *tvarb U*. According to the definition of binding (Section 5.1.1) *tvarb* denotes a type variable binding. Thus, we are imposing that each free variable inside T is bound in the environment E by a type variable binding.

The first lemma presented is named *wf_type_weaken* and intuitively states that: for any kind of types *T1,T2* and for any index X happens that when the predicate *get_tvar* on *E'* is false and implies that the same predicate on E is false, then if a type is well formed in E it has to be well formed in *E'* as well.

The proof proceeds by induction on the type T.

```
/* Wf_type_weaken Lemma */
DECLARE E:env, E':env, T:ty IN
(forall X:nat.((forall T1:ty. not(get_tvar E' X T1))=>
               (forall T2:ty. not(get_tvar E X T2))))
|-
(wf_type E T => wf_type E' T)
END
```

After the *wf_type_weaken* lemma has been proven, it is possible to show another important property (*wf_type_extensionality*) is valid.

The *wf_type_extensionality* property is quite similar to *wf_type_weaken* and it can be proven in similar fashion, combining the result got from *wf_type_weaken* lemma with induction on the type T.

```
/* Wf_type_extensionality */
DECLARE E:env, E':env, T:ty IN
(forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1)
|-
(wf_type E T => wf_type E' T)
END
```

Next lemmas are simple properties of environments before and after the insertion. This time the induction principle will work on the index X of the inserting function and the shifting property *tshift_tshift_prop* (presented in section 5.2.2) will be used to show their validity.

```
/* Get_bound_insert_bound_ge Lemma */
DECLARE X':nat, X:nat, E:env, E':env, T1:ty, T2:ty, T3:ty IN
insert_bound X' E E', natLessEq X' X ,
get_tvar E X T2 , typeshift_general T2 1 X' T3
```

```
|-
get_tvar E' X+1 T3
END


/* Get_bound_insert_bound_lt */
DECLARE X':nat, X:nat, E:env, E':env, T1:ty, T2:ty, T3:ty IN
insert_bound X' E E', natLess X X' ,
get_tvar E X T2 , typeshift_general T2 1 X' T3
|-
get_tvar E' X T3
END
```

Thanks to the last two lemmas we can claim that inserting a new *type variable binding*, considering respectively E and E' the environment before and after the insertion, preserves two important properties:

1. All the types that were well-formed in E are well-formed in E'.
2. If the environment E is well-formed then the new environment E' is also well-formed.

```
/* Insert_bound_wf_type Lemma */
DECLARE  E:env, E':env, T:ty, T1:ty, X:nat IN
insert_bound X E E', wf_type E T, typeshift_general T 1 X T1
|-
wf_type E' T1
END


/* Insert_bound_wf_env Lemma */
DECLARE X:nat, E:env, E':env IN
insert_bound X E E' , wf_env E
|-
wf_env E'
END
```

As last property we show that a type well-formed in an environment starting with any type variable binding, remains well formed in another environment that differs from the first only for its first type variable binding. The validity of this lemma is proven using the *wf_type_weaken* lemma and induction on the type T.

```
/* Lemma wf_type_ebound */
LEMMA lemma_wf_type_ebound
DECLARE E:env, T:ty, U:ty, V:ty IN
wf_type [tvarb(U)|E] T
|-
wf_type [tvarb(V)|E] T
END
```

5.2.4 Subtyping and Narrowing Lemmas

We start with simple well-formedness properties of types and environment involved in the subtyping judgment.

Here it is the first one (*sub_wf* lemma) that can be proved by induction on the structure of U.

```
/* Sub_wf Lemma */
DECLARE E:env, T:ty, U:ty IN
subt_rel E T U
|-
wf_env E ∧ wf_type E T ∧ wf_type E U
END
```

The next lemma is what has been called *A.1 Lemma* on the paper [ABF$^+$05] and as usual it can be proved by induction on type's structure of T.

```
/* Sub_reflexivity Lemma */
DECLARE E:env, T:ty IN
wf_env E , wf_type E T
|-
subt_rel E T T
END
```

The weakening lemma for subtyping relation (A.3 Lemma from [ABF$^+$05]) can be proved with nested inductions on both types T, T' and the natural X.

```
/* Sub_weakening_bound Lemma */
DECLARE X:nat. E:env. E':env. T:ty. T':ty.
        U:ty. V:ty. IN
insert_bound X E E', subt_rel E U V
typeshift_general U 1 X T', typeshift_general V 1 X T''
|-
subt_rel E' T' T''
END
```

Finally properties regarding the narrowing relation are proved in order to complete our lemma-infrastructure. More precisely if the narrowing relation holds for two environments *E,E'* then:

1. Any type that is well-formed in the first environment remains well-formed in the second.
2. If environment E is well-formed then E' is also well-formed.

Both proofs work using the definition of narrowing relation and induction on the natural number X.

```
/* Narrow_wf_type Lemma */
DECLARE X:nat, E:env, E':env, T:ty IN
narrow_rel X E E', wf_type E T
|-
wf_type E' T
END
```

```
/* Narrow_wf_env Lemma */
DECLARE X:nat, E:env, E':env IN
narrow_rel X E E' , wf_env E
|-
wf_env E'
END
```

5.2.5 Main Proof Lemmas

The main proof of the challenge consists in proving the *transitivity of subtyping* simultaneously with the *narrowing property*. In *Appendix A* of paper [ABF$^+$05] the two properties are proved simultaneously, by induction on the type's structure. Thus, at each stage of induction it is proven that the transitivity property is valid for the type T, and after that also the narrowing property holds for the same type T.

In this work, however, a different approach similar to the solution of the challenge presented by *Jerome Vouillon* in 2005 has been adopted. We used induction on type size and not on type structure so the presentation is slightly different from the paper proof. We decided to use that approach because in the paper proof the induction proceeds considering terms and types up to alpha conversions. In our case, we are using the *de Bruijn index* representation so we did not prove any lemmas regarding alpha conversions.

In order to reach the intended goal the proof has been split into three parts:

1. We show that transitivity holds if both transitivity and narrowing hold for a "smaller cut"type.
2. We prove that narrowing holds if we assume transitivity for types of same size.
3. We prove both transitivity and narrowing together using the previous results.

More precisely, as first, the validity of the following lemma is shown:

```
/* Transitivity_case Lemma */
DECLARE Q:ty IN
|- (forall Q':ty.
    (exists R1:nat. exists R2:nat. size Q R1 ∧ size Q' R2 ∧ natLess R2 R1)
      =>
        (transitivity_prop Q' ∧ narrowing_prop Q'))
                  => transitivity_prop Q
END
```

Although the proof of the above lemma is quite tedious, long and includes a big number of details and cases that have to be taken into account, it goes through a simple and clear proof-strategy.

Using induction on the type Q we show that *transitivity* holds for *top* and any *tvar(X)* variable(it easily comes from the definition of the transitivity proposition) and later using the assumptions that transitivity and narrowing are both valid for a smaller cut of Q we prove that transitivity holds either when Q is a type function or the universal type. Intuitively when Q is either the universal type(*tyall(T1,T2)*) or a type function(*tyfun(T1,T2)*), it must be the case that the two types *T1* and *T2* stay related to each other (with respect to the definition of the subtyping relation). Furthermore we are sure that the size of *T1* and *T2* is smaller than the size of Q, so we can now use the assumption to show that the property holds and to complete the proof.

The next step consists in proving the *narrowing_case lemma* for types of the same size:

```
/* Narrowing_case Lemma */
DECLARE Q:ty IN
|- (forall Q':ty. forall R1:nat.
     (size Q R1 ∧ size Q' R1) => transitivity_prop Q')
        => narrowing_prop Q
END
```

As for the latter proof, the amount of details and cases we have to deal with is quite heavy but the strategy that we used is simple.

The *narrowing proposition* informally states that if two environments *E,E'* are in a narrowing relation, then if one type T is subtype of another type S in E, T is also subtype of S in *E'* (See the definition of the *narrowing proposition* in 5.1.1). Thus the proof proceeds using a nested induction: first on the structure of the type Q and then on the structure of the types in the subtyping relation. Like in the *transitivity_case lemma* the first cases (when Q is either *Top* or *tvar(X)*) are simple and come straight from the definition of the *narrowing relation* and *subtyping relation* while the other cases (*tyfun(T1,T2)* and *tyall(T1,T2)*) require more sophisticated reasoning and use of the assumption in order to be proved (For any further information on the proof see the source code).

In the end, the last two lemmas needed to conclude the proof of the first point of POPLMark Challenge are : *transitivity_types_same_size* and *trans_narrow_ind_n*.

```
/* Transitivity_types_same_size */
DECLARE R:nat, Q:ty, Q':ty IN
size Q' R, size Q R  |- transitivity_prop Q'
END
```

```
/* Trans_narrow_ind_n */
DECLARE N:nat. Q:ty. R:nat IN
|- (size Q R ∧ natLess R N)
      =>
      (transitivity_prop Q ∧ narrowing_prop Q)
END
```

The first lemma (*Transitivity_types_same_size*) states that under the assumption that two types have the same size, we can show that the *transitivity proposition* holds for one of them. *Transitivity_types_same_size* can be proved with help of lemma *transitivity_case* using induction on the structure of *Q'*.

The *trans_narrow_ind_n* states that if a general type has limited size, then we can prove both *transitivity_proposition* and *narrowing_proposition*. The *trans_narrow_ind_n* is the most important lemma of our theory and it allows us to solve the first point of the challenge. The proof is quite intricate and needs almost all the lemmas previously introduced, so we show in detail how the *trans_narrow_ind_n* property is proved and how the tool helps in reaching that goal.

Proof Goal:

```
1)(size Q R ∧ natLess R N)
  |-
  (transitivity_prop Q ∧ narrowing_prop Q)
```

Induction on N : N=0 ∨ N=N'+1.
The base case is trivially true so we focus on N=N'+1.

```
2)(size Q R ∧ natLess R N'+1)
  |-
  (transitivity_prop Q ∧ narrowing_prop Q)
```

With a simple lemma of natural numbers we can show that:
If natLess R N'+1 then natLess R+1 N'+1 ∨ R=N'

```
3.1)(size Q R ∧ natLess R+1 N+1)
    |-
    (transitivity_prop Q ∧ narrowing_prop Q)
3.2)(size Q R ∧ R=N)
    |-
    (transitivity_prop Q ∧ narrowing_prop Q)
```

The node 3.1 can be discharged against node 1 due
to the VCPT discharge condition.
It remains:

```
4.1)(size Q N)
    |-
    transitivity_prop Q
4.2)(size Q N)
    |-
    narrowing_prop Q
```

Node 4.1 can be proved introducing the lemma
transitivity_types_same_size previously proved in this section.

To prove node 4.2, instead, the narrowing case lemma previously
proved in this section is used. Choosing for both types
T and T' of the lemma the type Q of node 4.2:

```
/* Narrowing_case Lemma */
(size T R1 ∧ size T' R1) => transitivity_prop T')
|-
narrowing_prop T
```

In this way we have that:

```
4.2)(size Q R ∧ size Q R) => transitivity_prop Q)
    =>
    narrowing_prop Q
    |-
    narrowing_prop Q
```

Node 4.2 now is trivially true because we proved the
transitivity of Q in node 4.1 so we have all the premises
to prove that narrow_prop Q holds too.

It is possible now to show the validity of the main goal:

```
/* Transitivity_narrowing Lemma */
DECLARE Q:ty IN
|- transitivity_prop Q /\ narrowing_prop Q
END
```

Once it has been proved lemma *trans_narrow_ind_n*, showing that transitivity and narrowing propositions hold for any kind of type is a simple game.

In fact, all what we have to do, is to show that for any type Q of size R, exists always a natural number N that is strictly greater than R. We reach the above goal simply using

induction on natural numbers.

5.3 Review on the other solutions

The aim of this section is to give a general idea on the proof-strategy used in other solutions of the *POPLMark Challenge*. We do not explore all the submitted solutions published on the *PoplMark Website* [Com] in detail but we focus just on some of them. Strategies used are compared, trying to identify advantages and drawbacks of each solution.

In order to do not make the section too verbose it is discussed only one solution for each possible encoding technique; specifically:

- Jerome Vouillon's solution with *de Bruijn Indices*.
- Xavier Leroy's solution with *Locally Nameless Representation*.
- Urban et al's solution with *Nominal Representation*.

5.3.1 Jerome Vouillon's solution with *de Bruijn Indices*

The proof strategy introduced in the last section 5.2 was inspired and follows the solution presented by Jerome Vouillon. The main differences between the two formalizations lie in the different theorem provers used(*Coq* for Vouillon and *VCPT* in this case) and consequently in the tactics and rules provided by each tool. Thus although the lemma-infrastructure built to reach the intended goal is almost the same, the two solutions present substantial differences due to the theorem provers' discharge strategy.

In order to show the differences between the tools' proof-strategies, we present the proof of one useful lemma for our theory (*wf_type_extensionality*) using both theorem provers, first *Coq* (from Vouillon's solution) and then *VCPT*.

Some clarifications are pointed out:

- The *Coq* function *get_bound e X* returns, if it exists, the type variable binding at position *X* in the environment *e*. The same function is encoded in *VCPT* with the predicate *get_tvar E X T1* (discussed in 5.1.2) where *T1*, as usual, is intended to be the result of the operation.
- The *Coq* function *wf_typ e T* tells if a type is either well-formed or not in the environment *e*. In *VCPT* it becomes *wf_type E T* (see 5.1.2).
- In order to prove the *wf_type_extensionality* lemma, Vouillon needed another lemma called *wf_typ_env_weaken*. The lemma *wf_typ_env_weaken* is supposed valid and already proved. Lemma *wf_type_env_weaken* is the following:

```
/* Wf_typ_env_weaken Lemma */
forall (T : typ) (e e' : env),
(forall (X : nat), get_bound e' X = None -> get_bound e X = None) ->
wf_typ e T -> wf_typ e' T
```

The proof of *wf_type_extensionality* lemma with *Coq* and *VCPT* is the following.

```
/* Coq Proof */
```
The lemma we want to prove is:

```
1) |- forall (T:typ) (e,e':env)
      (forall X:nat, get_bound e X = get_bound e' X) =>
      (wf_typ e T => wf_typ e' T)
```

After introducing the variables and splitting the implications:

```
2) T:typ
   e:env
   e':env
   H1: (forall X:nat, get_bound e X = get_bound e' X),
   H2: wf_typ e T
       |-
       wf_typ e' T
```

At this point Vouillon introduces the wf_typ_env_weaken lemma,
using the hypotesis H2.
The new goal then as the following shape:

```
3) T:typ
   e:env
   e':env
   H1: (forall X:nat, get_bound e X = get_bound e' X),
   H2: wf_typ e T
   |-
   forall X:nat. get_bound e' X=NONE => get_bound e X=NONE
```

Choosing a fresh variable for X and splitting the implication on
the right the node becomes:

```
4) T:typ
   e:env
   e':env
   H1: (forall X:nat, get_bound e X = get_bound e' X),
   H2: wf_typ e T
   n:nat
   H3: get_bound e' n = NONE
   |-
   get_bound e n = NONE
```

Rewriting now H1 and using H3, the proof is complete.

```
/* VCPT Proof */
```
Lemma wf_typ_extensionaly is proved
in a totally different way with VCPT.
First of all we didn't use any other lemma, second
we worked with induction on the structure of T.
Let's see how the proof evolves:

```
1) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
   wf_type E T
```

```
   |-
   wf_type E' T
```

As we said previously we work with induction on T so we have
four subgoals reflecting the four different types T can be:

```
/* T=tyvar(Y) */
2.1) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
      wf_type E tyvar(Y)
      |-
      wf_type E' tyvar(Y)

/* T=Top() */
2.2) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
      wf_type E top()
      |-
      wf_type E' top()

/* T=tyvfun(T1,T2) */
2.3) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
      wf_type E tyfun(T1,T2)
      |-
      wf_type E' tyfun(T1,T2)

/* T=tyall(T1,T2) */
2.4) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
      wf_type E tyall(T1,T2)
      |-
      wf_type E' tyall(T1,T2)
```

From the definition of predicate wf_type (see 5.1.2),
the first goal 2.1 is equal to:

```
2.1) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
      exists T:ty. get_tvar E Y T
      |-
      exists T:ty. get_tvar E' Y T
```

Node 2.1 is valid thanks to the first hypothesis.
Node 2.2 is valid because the type Top() is always well-formed.
We show now how to prove the validity of node 2.3, but not of node 2.4
because they are proved in the same fashion.
We can unfold the two definitions of the predicate wf_type (see 5.1.2)
on left and right side of turnstile obtaining:

```
2.3) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
      wf_type E T1, wf_type E T2
      |-
      wf_type E' T1 ∧ wf_type E T2
```

Splitting the 'and' right side we produce two subgoals:

```
2.3.1) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
       wf_type E T1, wf_type E T2
       |-
       wf_type E' T1
```

```
2.3.2) (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1),
       wf_type E T1, wf_type E T2
       |-
       wf_type E T2
```

Finally here, the tool comes in help and we can discharge both nodes against node number 1.

It is clear from the last example presented, how the proof-strategies used to solve the challenge, are deeply different as different is the help that the two tools give as assistant.

Personal experience, however, has shown how the use of VCPT to deal with interesting real cases of theorem proving is not that tricky as it could appear as first thought. In fact, comparing the two solutions previously presented, it is clear that the overhead introduced by using a first order logic is completely acceptable and that the global well-founded condition of VCPT allow users to deal with any kind of problems.

5.3.2 Xavier Leroy's solution with *Locally Nameless Representation*

Xavier Leroy solved the *POPLMark Challenge* using the *Locally Nameless Representation*. In chapter 3 we introduced the *Locally Nameless Representation* as combination of *de Bruijn Indices* and *Nominal* approach. Thus free variables are encoded with names and bound variables with *de Bruijn Indices*.

The first main difference between the two formalizations is that dealing with names the author had to implement all the functions to handle *α-conversions* (so equality on names, creation of fresh names and so on). However it has been shown that once this little obstacle is overcome the *Locally Nameless Representation* is the best way to deal with the challenge.

It combines the advantages of both the other strategies :

- The simplicity of the *Nominal representation*.
- The machine-oriented coding of *de Bruijn Indices*.

Furthermore using the indices only for bound variables we get the rid of all the shifting operations over terms and types that are the main drawback of the pure *de Bruijn Indices* representation.

After these clarifications we notice that the proof strategy proposed by the author is quite similar to ours except for the last proof of the main goal. Xavier Leroy first shows the validity of some important properties for the well-formedness of environments (for example "swapping types inside environments keep the environment well formed"). He proved properties regarding subtyping relation (for example "if two types are in subtyping relation each other in an environment then both types have to be well formed in that environment") and finally he proceeds with the proof of the final goal.

Whereas we proved the *transitivity of subtyping* and *narrowing* propositions with induction on the type's size, Leroy used induction on the structure of type. He followed then the proof strategy presented in the description of the challenge [ABF+05]

When we started our project, we took into account the possibility of solving the challenge using the *Locally Nameless Representation* since the results of last projects showed how such a representation is the most suitable and simple to deal with the *POPLMark Challenge*. Nevertheless we chose to follow a different route due to the low level of the theorem prover we used,

being aware that the difficulties we could face dealing with names and α-*conversions* could make the work harder than encoding the shifting operations on types and terms.

However, that was just an implementative choice, because previous works with VCPT showed how the use of names and the problems that arise with α-conversions can be handled in VCPT. *Erik Angelin* in his master thesis encoded the π-calculus in VCPT, including the functions to manage α-conversion.

5.4 Urban et al's solution with *Nominal Representation*

The approach of solving the *POPLMark Challenge* with *Nominal Representation* is clearly attractive, at least because it is the closest to usual mathematical practice, leading to statements and proofs that are very close to what it is accustomed to see in textbooks and research papers.

Nevertheless this approach requires to integrate somehow in the logic sophisticated mechanisms to deal with α-*conversions* and avoid capturing variables.

Despite the recent achievements in theorem proving, support for nominal logic within a proof assistant is still in its infancy, but it is making significant progress (See [A.M03] and [CU05] for more information).

Urban et al developed a package for Isabelle in order to overcome the well known problems arise with the *Nominal Representation* like dealing with binders, renaming bound variables and capture avoiding substitution. Their package is called *Nominal Isabelle* and it aims to make such proofs easy to formalize, providing an infrastructure for declaring nominal datatypes (that is alpha-equivalence classes) and for defining functions over them by structural recursion.

Thus *Nominal Isabelle* is well designed to deal with the *POPLMark Challenge* using the *Nominal Representation*. The proof strategy followed by the authors still trace out the usual way to solve the challenge we discussed more than once in the last sections:

- They prove crucial properties about well-formedness of environments.
- They make some reasoning about subtyping relation and properties related.
- They prove the *transitivity of subtyping* and *narrowing* proposition with induction on the type's structure.

We have to notice that in order to keep their theory sound *capture-avoiding substitutions for type's* mechanisms are needed. However using the *Nominal Isabelle* package proving such properties is easy and intuitive.

The final result is a good and clear solution of the challenge that trace out the proof presented in the paper.

Chapter 6

Conclusions and Future Works

This thesis has presented a solution to the first part of *POPLMark Challenge* using the VeriCode Proof Tool (VCPT). In this chapter we draw some considerations about the thesis and we give suggestions for possible future works.

6.1 Conclusions

POPLMark is a real challenge posed in 2005 by a community of researchers with the intent of measuring progresses in machine checked proofs [ABF+05]. Since 2005 the interest in solving the challenge has grown constantly. People have worked and presented their own solutions to POPLMark, due to needs of mechanizing and verifying code before the release. Thus, nowadays the challenge has been completely covered in all its aspects and it has been shown that it is possible to use theorem proving for formalizing and proving some important aspects of most programming languages.

At the beginning of our project, we already knew that what we were about to deal with, was something provable. The goal of this thesis, then, was to convince people that it is possible to use first order logic (the logic embedded in VCPT) instead of more powerful logics for theorem proving issues. At the same time, we wanted to show that the efforts in adopting first order logic are comparable with the complexity of solutions developed with other logics.

Now that the project is accomplished we have answers for those questions. First of all, since the challenge has been solved with VCPT, we can state that the tool is ready to face real case studies. Furthermore, since this thesis draws on Vouillon's solution (as discussed in Section 5.3), it is easy to notice how the complexity of the two solutions does not differ a lot. Once it has been overcome some initial problems in understanding how to express properties with first order logic, the rest of the project goes straightforwardly on in the same fashion of other solutions. The difficulties of the challenge are not related to the specific tool used but rather to inherit toughness that theorem proving carries on.

Personal experience has shown that VCPT is actually competitive and ready for assisting users in proving properties of real systems. The tool can be used as other common theorem provers like (Isabelle [Pau94b], Coq [BC04] or PVS [ORR+96]) without introducing further complications.

6.2 Future Works

In this section some possible directions for future works are outlined.

6.2.1 Solving other parts of the challenge

The most natural way of thinking about a possible extension of this thesis is to cover the other parts of the challenge starting from our work. As it has been already mentioned in previous chapters (Chapter 2 and Chapter 5) we addressed part 1.a. Parts 1.b, 2 and 3 have not been considered.

However, in the beginning, we did not know yet how many parts of the challenge we would have solved. For this reason we encoded some datatypes and functions that are not strictly related to part 1.a. Those functions can be considered as starting point for showing the rest of the POPLMark challenge using VCPT. Appendix C shows in detail those functions giving a short description for each of them.

6.2.2 Improving and extending VCPT

When it comes to the proof assistant, it seems clear that it is not currently in wide use, so there has been no real drive to provide practical usage support. However, some suggestions are presented here.

Documentation

The documentation would require some updates. Furthermore, a tutorial on the proof assistant would be very helpful for users. The answers of your questions can be found for the majority of the time only in the source code.

A tool for tactics

Writing tactics is one of the central issue of all VCPT theories. If users could have an automatic support for generating complex definitions lot of errors would be avoided and time would be saved.

General Comments about the tool

It has been noticed that the declaration of lemmas does not allow users to refer to ordinals. Since VCPT widely uses ordinals for variables and approximations, it could be helpful referring to them while declaring lemmas. This would make proofs more concise.

Another important improvement might be to enrich the tool introducing new SML functions to handle complex structures. For example, specifying general operations on lists could make the use of the tool more user friendly and intuitive. Furthermore, a more advanced typing system for the logic where predicates could have some simple polymorphism in the definition, would make the use of lists more interesting and usable for defining general operations.

These suggestions are based only on practical needs encountered during the project. The impact and the time required to augment the power of the tool has not been evaluated.

Appendix A

Appendix A

This Appendix shows two important properties (Lemma 2.1.2 and Lemma 2.1.3) of the shifting and substitution functions.

Let c, d, x be natural numbers, let t, s be generic nameless λ-terms. Lemma 2.1.2 states that the result of the d-place shift from cutoff c of the term obtained by the substitution of x with s in t, is the same result obtained by first d-shifting from cutoff c all the terms x, s, t and after computing the substitution.

Lemma 2.1.2 $\uparrow_c^d ([s/x]t) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (t)$

Let t be a λ-term, let Γ be an environment and let $rm_\Gamma(t)$ be the function that takes a λ-term, t, and returns the nameless representation of t.

Lemma 2.1.3 $rm_\Gamma([s/x]t) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]rm_\Gamma(t)$

Functions

Here there are listed functions necessary to prove Lemma 2.1.2 and Lemma 2.1.3.

Let k, c, d be natural numbers, let s, t_1, t_2 be λ-terms expressed with the de Bruijn representation, then a d-place shift of term t from cutoff c is defined recursively as in Table A.1:

$$\begin{array}{lcl} \uparrow_c^d (k) & = & \begin{cases} k & \text{if } k < c \\ k + d & \text{if } k \geq c \end{cases} \\ \uparrow_c^d (\lambda.t_1) & = & \lambda. \uparrow_{c+1}^d (t_1) \\ \uparrow_c^d (t_1\ t_2) & = & \uparrow_c^d (t_2) \uparrow_c^d (t_2) \end{array}$$

Table A.1: Shifting function with de Bruijn representation

Let x, s, y be λ-variables and let t_1, t_2 be λ-terms. The substitution function in λ-calculus is defined as in Table A.2.

$$
\begin{array}{lcll}
x[s/x] & = & s & \\
y[s/x] & = & y & \text{if } x \neq y \\
(\lambda y.t_1)[s/x] & = & \begin{cases} \lambda y.t_1 & \text{if } x = y \\ \lambda y.(t_1[s/x]) & \text{if } x \neq y \text{ and } y \notin FV(s) \end{cases} & \\
(t_1 t_2)[s/x] & = & (t_1[s/x])(t_2[s/x]) &
\end{array}
$$

Table A.2: Substitution function in lambda-calculus

Let k, j be natural numbers, let s, t_1, t_2 be λ nameless terms, then the substitution function with de Bruijn indices is represented as in Table A.3:

$$
\begin{array}{lcl}
k[s/j] & = & \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases} \\
(\lambda.t_1)[s/j] & = & \lambda.(t_1[\uparrow^1 (s)/j+1]) \\
(t_1\ t_2)[s/j] & = & (t_1[s/j])(t_2[s/j])
\end{array}
$$

Table A.3: Substitution function with de Bruijn representation

Let t be a λ-term and let Γ be an environment, we declare a function, $rm_\Gamma(t)$, that takes t and returns the nameless representation of t with respect of the environment Γ.

$$
\begin{array}{lcl}
rm_\Gamma(x) & = & \text{Index of the rightmost } x \text{ in } \Gamma. \\
rm_\Gamma(t_1 t_2) & = & rm_\Gamma(t_1) rm_\Gamma(t_2). \\
rm_\Gamma(\lambda y.t_1) & = & \lambda.rm_{\Gamma,y}(t_1).
\end{array}
$$

Table A.4: Remove Names Function

Proof of Lemma 2.1.2

In order to prove Lemma 2.1.2 it is necessary to prove Lemma A.0.1. Let k, i be natural numbers, and let t be a λ-term expressed with the de Bruijn representation. Suppose it is necessary to shift by one the term t from cutoff i and after, the result of such shifting has to be shifted by one from cutoff $k+1$. Lemma A.0.1 states that under the assumption that $i < k+1$ it is possible to invert the two shifting, computing first a shift by one of term t from cutoff k, and after shifting the result by one from cutoff i, obtaining the same result.

Lemma A.0.1 $\uparrow^1_{k+1} (\uparrow^1_i (t)) \equiv \uparrow^1_i (\uparrow^1_k (t))$
Under the assumption that $i < k+1$

Lemma A.0.1 is shown by induction on the structure of t. In Table A.5, x, i, k, w, z are natural numbers and t, t_1, t_2, s are general nameless λ-terms.

Case $t = x$:	
Case $x < i < k+1$:	$\uparrow^1_{k+1} (\uparrow^1_i (x)) \equiv \uparrow^1_i (\uparrow^1_k (x))$ $\uparrow^1_{k+1} (x) \equiv \uparrow^1_i (x)$ $x \equiv x$
Case $x = i = k$:	$\uparrow^1_{k+1} (\uparrow^1_i (x)) \equiv \uparrow^1_i (\uparrow^1_k (x))$ $\uparrow^1_{k+1} (x+1) \equiv \uparrow^1_i (x+1)$ $x+1+1 \equiv x+1+1$
Case $x = i < k$:	$\uparrow^1_{k+1} (\uparrow^1_i (x)) \equiv \uparrow^1_i (\uparrow^1_k (x))$ $\uparrow^1_{k+1} (x+1) \equiv \uparrow^1_i (x)$ $x+1 \equiv x+1$
Case $i < x < k$:	$\uparrow^1_{k+1} (\uparrow^1_i (x)) \equiv \uparrow^1_i (\uparrow^1_k (x))$ $\uparrow^1_{k+1} (x+1) \equiv \uparrow^1_i (x)$ $x+1 \equiv x+1$
Case $x > i, x \geq k$:	$\uparrow^1_{k+1} (\uparrow^1_i (x)) \equiv \uparrow^1_i (\uparrow^1_k (x))$ $\uparrow^1_{k+1} (x+1) \equiv \uparrow^1_i (x+1)$ $x+1+1 \equiv x+1+1$
Base case valid. Induction Hypothesis: $\uparrow^1_{k+1} (\uparrow^1_i (t)) \equiv \uparrow^1_i (\uparrow^1_k (t))$ **if** $i < k+1$	
Case $x = t_1 t_2$:	$\uparrow^1_{k+1} (\uparrow^1_i (t_1 t_2)) \equiv \uparrow^1_i (\uparrow^1_k (t_1 t_2))$ $\uparrow^1_{k+1} (\uparrow^1_i (t_1)\ \uparrow^1_i (t_2)) \equiv \uparrow^1_i (\uparrow^1_k (t_1)\ \uparrow^1_k (t_2))$ $(\uparrow^1_{k+1} (\uparrow^1_i (t_1)))(\uparrow^1_{k+1} (\uparrow^1_i (t_2))) \equiv (\uparrow^1_i (\uparrow^1_k (t_1)))(\uparrow^1_i (\uparrow^1_k (t_2)))$
Due to the induction hypothesis the terms of the equivalence above are equal	
Case $x = \lambda.t_1$:	$\uparrow^1_{k+1} (\uparrow^1_i (\lambda.t_1)) \equiv \uparrow^1_i (\uparrow^1_k (\lambda.t_1))$ $\uparrow^1_{k+1} (\lambda.\ \uparrow^1_{i+1} (t_1)) \equiv \uparrow^1_i (\lambda.\ \uparrow^1_{k+1} (t_1))$ $\lambda.\ \uparrow^1_{(k+1)+1} (\uparrow^1_{i+1} (t_1)) \equiv \lambda.\ \uparrow^1_{i+1} (\uparrow^1_{k+1} (t_1))$
Notice that giving a new name for $k+1$ **for example** z **and a new name for** $i+1$ **for example** w **we have:**	
	$\lambda.\ \uparrow^1_{z+1} (\uparrow^1_w (t_1)) \equiv \lambda.\ \uparrow^1_w (\uparrow^1_z (t_1))$
Furthermore, it is still true that $w < z+1$ **because** $i+1 < k+1+1$ **if** $i < k+1$ **The last case of the lemma is also valid for induction hypothesis**	

Table A.5: Proof of Lemma A.0.1

Using Lemma A.0.1, it is possible to prove Lemma 2.1.2 that is the goal of this Appendix A.

The proof proceeds by induction on the term t:

Case $t = x$:	$\uparrow_c^d ([s/x](x)) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (x)$ $\uparrow_c^d (s) \equiv \uparrow_c^d (s)$
Case $t = y \wedge y \neq x$:	$\uparrow_c^d ([s/x]y) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (y)$ $\uparrow_c^d (y) \equiv \uparrow_c^d (y)$
Base Case True. Induction Hypothesis:	$\uparrow_c^d ([s/x]t) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (t)$
Case $t = t_1\ t_2$:	$\uparrow_c^d ([s/x](t_1\ t_2)) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (t_1\ t_2)$ $\uparrow_c^d (([s/x](t_1))([s/x](t_2))) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (t_1) \uparrow_c^d (t_2)$ $\uparrow_c^d [s/x](t_1) \uparrow_c^d [s/x](t_2) \equiv ([\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (t_1))([\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (t_2))$
With induction hypothesis the equivalence above is true	
Case $t = \lambda.t_1$:	$\uparrow_c^d ([s/x](\lambda.t_1)) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)] \uparrow_c^d (\lambda.t_1)$ $\uparrow_c^d (\lambda.[\uparrow_0^1 (s)/ \uparrow_0^1 (x)](t_1)) \equiv [\uparrow_c^d (s)/ \uparrow_c^d (x)]\lambda.\ \uparrow_{c+1}^d (t_1)$ $\lambda.\ \uparrow_{c+1}^d ([\uparrow_0^1 (s)/ \uparrow_0^1 (x)](t_1)) \equiv \lambda.[\uparrow_0^1 (\uparrow_c^d (s))/ \uparrow_0^1 (\uparrow_c^d (x))] \uparrow_{c+1}^d (t_1)$
Using now the induction hypothesis on the left side of the equivalence, and getting rid of the firs λ both sides we have	
	$[(\uparrow_{c+1}^d \uparrow_0^1 (s))/(\uparrow_{c+1}^d \uparrow_0^1 (x))] \uparrow_{c+1}^d (t_1) \equiv [\uparrow_0^1 (\uparrow_c^d (s))/ \uparrow_0^1 (\uparrow_c^d (x))] \uparrow_{c+1}^d (t_1)$
The equivalence above is true due to Lemma A.0.1.	
More specifically:	$(\uparrow_{c+1}^d \uparrow_0^1 (s)) \equiv \uparrow_0^1 (\uparrow_c^d (s))$ **due to Lemma A.0.1** $(\uparrow_{c+1}^d \uparrow_0^1 (x)) \equiv \uparrow_0^1 (\uparrow_c^d (x))$ **due to Lemma A.0.1** $(\uparrow_{c+1}^d (t_1)) \equiv (\uparrow_{c+1}^d (t_1))$

Table A.6: Proof of Lemma 2.1.2

Proof of Lemma 2.1.3

Lemma A.0.2 is introduced in order to prove Lemma 2.1.3. Let t be a λ-term, let y be a λ-variable and let Γ be an environment.

Lemma A.0.2 $rm_{\Gamma,y}(t) \equiv (\uparrow_0^1 rm_\Gamma(t))$

For sake of simplicity we do not present the proof of lemma A.0.2, but it can be proven by induction on the structure of term t.

Let x, y be named-λ-variables, let t, t_1, t_2, s be general λ-terms and let Γ be an environment. Table A.7 shows a proof for Lemma 2.1.3.

Case $t = x$:	$rm_\Gamma([s/x]x) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]rm_\Gamma(x)$
	$rm_\Gamma(s) \equiv rm_\Gamma(s)$
Case $t = z$ **and** $z \neq x$:	$rm_\Gamma([s/x]z) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]rm_\Gamma(z)$
	$rm_\Gamma(z) \equiv rm_\Gamma(z)$
Base case valid. Induction Hypothesis: $rm_\Gamma([s/x]t) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]rm_\Gamma(t)$	
Case $t = t_1 t_2$:	$rm_\Gamma([s/x]t_1t_2) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]rm_\Gamma(t_1t_2)$
	$rm_\Gamma([s/x]t_1[s/x]t_2) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]((rm_\Gamma(t_1))(rm_\Gamma(t_2)))$
$(rm_\Gamma([s/x]t_1))(rm_\Gamma([s/x]t_1)) \equiv [rm_\Gamma(s)/rm_\Gamma(x)](rm_\Gamma(t_1))[rm_\Gamma(s)/rm_\Gamma(x)](rm_\Gamma(t_2))$	
With induction hypothesis the equivalence above is true	
Case $t = \lambda y.t_1$:	$rm_\Gamma([s/x]\lambda y.t_1) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]rm_\Gamma(\lambda y.t_1)$
	$rm_\Gamma(\lambda y.[s/x]t_1) \equiv [rm_\Gamma(s)/rm_\Gamma(x)]\lambda.rm_{\Gamma,y}(t_1)$
	$\lambda.rm_{\Gamma,y}([s/x]t_1) \equiv \lambda.[\uparrow_0^1 (rm_\Gamma(s))/ \uparrow_0^1 (rm_\Gamma(x))]rm_{\Gamma,y}(t_1)$
Applying the induction hypothesis on the left-side term:	
$[rm_{\Gamma,y}(s)/rm_{\Gamma,y}(x)]rm_{\Gamma,y}(t_1) \equiv [\uparrow_0^1 (rm_\Gamma(s))/ \uparrow_0^1 (rm_\Gamma(x))]rm_{\Gamma,y}(t_1)$	
The proof is now complete due to lemma A.0.2	
More specifically:	$(rm_{\Gamma,y}(s)) \equiv (\uparrow_0^1 (rm_\Gamma(s))$ **due to Lemma A.0.2**
	$(rm_{\Gamma,y}(x)) \equiv (\uparrow_0^1 (rm_\Gamma(x))$ **due to Lemma A.0.2**
	$(rm_{\Gamma,y}(t_1)) \equiv (rm_{\Gamma,y}(t_1))$

Table A.7: Proof of Lemma 2.1.3

Appendix B

Appendix B

Appendix B, lists all the lemma proved in this thesis, in order to reach the main goal of the challenge. The lemmas are grouped into 5 categories, as discussed in Section 5.1.3.

Natural Number Lemmas

```
/* Lemma commutativity_of_sum_of_nats */
LEMMA commutativity_of_sum_of_nats
DECLARE X : nat, Y : nat, Z : nat IN
        natPlus X Y Z
        |-
        natPlus Y X Z
PROOF from "./Naturals_Lemmas/commutativity_of_sum_of_nats.sml"
END

/* Lemma lemma_natPlus_sub1 */
LEMMA lemma_natplus_sub1
DECLARE X:nat, Y:nat, Z:nat IN
        natPlus X Y Z
        |-
        natPlus  X Y+1 Z+1
PROOF from "./Naturals_Lemmas/lemma_natplus_sub1.sml"
END

/* Lemma lemma_natPlus_sub2 */
LEMMA lemma_natplus_sub2
DECLARE X:nat, Y:nat, Z:nat IN
        natPlus X+1 Y+1 Z+1
        |-
        natPlus  X Y+1 Z
PROOF from "./Naturals_Lemmas/lemma_natplus_sub2.sml"
END

/* Lemma lemma_natless_zero */
LEMMA lemma_natless_zero
DECLARE X : nat IN
        natLess X 0
        |-
PROOF from "./Naturals_Lemmas/lemma_natless_zero.sml"
END
```

```
/* Lemma lemma_zero_plus_x */
LEMMA lemma_zero_plus_x
DECLARE X : nat IN
        |-
        natPlus 0 X X
PROOF from "./Naturals_Lemmas/lemma_zero_plus_x.sml"
END

/* Lemma lemma_nat_eq_or_div */
LEMMA lemma_nat_eq_or_div
      |-
      forall X:nat. forall Y:nat. ((X=Y)∨(not(X=Y)))
PROOF from "./Naturals_Lemmas/lemma_nat_eq_or_div.sml"
END

/* Lemma lemma_prop_natless_1 */
LEMMA lemma_prop_natless_1
DECLARE X1 : nat, X2 : nat  IN
        natLess X1 X2+1
        |-
        (natLess X1+1 X2+1 ∨ X1=X2)
PROOF from "./Naturals_Lemmas/lemma_prop_natless_1.sml"
END

/* Lemma lemma_natplus_one_banal */
LEMMA lemma_natplus_one_banal
DECLARE X : nat, Y : nat  IN
        natPlus X 1 Y
        |-
        X+1=Y
PROOF from "./Naturals_Lemmas/lemma_natplus_one_banal.sml"
END

/* Lemma no_less_itself */
LEMMA lemma_no_less_itself
DECLARE X:nat IN
        natLess X X
        |-
PROOF from "./Naturals_Lemmas/lemma_no_less_itself.sml"
END

/* Lemma less_itself_plus_one */
LEMMA lemma_less_itself_plus_one
      |-
      forall X:nat. natLess X X+1
PROOF from "./Naturals_Lemmas/lemma_less_itself_plus_one.sml"
END

/* Lemma x_less_sum_x_y_1 */
LEMMA lemma_x_less_sum_x_y_1
DECLARE X:nat, Y:nat, Z:nat IN
        natPlus X Y Z
        |-
        natLess X Z+1
PROOF from "./Naturals_Lemmas/lemma_x_less_sum_x_y_1.sml"
```

```
END

/* Lemma y_less_sum_x_y_1 */
LEMMA lemma_y_less_sum_x_y_1
DECLARE X:nat, Y:nat, Z:nat IN
        natPlus X Y Z
        |-
        natLess Y Z+1
PROOF from "./Naturals_Lemmas/lemma_y_less_sum_x_y_1.sml"
END

/* Lemma equality_plus */
LEMMA equality_plus
DECLARE A : nat, B : nat, C : nat, D : nat IN
        natPlus D A B, natPlus D A C
        |-
        B = C
PROOF from "./Naturals_Lemmas/equality_plus.sml"
END

/* Lemma equation */
LEMMA equation
DECLARE A : nat, B : nat, C : nat, D : nat, E : nat IN
        natPlus A B C , natPlus D B E, natLess E C
        |-
        natLess D A
PROOF from "./Naturals_Lemmas/equation.sml"
END

/* Lemma equation1 */
LEMMA equation1
DECLARE A : nat, B : nat, C : nat, D : nat, E : nat IN
        natPlus A B C, natPlus D B E, natLess A D
        |-
        natLess C E
PROOF from "./Naturals_Lemmas/equation1.sml"
END

/* Lemma sum_three_terms2 */
LEMMA sum_three_terms2
DECLARE A : nat, B : nat, C : nat, D : nat, E : nat, F : nat IN
        natPlus A B C , natPlus D C E, natPlus D A F
        |-
        natPlus F B E
PROOF from "./Naturals_Lemmas/sum_three_terms2.sml"
END

/* Lemma property_of_sum1 */
LEMMA property_of_sum1
DECLARE A : nat, B : nat, C : nat, D : nat IN
        natPlus A B C , natLess D A
        |-
        natLess D C
PROOF from "./Naturals_Lemmas/property_of_sum1.sml"
END
```

```
/* Lemma property_of_sum2 */
LEMMA property_of_sum2
DECLARE A : nat, B : nat, C : nat, D : nat IN
        natPlus A B C , natLess D B
        |-
        natLess D C
PROOF from "./Naturals_Lemmas/property_of_sum2.sml"
END

/* Lemma property_of_sum3 */
LEMMA property_of_sum3
DECLARE A : nat, B : nat, C : nat, D : nat IN
        natPlus A B C , natLess C D
        |-
        natLess A D
PROOF from "./Naturals_Lemmas/property_of_sum3.sml"
END

/* Lemma equality double natPlus */
LEMMA lemma_eq_double_natplus
DECLARE A:nat, B:nat, C:nat, D:nat, E:nat, F:nat IN
        natPlus A B C, natPlus D E F
        |-
        ((A=D ∧ B=E) => C=F)
PROOF from "./Naturals_Lemmas/lemma_eq_double_natplus.sml"
END

/* Lemma lemma_natlesseq_x_y_or_natless_y_x */
LEMMA lemma_natlesseq_x_y_or_natless_y_x
      |-
      forall X:nat. forall Y:nat. (natLessEq X Y ∨ natLess Y X)
PROOF from "./Naturals_Lemmas/lemma_natlesseq_x_y_or_natless_y_x.sml"
END

/* Lemma lemma_eq_same_plus */
LEMMA lemma_eq_same_plus
DECLARE X : nat, Z : nat, Y :nat, K : nat  IN
        natPlus X Z Y, natPlus X Z K
        |-
        Y=K
PROOF from "./Naturals_Lemmas/lemma_eq_same_plus.sml"
END

/* Lemma zero_lessereq_nat_r */
LEMMA zero_lessereq_nat_r
DECLARE X : nat IN
        |-
        natLessEq 0 X
PROOF from "./Naturals_Lemmas/zero_lessereq_nat_r.sml"
END

/* Lemma  less_plus_one */
LEMMA less_plus_one
DECLARE A : nat, B : nat, C : nat, D : nat IN
        natLessEq A B , natPlus 1 A C, natPlus 1 B D
        |-
```

```
        natLessEq C D
PROOF from "./Naturals_Lemmas/less_plus_one.sml"
END

/* Lemma trans_less */
LEMMA trans_less
DECLARE A : nat, B : nat, C : nat IN
        natLessEq A B , natLess C A
        |-
        natLess C B
PROOF from "./Naturals_Lemmas/trans_less.sml"
END

/* Lemma lemma_trans_less_adding_positive1 */
LEMMA lemma_trans_less_adding_positive1
DECLARE X0 : nat, X1 : nat  IN
        natLess X0 X1
        |-
        natLess X0 X1+1
PROOF from "./Naturals_Lemmas/lemma_trans_less_adding_positive1.sml"
END

/* Lemma natplus_x_plus_one_x_plus_two */
Lemma natplus_x_plus_one_x_plus_two
DECLARE X:nat, Y:nat, Z:nat IN
        natPlus X 1 Y, natPlus X+1 1 Z
        |-
        Z=Y+1
PROOF from "./Naturals_Lemmas/natplus_x_plus_one_x_plus_two.sml"
END
```

Shifting Lemmas

```
/* Lemma lemma_tshift_tshift_prop1 */
LEMMA lemma_tshift_tshift_prop1
DECLARE N:nat, N':nat, T:ty, Z:nat, Z':nat, Tcurry:ty,
        Tcurry':ty, Tcurry'':ty, Tcurry''':ty IN
        natPlus N N' Z,typeshift_general T 1 Z Tcurry,
        typeshift_general Tcurry 1 N Tcurry', typeshift_general T 1 N Tcurry'',
        natPlus 1 Z Z',typeshift_general Tcurry'' 1 Z' Tcurry'''
        |-
        Tcurry'=Tcurry'''
PROOF from "./Shift_Subst_Lemmas/lemma_tshift_tshift_prop1.sml"
END

/* Lemma typeshift_shift_or_natless_indx_tyvar */
LEMMA lemma_typeshift_shift_or_natless_indx_tyvar
DECLARE X:nat, Y:nat, T:ty IN
        typeshift_general tyvar(X) 1 Y T
        |-
        (typeshift_general tyvar(X) 1 Y tyvar(X+1) ∨ natLess X Y)
PROOF from "./Shift_Subst_Lemmas/lemma_typeshift_shift_or_natless_indx_tyvar.sml"
END
```

```
/* Lemma typeshift_no_shift_tyvar */
LEMMA lemma_typeshift_no_shift_tyvar
DECLARE X:nat, Y:nat, T:ty IN
        typeshift_general tyvar(X) 1 Y T, natLess X Y
        |-
        T=tyvar(X)
PROOF from "./Shift_Subst_Lemmas/lemma_typeshift_no_shift_tyvar.sml"
END

/* Lemma lemma_same_typeshift_general */
LEMMA lemma_same_typeshift_general
DECLARE N:nat, C:nat, T:ty, Tcurry:ty, Tcurry':ty IN
        typeshift_general T N C Tcurry, typeshift_general T N C Tcurry'
        |-
        Tcurry = Tcurry'
PROOF from "./Shift_Subst_Lemmas/lemma_same_typeshift_general.sml"
END

/* Lemma same_typeshift_same_start */
LEMMA lemma_same_typeshift_same_start1
DECLARE T1:ty, T2:ty, T':ty, T'':ty, N:nat, X:nat IN
        typeshift_general T1 N X T', typeshift_general T2 N X T''
        |-
        ((T1=T2) => (T'=T''))
PROOF from "./Shift_Subst_Lemmas/lemma_same_typeshift_same_start1.sml"
END

/* Lemma ad_hoc_for_insert_bound_ge */
LEMMA lemma_ad_hoc_for_insert_bound_ge
DECLARE T1:ty, T2:ty, T3:ty, T4:ty, T5:ty, T6:ty, X:nat, Y:nat IN
        typeshift_general T1 1 X T2, typeshift_general T3 1 X T4,
typeshift_general T4 1 Y+1 T5
|-
(natLessEq X Y =>((exists T6:ty. typeshift_general T3 1 Y T6 ∧ T1=T6)=> T2=T5))
PROOF from "./Shift_Subst_Lemmas/lemma_ad_hoc_for_insert_bound_ge.sml"
END
```

Well-Formedness Lemmas

```
/* Lemma exists_wf_type_tyvar */
LEMMA lemma_exists_wf_type_tyvar
DECLARE E:env, X:nat, T:ty IN
        get_tvar E X T
        |-
        wf_type E tyvar(X)
PROOF from "./Well_Formedness_Lemmas/lemma_exists_wf_type_tyvar.sml"
END

/* Lemma wf_type_ebound */
LEMMA lemma_wf_type_ebound
DECLARE E:env, T:ty IN
        |-
```

```
        forall U:ty. forall V:ty.
        wf_type [tvarb(U)|E] T => wf_type [tvarb(V)|E] T
PROOF from "./Well_Formedness_Lemmas/lemma_wf_type_ebound.sml"
END

/* Lemma exists_get_bgn_varb_rhs */
LEMMA lemma_exists_get_bgn_varb_rhs
DECLARE E:env, T:ty, X:nat IN
        exists T1:ty. get_tvar E X T1
        |-
        exists T2:ty. get_tvar [varb(T)|E] X T2
PROOF from "./Well_Formedness_Lemmas/lemma_exists_get_bgn_varb_rhs.sml"
END

/* Lemma get_tvar_same_env_same_indx */
LEMMA lemma_get_tvar_same_env_same_indx
DECLARE E : env, X:nat, T1:ty, T2:ty IN
        get_tvar E X T1, get_tvar E X T2
        |-
        T1 = T2
PROOF from "./Well_Formedness_Lemmas/lemma_get_tvar_same_env_same_indx.sml"
END

/* Lemma lemma_get_tvar_remove_tvar_ad_hoc_for_narrow */
LEMMA lemma_get_tvar_remove_tvar_ad_hoc_for_narrow
DECLARE E:env, T1:ty, U:ty, X:nat IN
        exists T:ty. get_tvar E X T
        |-
        exists T1:ty. get_tvar [tvarb(U)|E] X+1 T1
PROOF from "./Well_Formedness_Lemmas/lemma_get_tvar_remove_tvar_ad_hoc_for_narrow.sml"
END

/* Lemma get_tvar_remove_tvar */
LEMMA lemma_get_tvar_remove_tvar
DECLARE E:env, T:ty, T1:ty, U:ty, X:nat IN
        get_tvar E X T
        |-
        exists T1:ty. get_tvar [tvarb(U)|E] X+1 T1
PROOF from "./Well_Formedness_Lemmas/lemma_get_tvar_remove_tvar.sml"
END

/*LEMMA forall_not_gett_add_tvar_for_variant */
LEMMA lemma_forall_not_gett_add_tvar_for_variant
DECLARE E:env, E':env, T3:ty IN
       (forall X:nat. ( (forall T1:ty. not(get_tvar E' X T1))
        =>(forall T2:ty. not(get_tvar E X T2))) )
       |-
       (forall X:nat. ((forall T1:ty. not(get_tvar [tvarb(T3)|E'] X T1))
         =>(forall T2:ty. not(get_tvar [tvarb(T3)|E] X T2))) )
PROOF from "./Well_Formedness_Lemmas/lemma_forall_not_gett_add_tvar_for_variant.sml"
END

/* Lemma wf_type_weaken_variant1 */
LEMMA lemma_wf_type_weaken_variant1
DECLARE E:env, E':env, T:ty IN
       (forall X:nat. ( (forall T1:ty. not(get_tvar E' X T1))
```

```
          => (forall T2:ty. not(get_tvar E X T2))) )
       |-
       (wf_type E T => wf_type E' T)
PROOF from "./Well_Formedness_Lemmas/lemma_wf_type_weaken_variant1.sml"
END

/* Lemma forall_gett_add_tvar */
LEMMA lemma_forall_gett_add_tvar
DECLARE E:env, E':env, T1:ty IN
       (forall T:ty. forall X:nat.
       get_tvar E X T => get_tvar E' X T)
       |-
       (forall T:ty. forall Y:nat. get_tvar [tvarb(T1)|E] Y T
        => get_tvar [tvarb(T1)|E'] Y T )
PROOF from "./Well_Formedness_Lemmas/lemma_forall_gett_add_tvar.sml"
END

/* Lemma wf_type_extensionality */
LEMMA lemma_wf_type_extensionality
DECLARE E:env, E':env, T:ty IN
        |-
        (forall T1:ty. forall X:nat. get_tvar E X T1 => get_tvar E' X T1)
          => (wf_type E T => wf_type E' T)
PROOF from "./Well_Formedness_Lemmas/lemma_wf_type_extensionality.sml"
END

/* Lemma get_bound_add_varb */
LEMMA lemma_get_bound_add_varb
DECLARE E:env, V:ty  IN
        |-
        (forall T:ty. forall X:nat. get_tvar E X T
          => get_tvar [varb(V)|E] X T)
PROOF from "./Well_Formedness_Lemmas/lemma_get_bound_add_varb.sml"
END

/* Lemma wf_env_add_tvar */
LEMMA lemma_wf_env_add_tvar
DECLARE E:env, T:ty IN
        wf_env E, wf_type E T
        |-
        wf_env [tvarb(T)|E]
PROOF from "./Well_Formedness_Lemmas/lemma_wf_env_add_tvar.sml"
END

/* Lemma wf_env_add_var */
LEMMA lemma_wf_env_add_var
DECLARE E:env, T:ty IN
        wf_env E, wf_type E T
        |-
        wf_env [varb(T)|E]
PROOF from "./Well_Formedness_Lemmas/lemma_wf_env_add_var.sml"
END

/* Lemma get_bound_insert_bound_ge */
LEMMA lemma_get_bound_insert_bound_ge
      |-
```

```
      forall X':nat. forall X:nat. forall E:env.
      forall E':env. forall T1:ty. forall T2:ty. forall T3:ty.
      (insert_bound X' E E'∧ natLessEq X' X ∧ get_tvar E' X+1 T1 ∧
      get_tvar E X T2 ∧ typeshift_general T2 1 X' T3) => T1 = T3
PROOF from "./Well_Formedness_Lemmas/lemma_get_bound_insert_bound_ge.sml"
END

/* Lemma get_bound_insert_bound_ge1 */
LEMMA lemma_get_bound_insert_bound_ge1
DECLARE X':nat, X:nat, E:env, E':env, T1:ty, T2:ty, T3:ty IN
        insert_bound1 X' E E', natLessEq X' X ,
        get_tvar E X T2 , typeshift_general T2 1 X' T3
        |-
        get_tvar E' X+1 T3
PROOF from "./Well_Formedness_Lemmas/lemma_get_bound_insert_bound_ge1.sml"
END

/* Lemma get_bound_insert_bound_lt */
LEMMA lemma_get_bound_insert_bound_lt
      |-
      forall X':nat. forall X:nat. forall E:env. forall E':env.
      forall T1:ty. forall T2:ty. forall T3:ty.
      (insert_bound X' E E'∧ natLess X X'
      ∧ get_tvar E' X T1 ∧ get_tvar E X T2 ∧
      typeshift_general T2 1 X' T3) => T1 = T3
PROOF from "./Well_Formedness_Lemmas/lemma_get_bound_insert_bound_lt.sml"
END

/* Lemma get_bound_insert_bound_lt1 */
LEMMA lemma_get_bound_insert_bound_lt1
DECLARE X':nat, X:nat, E:env, E':env, T1:ty, T2:ty, T3:ty IN
        insert_bound X' E E', natLess X X' ,
        get_tvar E X T2 , typeshift_general T2 1 X' T3
        |-
        get_tvar E' X T3
PROOF from "./Well_Formedness_Lemmas/lemma_get_bound_insert_bound_lt1.sml"
END

/* Lemma insert_bound_wf_type_int1 */
LEMMA lemma_insert_bound_wf_type_int1
      |-
      forall X:nat. forall E:env.
      forall E':env. forall T:ty. forall T1:ty.
      insert_bound X E E' =>( wf_type E T =>
                             (typeshift_general T 1 X T1 => wf_type E' T1))
PROOF from "./Well_Formedness_Lemmas/lemma_insert_bound_wf_type_int1.sml"
END

/* Lemma insert_bound_wf_type1 */
LEMMA lemma_insert_bound_wf_type1
DECLARE  E:env, E':env, T:ty, T1:ty, X:nat IN
         insert_bound X E E', wf_type E T,
         typeshift_general T 1 X T1
         |-
         wf_type E' T1
PROOF from "./Well_Formedness_Lemmas/lemma_insert_bound_wf_type1.sml"
```

```
END

/* Lemma insert_bound_wf_env_int1 */
LEMMA lemma_insert_bound_wf_env_int1
      |-
      forall X:nat. forall E:env. forall E':env.
      insert_bound X E E' => ( wf_env E => wf_env E' )
PROOF from "./Well_Formedness_Lemmas/lemma_insert_bound_wf_env_int1.sml"
END

/* Lemma insert_bound_wf_env1 */
LEMMA lemma_insert_bound_wf_env1
DECLARE X:nat, E:env, E':env IN
        insert_bound X E E' , wf_env E
        |-
        wf_env E'
PROOF from "./Well_Formedness_Lemmas/lemma_insert_bound_wf_env1.sml"
END

/* Lemma lemma_utility1_narrow_ne_variant */
LEMMA lemma_utility1_narrow_ne_variant
DECLARE E:env, E':env, T1:ty, T2:ty, X:nat IN
       (exists T:ty. get_tvar E X T ∧ get_tvar E' X T)
       |-
       (exists T:ty. get_tvar [tvarb(T1)|E] X+1 T ∧
        get_tvar [tvarb(T2)|E'] X+1 T)
PROOF from "./Well_Formedness_Lemmas/lemma_utility1_narrow_ne_variant.sml"
END

/* Lemma lemma_utility2_narrow_ne_variant */
LEMMA lemma_utility2_narrow_ne_variant
DECLARE E:env, E':env, T1:ty, T2:ty, T:ty, X:nat IN
       ((forall T:ty. not(get_tvar E X T)) ∧
        (forall T':ty. not(get_tvar E' X T')) )
       |-
       ((forall T:ty. not(get_tvar [tvarb(T1)|E] X+1 T)) ∧
        (forall T':ty.not(get_tvar [tvarb(T2)|E'] X+1 T')))
PROOF from "./Well_Formedness_Lemmas/lemma_utility2_narrow_ne_variant.sml"
END

/* Lemma lemma_utility3_narrow_ne_variant */
LEMMA lemma_utility3_narrow_ne_variant
DECLARE E:env, E':env, T1:ty, T2:ty, X:nat IN
       (exists T:ty. get_tvar E X T ∧ get_tvar E' X T)
       |-
       (exists T:ty. get_tvar [varb(T1)|E] X T ∧ get_tvar [varb(T2)|E'] X T)
PROOF from "./Well_Formedness_Lemmas/lemma_utility3_narrow_ne_variant.sml"
END

/* Lemma lemma_utility4_narrow_ne_variant */
LEMMA lemma_utility4_narrow_ne_variant
DECLARE E:env, E':env, T1:ty, T2:ty, T:ty, X:nat IN
        ((forall T:ty. not(get_tvar E X T)) ∧
        (forall T':ty. not(get_tvar E' X T')) )
        |-
        ((forall T:ty. not(get_tvar [varb(T1)|E] X T)) ∧
```

```
        (forall T':ty.not(get_tvar [varb(T2)|E'] X T')))
PROOF from "./Well_Formedness_Lemmas/lemma_utility4_narrow_ne_variant.sml"
END
```

Subtyping and Narrowing Lemmas

```
/* Lemma sub_wf */
LEMMA lemma_sub_wf
DECLARE E:env, T:ty, U:ty IN
        subt_rel E T U
        |-
        wf_env E ∧ wf_type E T ∧ wf_type E U
PROOF from "./Subtyping_Lemmas/lemma_sub_wf.sml"
END

/* A1 Lemma -> REFLEXIVITY1 */
LEMMA lemma_sub_reflexivity1
DECLARE E:env, T:ty IN
        wf_env E , wf_type E T
        |-
        subt_rel E T T
PROOF from "./Subtyping_Lemmas/lemma_sub_reflexivity1.sml"
END

/* Lemma get_bound_narrow_ne_variant1 */
LEMMA lemma_get_bound_narrow_ne_variant1
DECLARE E:env, E':env, X:nat IN
        narrow_rel X E E'
        |-
        forall X':nat. (not(X'=X) =>
        ((exists T:ty. (get_tvar E X' T ∧ get_tvar E' X' T)) ∨
                         ((forall T:ty. not(get_tvar E X' T)) ∧
                          (forall T':ty. not(get_tvar E' X' T')) )) )
PROOF from "./Subtyping_Lemmas/lemma_get_bound_ne_variant1.sml"
END

/* Lemma sub_extensionality */
LEMMA lemma_sub_extensionality
DECLARE E:env, E':env, U:ty, V:ty IN
        |-
        (forall T:ty. forall X:nat. get_tvar E X T => get_tvar E' X T)
         => (wf_env E' => (subt_rel E U V => subt_rel E' U V))
PROOF from "./Subtyping_Lemmas/lemma_sub_extensionality.sml"
END

/* Lemma sub_weakening_var */
LEMMA lemma_sub_weakening_var
DECLARE E:env, E':env, X:nat, T:ty, U:ty, V:ty IN
        |-
        wf_type E V => (subt_rel E T U
                          => subt_rel [varb(V)|E] T U)
PROOF from "./Subtyping_Lemmas/lemma_sub_weakening_var.sml"
```

```
END

/* Lemma sub_weakening_bound_int1 */
LEMMA lemma_sub_weakening_bound_int1
      |-
      forall X:nat. forall E:env. forall E':env.
      forall T':ty. forall T'':ty. forall U:ty. forall V:ty.
      insert_bound1 X E E' => ( subt_rel E U V =>
      (typeshift_general U 1 X T' =>
      (typeshift_general V 1 X T'' => subt_rel E' T' T'')) )
PROOF from "./Subtyping_Lemmas/lemma_sub_weakening_bound_int1.sml"
END

/* Lemma get_bound_narrow_eq_int1 */
LEMMA lemma_get_bound_narrow_eq_int1
DECLARE X:nat, E:env, E':env IN
        narrow_rel X E E'
        |-
        exists T : ty. get_tvar E X T
PROOF from "./Subtyping_Lemmas/lemma_get_bound_narrow_eq_int1.sml"
END

/* Lemma get_bound_narrow_eq_int2 */
LEMMA lemma_get_bound_narrow_eq_int2
DECLARE X:nat, E:env, E':env IN
        narrow_rel X E E'
        |-
        exists T : ty. get_tvar E' X T
PROOF from "./Subtyping_Lemmas/lemma_get_bound_narrow_eq_int2.sml"
END

/* Lemma get_bound_narrow_eq1 */
LEMMA lemma_get_bound_narrow_eq1
DECLARE X:nat, E:env, E':env IN
        narrow_rel X E E'
        |-
        (exists T:ty. exists T':ty. (get_tvar E X T ∧ get_tvar E' X T' ∧
                                     subt_rel E' T' T) )
PROOF from "./Subtyping_Lemmas/lemma_get_bound_narrow_eq1.sml"
END

/* Lemma wf_second_env_narrow */
LEMMA lemma_wf_second_env_narrow
DECLARE X:nat, E:env, E':env IN
        narrow_rel X E E'
        |-
        wf_env E'
PROOF from "./Subtyping_Lemmas/lemma_narrow_wf_env.sml"
END

/* Lemma narrow_wf_env1 */
LEMMA lemma_narrow_wf_env1
DECLARE X:nat, E:env, E':env IN
        narrow_rel X E E' , wf_env E
        |-
        wf_env E'
```

```
PROOF from "./Subtyping_Lemmas/lemma_narrow_wf_env1.sml"
END

/* Lemma narrow_wf_type */
LEMMA lemma_narrow_wf_type
DECLARE X:nat, E:env, E':env, T:ty IN
        |-
        narrow_rel X E E' => (wf_type E T => wf_type E' T)
PROOF from "./Subtyping_Lemmas/lemma_narrow_wf_type.sml"
END

/* Lemma narrow_wf_type1 */
LEMMA lemma_narrow_wf_type1
DECLARE X:nat, E:env, E':env, T:ty IN
        narrow_rel X E E' , wf_type E T
        |-
        wf_type E' T
PROOF from "./Subtyping_Lemmas/lemma_narrow_wf_type1.sml"
END

/* Lemma typeshift preserves size */
LEMMA lemma_typeshift_preserves_size
DECLARE T:ty, T':ty, X:nat, R:nat IN
        |-
        typeshift_general T 1 X T' => (size T' R  => size T R)
PROOF from "./Subtyping_Lemmas/lemma_typeshift_preserves_size.sml"
END

/* Lemma typeshift preserves size variant */
LEMMA lemma_typeshift_preserves_size_variant
DECLARE T:ty, T':ty, X:nat, R:nat IN
        typeshift_general T 1 X T' , size T' R
        |-
        size T R
PROOF from "./Subtyping_Lemmas/lemma_typeshift_preserves_size_variant.sml"
END

/* Lemma that says if two terms are equal, they have the same size */
LEMMA lemma_terms_eq_same_size
DECLARE T1:ty, R1:nat, R2:nat IN
        size T1 R1, size T1 R2
        |-
        R1=R2
PROOF from "./Subtyping_Lemmas/lemma_terms_eq_same_size.sml"
END

/* Lemma size of tyfun greater then size of one of its arguments */
LEMMA lemma_tyfun_size_1
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        |-
        size tyfun(T1,T2) R1 => (size T1 R2 => natLess R2 R1)
PROOF from "./Subtyping_Lemmas/lemma_tyfun_size_1.sml"
END

/* Lemma size of tyfun greater then size of one of its arguments */
LEMMA lemma_tyfun_size_1_variant
```

```
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        size tyfun(T1,T2) R1 , size T1 R2
        |-
        natLess R2 R1
PROOF from "./Subtyping_Lemmas/lemma_tyfun_size_1_variant.sml"
END

/* Lemma size of tyfun greater then size of one of its arguments */
LEMMA lemma_tyfun_size_2
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        |-
        size tyfun(T1,T2) R1 => (size T2 R2 => natLess R2 R1)
PROOF from "./Subtyping_Lemmas/lemma_tyfun_size_2.sml"
END

/* Lemma size of tyfun greater then size of one of its arguments */
LEMMA lemma_tyfun_size_2_variant
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        size tyfun(T1,T2) R1 , size T2 R2
        |-
        natLess R2 R1
PROOF from "./Subtyping_Lemmas/lemma_tyfun_size_2_variant.sml"
END

/* Lemma size of tyall greater then size of one of its arguments */
LEMMA lemma_tyall_size_1
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        |-
        size tyall(T1,T2) R1 => (size T1 R2 => natLess R2 R1)
PROOF from "./Subtyping_Lemmas/lemma_tyall_size_1.sml"
END

/* Lemma size of tyall greater then size of one of its arguments */
LEMMA lemma_tyall_size_1_variant
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        size tyall(T1,T2) R1, size T1 R2
        |-
        natLess R2 R1
PROOF from "./Subtyping_Lemmas/lemma_tyall_size_1_variant.sml"
END

/* Lemma size of tyall greater then size of one of its arguments */
LEMMA lemma_tyall_size_2
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        |-
        size tyall(T1,T2) R1 => (size T2 R2 => natLess R2 R1)
PROOF from "./Subtyping_Lemmas/lemma_tyall_size_2.sml"
END

/* Lemma size of tyall greater then size of one of its arguments */
LEMMA lemma_tyall_size_2_variant
DECLARE T1:ty, T2:ty, R1:nat, R2:nat IN
        size tyall(T1,T2) R1, size T2 R2
        |-
        natLess R2 R1
PROOF from "./Subtyping_Lemmas/lemma_tyall_size_2_variant.sml"
```

```
END
```

Main Proof Lemmas

```
/* Crucial step in the proof, it says that transitivity of Q holds if we suppose
   both transitivity and narrowing hold for smallet cut type Q' */
LEMMA transitivity_case
      |-
      forall Q:ty.
     (forall Q':ty.
     (exists R1:nat. exists R2:nat.
      size Q R1 ∧ size Q' R2 ∧ natLess R2 R1)
      => (transitivity_prop Q' ∧ narrowing_prop Q'))  => transitivity_prop Q
PROOF from "./Main_Lemmas/Transitivity_Case.sml"
END

/* Crucial step in the proof of narrowing, showing that narrow Q holds if
   we assume transitivity of types of same size of Q */
LEMMA narrowing_case
      |-
      forall Q:ty.
      (forall Q':ty. forall R1:nat.
      (size Q R1 ∧ size Q' R1)
        => transitivity_prop Q') => narrowing_prop Q
PROOF from "./Main_Lemmas/Narrowing_Case.sml"
END

/* Lemma that says if two type have the same size, the transitivity holds */
LEMMA transitivity_types_same_size
DECLARE R:nat, Q:ty, Q':ty IN
        size Q' R, size Q R
        |-
        transitivity_prop Q'
PROOF from "./Main_Lemmas/Transitivity_Type_Same_Size.sml"
END

LEMMA trans_narrow_ind_n
      |-
      forall N:nat. forall Q:ty. forall R:nat.
      (size Q R ∧ natLess R N) =>
      (transitivity_prop Q ∧ narrowing_prop Q)
PROOF from "./Main_Lemmas/Trans_Narrow_ind_n.sml"

END

/* Final Proof of transitivity and narrowing for all types Q */
LEMMA transitivity_and_narrowing
      |-
      forall Q:ty.
      transitivity_prop Q ∧ narrowing_prop Q
PROOF from "./Main_Lemmas/Transitivity_and_Narrowing.sml"
END
```

Appendix C

Appendix C

Appendix C shows some further functions encoded in VCPT that are not related with the part of the challenge solved. These functions can be used to start approaching other parts of the challenge.

Further Datatypes and Functions

Part 1.a of the challenge deals only with types without considering terms. Since the material of this Appendix is out of the scope of this thesis we limit to short introductions. Further information can be retrieved from [ABF+05].

Let x be a variable name, let t be a generic λ-term and let X,T be types. The syntax of λ-terms (as introduced in paper [ABF+05]) is showed in table C.1.

t	::=		terms
		x	variable
		$\lambda x : T.t$	abstraction
		$t\ t$	application
		$\lambda X <: T.t$	type abstraction
		$t\ [T]$	type application

Table C.1: Syntax of Terms

The syntax in Table C.1 can straightforwardly be encoded in VCPT as follows:

```
datatype        term =  tmvar of nat
                      | tmabs of ty * term
                      | tmtabs of ty * term
                      | tmapp of term * term
                      | tmtapp of term * ty;
```

Terms have been represented with the de Bruijn indices representation discussed in Chapter 2. For this reason, functions to shift terms and types within terms are required.

More specifically: let $T, Tcurry$ be λ-terms, let D, C be natural numbers, then the function `termshift_term` T D C $Tcurry$ shifts a term T from cutoff C by D. As usual the result is stored in an additional variable called $Tcurry$. The function `termshift_terms` is declared in VCPT in the following way:

```
termshift_term : term --> nat --> nat --> term --> prop <=
                    \ T:term. \ D:nat. \ C:nat. \ Tcurry:term.
 cases T of
       tmvar(X)       => if( natLess X C ) then
                           (Tcurry = tmvar(X))
                          else
                           (exists Z:nat. (natPlus X D Z ∧
                            Tcurry = tmvar(Z))
                           )
       |tmabs(Ty,Tm)  => (exists Tm1:term. exists Z:nat.
                          natPlus 1 C Z ∧
                          termshift_term Tm D Z Tm1 ∧
                          Tcurry = tmabs(Ty , Tm1)
                         )
       |tmtabs(Ty,Tm) => (exists Tm1:term.
                          termshift_term Tm D C Tm1 ∧
                          Tcurry = tmtabs(Ty , Tm1)
                         )
       |tmapp(Tm1,Tm2)=> (exists Tm11:term. exists Tm21:term.
                          termshift_term Tm1 D C Tm11 ∧
                          termshift_term Tm2 D C Tm21 ∧
                          Tcurry = tmapp(Tm11 , Tm21)
                         )
       |tmtapp(Tm,Ty) => (exists Tm1:term.
                          termshift_term Tm D C Tm1 ∧
                          Tcurry = tmtapp(Tm1 , Ty)
                         )
    end
end;
```

Function `termshift_type` $T\ D\ C\ Tcurry$, in the same fashion of `termshift_term` $T\ D\ C\ Tcurry$, shifts types within a generic term T. All the free variables of types within term T will be shifted by D. The function `typeshift_general` has been introduced in Section 5.1.2.

```
termshift_type : term --> nat --> nat --> term --> prop <=
                    \T:term. \D:nat. \C:nat. \Tcurry:term.
   cases T of
          tmvar(X)     => Tcurry = tmvar(X)
         |tmabs(Ty,Tm)  => (exists Ty1:ty. exists Tm1:term.
                            typeshift_general Ty D C Ty1 ∧
                            termshift_type Tm D C Tm1 ∧
                            Tcurry = tmabs(Ty1 , Tm1)
                           )
         |tmtabs(Ty,Tm) => (exists Ty1:ty. exists Tm1:term.
                            exists Z:nat.
                            typeshift_general Ty D C Ty1 ∧
                            natPlus 1 C Z ∧
                            termshift_type Tm D Z Tm1 ∧
                            Tcurry = tmtabs(Ty1 , Tm1)
                           )
         |tmapp(Tm1,Tm2)=> (exists Tm11:term. exists Tm21:term.
                            termshift_type Tm1 D C Tm11 ∧
                            termshift_type Tm2 D C Tm21 ∧
                            Tcurry = tmapp(Tm11 , Tm21)
```

```
                  )
         |tmtapp(Tm,Ty) => (exists Tm1:term. exists Ty1:ty.
                            termshift_type Tm D C Tm1 ∧
                            typeshift_general Ty D C Ty1 ∧
                            Tcurry = tmtapp(Tm1 , Ty1)
                           )
 end
end;
```

Let now $T, Tx, Tcurry$ be nameless λ-types and let K be a natural number. Function `type_subst` $T\ K\ Tx\ Tcurry$ substitutes the type variable K with Tx within T and returns the result in $Tcurry$.

```
type_subst : ty --> nat --> ty --> ty --> prop <=
      \T:ty. \K:nat. \Tx:ty. \Tcurry:ty.
cases T of
        tyvar(X)        => if(X = K) then
                  (Tcurry = Tx)
                           else
                            if (natLess X K) then
                             (Tcurry = tyvar(X))
                            else
                             (exists Z:nat.
                              natMinus X 1 Z ∧
                              Tcurry = tyvar(Z)
                             )
        |top()           => Tcurry = top()
        |tyfun (T1 , T2) =>(exists T3:ty. exists T4:ty.
                            type_subst T1 K Tx T3 ∧
                            type_subst T2 K Tx T4 ∧
                            Tcurry = tyfun(T3 , T4)
                           )
        |tyall (T1 , T2)  =>(exists T5:ty. exists Z:nat.
                             exists T3:ty. exists T4:ty.
                             natPlus 1 K Z ∧
                             typeshift_byone Tx T3 ∧
                             type_subst T1 K Tx T5 ∧
                             type_subst T2 Z T3 T4 ∧
                             Tcurry = tyall(T5 , T4 )
                            )
     end
end;
```

Let $T, T2, Tcurry$ be nameless λ-terms and let X be a natural number.

Function `term_subst_term` $T\ X\ T2\ Tcurry$ substitutes the term variable X with $T2$ within T and returns the result in $Tcurry$.

```
term_subst_term : term --> nat --> term --> term --> prop <=
                  \T:term. \X:nat. \T2:term. \Tcurry:term.
  cases T of
         tmvar(Y)     =>if ( Y = X ) then
                          (Tcurry = tmvar(Y))
                         else
```

```
                        if (natLess Y X) then
                         (Tcurry = tmvar(Y))
                        else
                         (exists Z:nat.
                          natMinus X 1 Z ∧
                          Tcurry = tmvar(Z)
                         )
        |tmabs(Ty,Tm)  =>(exists Z:nat. exists T21:term.
                          exists Tm1:term.
                          natPlus 1 X Z ∧
                          termshift_term T2 1 0 T21 ∧
                          term_subst_term Tm Z T21 Tm1 ∧
                          Tcurry = tmabs(Ty , Tm1)
                         )
        |tmtabs(Ty,Tm) =>(exists T21:term. exists Tm1:term.
                          termshift_type T2 1 0 T21 ∧
                          term_subst_term Tm X T21 Tm1 ∧
                          Tcurry = tmtabs(Ty , Tm1)
                         )
        |tmapp(Tm1,Tm2)=>(exists Tm11:term. exists Tm21:term.
                          term_subst_term Tm1 X T2 Tm11 ∧
                          term_subst_term Tm2 X T2 Tm21 ∧
                          Tcurry = tmapp(Tm11 , Tm21)
                         )
        |tmtapp(Tm,Ty) =>(exists Tm1:term.
                          term_subst_term Tm X T2 Tm1 ∧
                          Tcurry = tmtapp(Tm1 , Ty)
                         )
    end
end;
```

In the end, let $T, Tcurry$ be λ-terms, let Ty be a λ-type and let X be a natural number. Function `term_subst_type` $T\ X\ Ty\ Tcurry$ substitutes the type variable indexed X with Ty within T and returns the result in $Tcurry$.

```
term_subst_type : term --> nat --> ty --> term --> prop <=
                  \T:term. \X:nat. \Ty:ty. \Tcurry:term.
     cases T of
            tmvar(Y)      => Tcurry = tmvar(Y)
           |tmabs(Ty1,Tm) =>(exists Ty11:ty. exists Tm1:term.
                             type_subst Ty1 X Ty Ty11 ∧
                             term_subst_type Tm X Ty Tm1 ∧
                             Tcurry = tmabs(Ty11 , Tm1)
                            )
           |tmtabs(Ty1,Tm)=>(exists Tys:ty. exists Ty11:ty.
                             exists Tm1:term. exists Z:nat.
                             type_subst Ty1 X Ty Ty11 ∧
                             natPlus 1 X Z ∧
                             typeshift_general Ty 1 0 Tys ∧
                             term_subst_type Tm Z Tys Tm1 ∧
                             Tcurry = tmtabs(Ty11 , Tm1)
                            )
           |tmapp(Tm1,Tm2)=>(exists Tm11:term. exists Tm21:term.
                             term_subst_type Tm1 X Ty Tm11 ∧
                             term_subst_type Tm2 X Ty Tm21 ∧
```

```
                            Tcurry = tmapp(Tm11 , Tm21)
                           )
          |tmtapp(Tm,Ty1)=>(exists Tm1:term. exists Ty11:ty.
                            term_subst_type Tm X Ty Tm1 ∧
                            type_subst Ty1 X Ty Ty11 ∧
                            Tcurry=tmtapp(Tm1 , Ty11)
                            )
      end
end;
```

Bibliography

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbair, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses : The *POPLMARK* challenge. `http://www.cis.upenn.edu/~plclub/wiki-static/poplmark.pdf`, May 2005.

[A.M03] Pitts A.M. Nominal logic, a first order theory of names and bindings. information and computation, 2003.

[Bar07] H. P. Barendregt. *The Lambda Calculus*. North-Holland, 10 2007.

[BC04] Yves Bertot and Pierre Casteran. Interactive theorem proving and program development, 2004.

[BW] B. Barras and B. Werner. Coq in coq. to appear in journal of automated reasoning.

[Can82] G. Cantor. Contributions to the founding of the theory of transfinite numbers, 1882.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic, 1982.

[CG94] Pier-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $f_{<:}$, 1994.

[Cha09] Arthur Charguéraud. The locally nameless representation, 2009.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system f with subtyping. *Informations and Computation*, 1994.

[Com] PoplMark Community. Poplmark's solutions. `http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/index.php?title=Submitted_solutions`.

[CU05] Christine Tasson Christian Urban. Nominal reasoning techniques in isabelle/hol, 2005.

[Dam98] Mads Dam. Proving properties of dynamic process network. *Information and computation*, 1998.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the chuirch rosser theorem., 1972.

[DG00] Mads Dam and Dilian Gurov. μ-calculus with explicit points and approximations. *In fixed points in computer science*, 2000.

[Fre01] Lars–åke Fredlund. A framework for reasoning about erlang code. phd thesis. *Swedish Institute of Computer Science*, 2001.

[GM93] M.J.C. Gordon and T.F. Melham. Introduction to hol: A theorem proving enviroment for higher order logic, 1993.

[MCP93] Colin Mayers, Chris Clack, and Ellen Poon. *Programming with standard ML*. Prentice-Hall, 1993.

[MT92] Karl Meinke and John V. Tucker. Universal algebra, 1992.

[Nip01] T. Nipkow. More church-rosser proofs(in isabelle/hol). journal of automated reasoning, 2001.

[ORR+96] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. Pvs: Combining specification, proof checking, and model checking, 1996.

[Pau94a] Lawrence C. Paulson. Designing a theorem prover, 1994.

[Pau94b] L.C. Paulson. Isabelle: A generic theorem prover, 1994.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar, 1982.

[SD03] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning, circular and tree-shaped proofs in the μ-calculus, 2003.

[Sti01] Julian Bradfield Colin Stirling. Modal mu-calculi:an introduction, 2001.

[Win93] Glynn Winskell. *The formal semantic of computer programming languages:An introduction*. MIT Press, 1993.