Functional Program Verification in Theorema

DISSERTATION

zur Erlangung des akademischen Grades

DOKTOR DER TECHNISCHEN WISSENSCHAFTEN

Angefertigt am *Institut für Symbolisches Rechnen*

Begutachter:

Prof. Dr. Tudor Jebelean
Prof. Dr. Josef Schicho, RICAM Linz, ÖAW

Eingereicht von:

Nikolaj Popov

Linz, June 2008

Abstract

In this thesis we present a method for verifying recursive functional programs. We define a Verification Condition Generator (VCG) which covers the most frequent types of recursive programs (including nested recursive and mutual recursive ones). These programs may operate on arbitrary domains. We prove *Soundness* and *Completeness* of the VCG and this provides a warranty that any system based on our results will be sound.

As a distinctive feature of our method, the verification conditions do not refer to a theoretical model for program semantics or program execution, but only to the theory of the domain used in the program. This is very important for the automatic verification, because any additional theory present in the system would significantly increase the proving effort.

We introduce here the notion of *Completeness* of a VCG as a duality of *Soundness*. It is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition will carry over to a counterexample for the given program and specification. Moreover, the failing proof gives information about the place of the bug.

Furthermore, we introduce a specialized strategy for termination. The termination problem is reduced to the termination of a simplified version of the program. The conditions for the simplified versions are identical for entire classes of functional programs, thus they are highly reusable.

The research presented in this thesis is performed in the frame of the *Theorema* system, a mathematical computer assistant which aims at supporting the entire process of mathematical theory exploration. Our results on verification conditions complement the Theorema capabilities for programming and proving.

Zusammenfassung

In dieser Dissertation stellen wir eine experimentelle Prototyp-Umgebung für die Definition und Verifikation rekursiver funktionaler Programme vor. Wir definieren einen Verifikationsbedingungserzeuger (VCG), welcher für die häufigsten Typen von Programmen ausreicht, die in beliebigen Bereichen arbeiten können (einschließlich Programmen mit geschachtelter oder wechselseitiger Rekursion). Wir beweisen Korrektheit und Vollständigkeit des VCG, und das garantiert, daß jedes auf unseren Ergebnissen beruhende System korrekt ist.

Ein besonderer Gesichtspunkt unserer Methode ist, daß sich die Verifikationsbedingungen nicht auf ein bestimmtes theoretisches Modell für die Semantik oder Exekution eines Programmes beziehen, sondern ausschließlich auf die Theorie des Bereichs, der im Programm verwendet wird. Das ist für die automatische Verifikation sehr wichtig, da jede zusätzliche dem System innewohnende Theorie den für den Beweis nötigen Aufwand beträchtlich erhöhen würde.

Wir führen hier den Begriff der Vollständigkeit eines VCG als Dual zur Korrektheit ein. Das ist aus den folgenden beiden Gründen wichtig: theoretisch als das Dual zur Korrektheit, und praktisch als eine Hilfe beim Finden und Korrigieren von Fehlern. Jedes Gegenbeispiel für eine fehlerhafte Verifikationsbedingung führt zu einem Gegenbeispiel für das Programm und seine Spezifikation. Darüber hinaus sind wir in der Lage, die genaue Stelle des Fehler festzustellen.

Weiters führen wir eine spezielle Strategie für Terminationsbeweise ein. Sie werden auf den Nachweis von Eigenschaften vereinfachter Versionen zurückgeführt, welche ihrerseits neu verwendet werden können, sodaß genaue Terminationsbeweise in vielen Fällen ausgelassen werden können.

Die Forschungsarbeit, die in dieser Dissertation präsentiert wird, wird im Rahmen des Theorema-Systems durchgeführt, einem mathematischen Computer-Assistenten mit dem Ziel, den gesamten Prozeß der Erforschung einer mathematischen Theorie zu unterstützen: die Erfindung mathematischer Begriffe, Finden und Verifikation (Beweis) von Aussagen A über Begriffe, das Finden von Problemen, die mit Hilfe der Begriffe formuliert werden könnnen, Erfindung und Verifikation (Korrektheitsbeweis) von Algorithmen, sowie Abspeicherung und Wiederfinden von Formeln, die während dieses Prozeßes gefunden und verifiziert wurden. Das System enthält eine Sammlung von allgemeinen und speziellen Beweisern für verschiedene interessante Bereiche (z.B. den ganzen Zahlen, Mengen, reellen Zahlen, Tupeln, usw.).

This work is dedicated to my teachers:

Ljubomir Popov,
Rayna Chavdarova,
Anatoly Buda,
Tudor Jebelean, and
Bruno Buchberger,

who helped me to find myself in science. Not only did they help me studying mathematics and logic, but what I learned has influenced the way I think.

Acknowledgements

I would like to thank my advisor Tudor Jebelean for the incredible support he offered me during all the phases of my PhD studies. We had, still have, and (I am sure) will have mathematical and non-mathematical discussions which were (are, will be) very provocative and inspirative. They influenced much the way I think.

I would like to thank the advisor of my advisor Bruno Buchberger for teaching me how to become a scientist. Moreover, he was next to me throughout all the steps I made as young researcher. I wish him being next to me for as long as possible.

I would like to thank my second advisor Josef Schicho for kindly agreeing to be the second advisor of this thesis. Moreover, I am very grateful for the discussions we had, sometimes (in fact, many times) going beyond mathematics and philosophy. I will always remember them.

I would like to thank Wolfgang Schreiner for his "small" advices—they were always just-in-time.

I would like to thank Heinrich Rolletschek for the hundreds of hours he spent with me during our seminars and private discussions.

I would like to thank all my colleagues from the Theorema group: Adrian Craciun, Alexander Zapletal, Besik Dundua, Camelia Rosenkranz, Florina Piroi, Gabor Kusper, Georg Regensburger, Judit Robu, Koji Nakagawa, Laura Kovacs, Loredana Tec, Martin Giese, Madalina Erascu, Madalina Hodorog, Robert Vajda, Temur Kutsia and Wolfgang Windsteiger. Working with them is a great joy.

I would like to thank Temur Kutsia and Mohamed Shalaby for their friendship. There are advises that only an elder brother may give you.

I would like to thank Markus Rosenkranz for his friendship. He helped me a lot during my first years at RISC—without him my live would have been much more difficult.

I would like to thank Burkhard Zimmermann for the help he offered me when I needed to learn some details from his subject—Algorithmic Combinatorics.

I would like to thank Veronika Pillwein, Manuel Kauers and Christoph Koutschan for the technical support they offered me. They are my lovely neighbors.

I would like to thank Karoly Erdei and Werner Danielczyk-Landerl—they are the system administrators of the RISC institute. Without their competent work, my time would be much more inefficient.

The work presented in this thesis was supported by:

- Upper Austrian Government grant for PhD studies, 2000-2002,
- The project “e-Austria Timisoara” funded by BMBWK (Austrian Ministry of Education, Science, and Culture), 2002-2003,
- Austrian Science Foundation (FWF) project SFB F1302, 2003-2008,
- INTAS Project “Practical Formal Verification Using Automated Reasoning and Model Checking” (Nr 05-1000008-8144), 2006-2008.

Contents

Chapter 1

Introduction

The introduction at hand exceeds the standards of a thesis introduction, however, we want to make the thesis readable and understandable for a bigger audience. Thus, in order to make the content selfcontained, we chose a tutorial-like style. The introduction contains a brief presentation of matters which are closely related to the topic of the thesis.

Although the beginning of program verification dates back to 1950-s, the level it has achieved so far is not satisfactory. At the same time, the software production is rapidly growing, and in fact, only very small part of that production goes trough (some kind of) verification (or validation) process. We are convinced that in order to increase the quality of the software production, program verification, and formal methods should play a bigger role during the process of software design and composition.

The thesis at hand is dedicated to the development of a relevant theory, which may serve the practical need of proving program correctness in an automatic manner. Along with the theoretical development, many practical examples are demonstrated. Readers who would not be interested on reading the mathematical proofs of the theorems presented here may look at the main results and at the examples.

Although the examples presented here appear to be relatively simple, they already demonstrate the usefulness of our approach in the general case. We aim at extending these experiments to industrial-scale examples, which are in fact not more complex from the mathematical point of view. Furthermore we aim at improving the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

The main literature sources we used during the preparation of this chapter are [46], [44], [2], [9], [61], [58], and [65].

1.1 Specification and Verification

Program specification (or formal specification of a program) is the definition of what a program is expected to do. Normally, it does not describe, and it should not, how the program is implemented. The specification is usually provided by logical formulae describing a relationship between input and output parameters. We will consider specifications which are pairs, containing a precondition (input condition) and a postcondition (output condition).

Given such a specification, it is possible to use formal verification techniques to demonstrate that a program is correct with respect to the specification.

A precondition (or input predicate) of a program is a condition that must always be true just prior to the execution of that program. It is expressed by a predicate on the input of the program. If a precondition is violated, the effect of the program becomes undefined and thus may or may not carry out its intended work. For example: the factorial is only defined for integers greater than or equal to zero. So a program that calculates the factorial of an input number would have preconditions that the number be an integer and that it be greater than or equal to zero.

A postcondition (or output predicate) of a program is a condition that must always be true just after the execution of that program. It is expressed by a predicate on the input and the output of the program.

Remark: We do not consider here informal specifications, which are normally written as comments between the lines of code.

Formal verification (from Latin: verus - true) is, in general, the act of proving mathematically the correctness of a program with respect to a certain formal specification. Software testing, in contrast to verification, cannot prove that a system does not contain any defects, neither that it has a certain property, e.g., correctness with respect to a specification. Only the process of formal verification can prove that a system does not have a certain defect or does have a certain property.

There are roughly two main approaches to formal verification: *Model checking* [19], which consists of an exploration of the mathematical model—that is only possible for finite models; *Program verification*—where our contribution falls—consists of using logical reasoning about the program, usually with the help of a theorem prover [44].

Model checking is the process of checking whether a given structure is a model of a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is testing whether a given formula in the propositional logic is satisfied by a given structure.

The structure is usually given as a source code in a special-purpose language. Such a program corresponds to a finite state machine, i.e., a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.

Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula p, and a structure M with initial state s, decide if

$$M, s \models p.$$

If M is finite, as it is in hardware, model checking reduces to a graph search.

Program verification, in contrast to model checking, may deal with infinitely many instances of input objects and may provide a rigorous formal mathematical warranty that the desired specification is obeyed.

The problem of verifying programs is usually split into two subproblems: generate verification conditions which are sufficient for the program to be correct and prove the verification conditions, within the theory of the domain for which the program is defined. In this thesis, we address mainly the first of these subproblems, namely the generation of the verification conditions.

1.2 Verification Condition Generator. Correctness and Completeness

A common approach to program verification is the axiomatic reasoning, which was initially developed by Floyd [24] for the verifications of flowcharts. This method was then further developed by Hoare [31] for dealing with *while*-programs and became known as Hoare Logic. The method provides a set of logical rules allowing to reason about the correctness of computer programs with the methods of mathematical logic.

The central feature of Hoare logic is the Hoare triple. It describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form:

$$\{P\}\ C\ \{Q\}, \tag{1.1}$$

where P and Q are assertions (normally given by logical formulae) and C is a command, or a program. In the literature, P is called the precondition and Q the postcondition. Hoare logic has axioms and inference rules for all the constructs of simple imperative programming languages.

In order to reason about programs, one first translates the problem of a proving program property into a problem of proving logical assertions, and then prove the assertions. A common way for making such translation is by some verification condition generator—its name is self-explanatory.

A Verification Condition Generator (VCG) is a device—normally implemented by a program—which takes a program, actually its source code, and the specification, and produces verification conditions. These verification conditions do not contain any part of the program text, and are expressed in a different language, namely they are logical formulae.

$$\langle Program, Specification\rangle \underset{VCG}{\longrightarrow} Verification\ \ Conditions \tag{1.2}$$

Let us say, the program is F and the specification I_F (input predicate), and O_F (output predicate) is provided. The verification conditions generated by VCG are: $VC_1, VC_2, \ldots, VC_n$, that is:

$$\langle F, \langle I_F, O_F\rangle\rangle \underset{VCG}{\longrightarrow} VC_1 \wedge \cdots \wedge VC_n. \tag{1.3}$$

After having the verification conditions at hand, one has to prove them as logical formulae in the theory of the domain on which the program is defined, e.g., integers, reals, etc. We denote this theory by $Th[\mathfrak{D}_F]$.

Normally, these conditions are given to an automatic or semi-automatic theorem prover. If all of them hold, then the program is correct with respect to its specification. The latter statement we call *Soundness* of the VCG, namely:

Given a program F and a specification I_F (input condition), and O_F (output condition), if the verification conditions generated by the VCG hold as logical formulae, then the program F is correct with respect to the specification $\langle I_F, O_F\rangle$:

$$if \qquad Th[\mathfrak{D}_F] \models VC_1 \wedge \ldots \wedge VC_n \qquad then \qquad F\ is\ correct\ with\ respect\ to\ \langle I_F, O_F\rangle. \tag{1.4}$$

It is clear that whenever one defines a VCG, the first task to be done is proving its soundness statement—otherwise it would not be properly called: Verification Condition Generator.

Completing the notion of *Soundness* of a VCG, we introduce its dual—*Completeness*—which is introduced by the author of this thesis in [40]. The respective *Completeness* statement of the VCG is :

Given a program F and a specification I_F (input condition), and O_F (output condition), if the program F is correct with respect to the specification $\langle I_F, O_F \rangle$, then the verification conditions generated by the VCG hold as logical formulae:

$$if \qquad F\, is\, correct\, with\, respect\, to\, \langle I_F, O_F \rangle \qquad then \qquad Th[\mathfrak{D}_F] \models VC_1 \wedge \ldots \wedge VC_n. \tag{1.5}$$

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on “what is wrong”.

Indeed, most books about program verification present methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

1.3 Partial and Total Correctness of a Program

A distinction is made between total correctness, which additionally requires that the program terminates, and partial correctness, which simply requires that if an answer is returned (that is, the program terminates) it will be correct.

For example, if we are successively searching through integers $1, 2, 3, \dots$ to see if we can find an example of some phenomenon—say an odd perfect number—it is quite easy to write a partially correct program (use integer factorization to check n as perfect or not). But to say this program is totally correct would be to assert something currently not known in number theory.

The relation between partial and total correctness is informally given by:

$$Total\ Correctness\ =\ Partial\ Correctness\ +\ Termination.$$

We give the necessary formal definitions in what follows. Let us say, given the program F and the specification I_F (input predicate), and O_F (output predicate). Henceforth, by $\downarrow$ we denote the predicate expressing termination. We will write $F[x]\downarrow$ and say "F terminates on x". Partial correctness of F is expressed by the formula:

$$(\forall x : I_F[x])\ (F[x]\downarrow\ \Longrightarrow O_F[x, F[x]]). \tag{1.6}$$

Termination of F is expressed by:

$$(\forall x : I_F[x])\ F[x]\downarrow, \tag{1.7}$$

and total correctness of F respectively:

$$(\forall x : I_F[x])\ (F[x]\downarrow \wedge O_F[x, F[x]]). \tag{1.8}$$

Logically, it is clear that partial correctness (1.6) and termination (3.9) imply total correctness (1.8), and throughout this thesis we make use of this fact.

The following example of a powering program accomplished with an appropriate specification illustrates the previous formulae. Let pow be the program:

$$pow[x, n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } x * pow[x, n-1], \tag{1.9}$$

which for any two numbers x and n, such that $x \in \mathbb{R}$ and $n \in \mathbb{N}$, computes the number x^n.

Now, the constraint that $x \in \mathbb{R}$ and $n \in \mathbb{N}$ forms the input condition, namely

$$(\forall x, n)\ \ (I_{pow}[x, n]\ \ \Longleftrightarrow\ \ x \in \mathbb{R} \wedge n \in \mathbb{N}), \tag{1.10}$$

and the desire that for such x and n the program $pox[x, n]$ computes x^n forms the output condition:

$$(\forall x, n, y)\ \ (O_{pow}[x, n, y]\ \ \Longleftrightarrow\ \ x^n = y). \tag{1.11}$$

The total correctness formula is now:

$$(\forall x : x \in \mathbb{R})(\forall n : n \in \mathbb{N})\ (pow[x, n]\downarrow\ \wedge x^n = pow[x, n]). \tag{1.12}$$

Proving the correctness formula (1.12) in the theory of $\mathbb{N}$ and $\mathbb{R}$ is actually proving the total correctness of the program (1.9) with respect to the specification (1.10), (1.11).

1.4 Imperative and Functional Programming

Programming languages are generally divided into two major groups, namely imperative and functional ones. Although, "real" programming languages are often hybrids of both programming paradigms, it is easier to split them and reason about them separately.

Imperative programming, in contrast to functional programming, is a programming paradigm that describes computation as statements that change a program state. In much the same way as the imperative mood in natural languages expresses commands to take action, imperative programs are a sequence of commands for the computer to perform.

For a comprehensive overview on the recent achievements in verification of imperative programming we refer to the work of Kovacs [38].

Functional programming is very different from imperative programming. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement the state. Pure functional programming does not allow side effects at all, thus it is easier to verify, optimize, and parallelize programs, and easier to write automated tools to perform those tasks.

Functional programming is a programming paradigm that treats computation as the evaluation of functions. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state [33].

From logical point of view, functional programs are equivalent to conditional equalities, and, therefore they are logical formulae.

Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be repeated.

The main focus of this thesis is the development of tools for verification of pure functional recursive programs. Henceforth, "pure" and "functional" front of recursive programs, will be omitted in order to simplify the presentation.

1.5 Automatic Theorem Proving. The Theorema System

Automatic Theorem Proving, and more generally, Automated Reasoning is a border area of computer science and mathematics dedicated to understanding different aspects of reasoning in a way that allows the creation of software which makes computers to reason completely or nearly completely automatically. Automatic theorem proving is, in particular, the proving of mathematical theorems by an algorithm. In contrast to proof checking, where an existing proof for a theorem is certified valid, automatic theorem provers generate the proofs themselves. A recent and relatively comprehensive overview on that aria may be found in [64].

The research presented in this thesis is performed in the frame of the *Theorema* system [14], a mathematical computer assistant which aims at supporting the entire process of mathematical theory exploration: invention of mathematical concepts, invention and verification (proof) of propositions about concepts, invention of problems formulated in terms of concepts, invention and verification (proof of correctness) of algorithms, and storage and retrieval of the formulae invented and verified during this process. The system includes a collection of general as well as specific provers for various interesting domains (e. g. integers, sets, reals, tuples, etc.). More details about *Theorema* are available at `www.theorema.org`. The papers [14], [15] are surveys, and point to earlier relevant papers.

1.6 Related Research

There is a wealth of related work in program verification and a comprehensive overview on the topic may evolve into PhD thesis itself. However, in the literature there are two main types of sources, namely classical books and lecture notes, and, tools for proving program correctness automatically or semiautomatically.

Proofs exposed in classical books (e.g., [44], [46]) are very comprehensive, however, their orientation is theoretical rather than practical and mechanized. Verification there is normally a process in which the reader is required to understand the concept and perform creativity.

Furthermore, in order to perform verification, one uses the model of computation, which significantly increases the proving effort.

In contrast to classical books, computer aided verification is oriented towards verification of practical and popular types of programs, e.g., primitive recursive functions, mutual recursive functions, etc. Performing creativity there is normally not required and the aim is to speed up the verification of quantitative programs.

There are various tools for proving program correctness automatically or semiautomatically, and this is where our contribution falls into. In what follows, we only name the most significant approaches.

In the PVS system [28] the approach is type theoretical and relies on exploration of certain subtyping properties. The realization is based on Church's higher-order logic. The system is one of the most accepted and popular tools for verification.

The HOL system [32], originally constructed by Gordon, is also based on generalization of Church's higher-order logic. It mainly deals with primitive recursive functions, however, there is a very interesting work dedicated to transforming non-primitive recursive to primitive recursive functions [63]. There are various versions of HOL—in [26] one may find how it evolved over the years.

The Coq system [4] is based on a framework called "Calculus of Inductive Constructions" that is both a logic and a functional programming language. Coq has relatively big library with theories (e.g., $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, lists, etc.) where the individual proofs of the verification conditions may be carried over [5].

The KeY system [3] is not a classical verification condition generator, but a theorem prover for program logic. It is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved. KeY is manly dedicated to support object-oriented models, where for loop- and recursion-free programs, symbolic execution is performed in an automated manner.

The Sunrise system [62] contains embedding of an imperative language within the HOL theorem prover. A very specific feature of the system is that its VCG is verified as sound, and that soundness proof is checked by the HOL system. The programming language containing assignments, conditionals, and *while* commands, and also mutually recursive procedures, however, all variables have the type $\mathbb{N}$.

The ACL2 system [1] is, in our opinion, one of the most comprehensive systems for program verification. It contains a programming language, an extensible theory in a first-order logic, and a theorem prover. The language and implementation of ACL2 are built on Common Lisp. ACL2 is intended to be an industrial strength version of the Boyer-Moore theorem prover NQTHM [8], however its logical basis remains the same.

Furthermore, in [49] it is shown how a theorem prover may be used directly on the operational semantics to generate verification conditions. Thus no separate VCG is necessary, and the theorem prover can be employed both to generate and to prove the verification conditions.

The main (and also very essential) difference of our approach is that we are able to formulate conditions which are not only sufficient but also necessary in order for the program to be correct.

In contrast to other tools, which expose methods for verifying correct programs, we put special emphasize on verifying incorrect programs.

1.7 Main Contributions

The following are the new contributions:

- We define a Verification Condition Generator (VCG) which covers the most frequent types of recursive programs (including nested recursive and mutual recursive ones). These programs may operate on arbitrary domains. We prove *Soundness* and *Completeness* of the VCG and this provides a warranty that any system based on our results will be sound.
- As a distinctive feature of our method, the verification conditions do not refer to a theoretical model for program semantics or program execution, but only to the theory of the domain used in the program. This is very important for the automatic verification, because any additional theory present in the system would significantly increase the proving effort.
- We introduce here the notion of *Completeness* of a VCG as a duality of *Soundness*. It is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition will carry over to a counterexample for the given program and specification. Moreover, the failing proof gives information about the place of the bug.
- We introduce a specialized strategy for termination. The termination problem is reduced to the termination of a simplified version of the program. The conditions for the simplified versions are identical for entire classes of functional programs, thus they are highly reusable.

In order to be clear what our contributions are, throughout the thesis we use systematically the following two forms:

- “we define”, “we formulate”, etc. – that is our contribution;
- “is defined”, “is formulated”, etc – that is taken from the literature.

1.8 Organization of the thesis

In Chapter (2) we present (some part of) the theory of computability, more precisely fixpoint theory of programs. The reader is not expected to be a specialist in order to follow the presentation.

In Chapter (3) we develop a theoretical framework whose results are then used for automatic verification. We define there necessary and also sufficient conditions for a program to be totaly correct. We then construct a VCG which generates these conditions.

In a series of theorems, we prove soundness and completeness of the respective verification conditions. This implies that the truth of the verification conditions is necessary and sufficient to verify the total correctness of the program under consideration.

At the end of Chapter (3), we present various examples and we discuss rigorously possible applications of our research in domains which seem to be a bit far from program verification, namely Teaching Formal Methods to graduate students, and, Algorithm investigation for scientific computing.

We conclude with chapter (4).

Chapter 2

Logical Basis: Fixpoint Theory of Programs

This chapter is dedicated to the theoretical results we use throughout the thesis. We make a selection and present here results from the theory of computability, more precisely fixpoint theory of programs, which are well studied in the literature.

Following the fixpoint theory of programs, developed by D. Scott [23], we use the semantics of programs defined as least fixpoints of recursive operators. In that context, we define properties of programs and prove them by using the method of Scott induction [46].

Throughout this section we study (the most interesting) properties of computer science operators, e.g., compact, effective and recursive ones. However, as we are going to observe here, the notion of operator in the mathematical sense has very similar (even identical) meaning as the one in computer science.

A comprehensive overview on the theory of computability (ours is not such) starts with defining the programming paradigm (programming language), e.g., Turing Machines, Unbounded Register Machines, μ-Recursive Functions etc. We do not specify here the programming language, because, as it is well known, most of the programming paradigms have the same computing power. We assume that our programs are defined in a language (arbitrary but fixed) with the computing power of Turing Machines.

According to the Church-Turing thesis, any function which has an algorithm (or a program) is computable. The Church-Turing thesis says that any real-world computation can be translated into an equivalent computation involving a Turing machine.

Throughout this chapter, we follow the standard way of presenting results from the theory of computability, namely, all the functions are defined over the *naturals*, that is, $\mathbb{N}$ is the universe. However, this is not a restriction and the results apply to programs defined on other domains, e.g., *reals*, etc. The idea is that computers may work only with countable sets, e.g. *rationals*, *computable reals*, *lists*, etc., which themselves may be encoded by *naturals*.

For a comprehensive overview on such "lifting" we refer to [7] and [6].

The main literature sources we used during the preparation of this chapter are [61], [46] and [44]. Additionally, some notations are influenced by [57].

2.1 Prerequisites

Reading this chapter requires some basic knowledge from the theory of computability. However, we tried to keep the presentation such that it is maximally selfcontained.

Here we give a non-comprehensive list of common notions and notations which are used throughout the whole thesis.

- *Computable function.*

  Any function which has an algorithm (or a program) is computable. These functions are also called *partial recursive functions*.

- *Partial function.*

  In contrast to a total function, partial function is a function which may not be defined for some arguments. For instance, division is a partial function of two arguments. A partial function, however, is not necessarily computable.

- *Programs and functions.*

  To any program P, there is an associated function f computed by the program, defined as follows: for any x, if the program P terminates on x and computes y, then $f[x] = y$. If the program P does not terminate on x, then $f[x]$ is not defined. These functions are also called *partial recursive functions*.

- *Total function.*

  A total function is a function defined for all arguments. For instance, multiplication is a total function of two arguments. A total function, however, is not necessarily computable.

- *Recursive function.*

  Recursive functions are those which are both computable and total.

- *Termination of a program.*

  In order to express the phenomenon of program termination and non-termination, the termination symbol $\downarrow$ is introduced. One says: "The program P terminates on x." and write $P[x]\downarrow$. Complementary, one says: "The function f is defined on x." and write $f[x]\downarrow$.

  For example, the nowhere defined function Ω is defined as $\forall x\ \neg(\Omega[x]\downarrow)$. Note that Ω is computable function.

  In order to simplify the presentation, we use one new symbol $\perp$, for expressing non-termination. When we have P non-terminating on x, we write $P[x] = \perp$, and respectively $f[x] = \perp$ for expressing that f is not defined on x. In fact, one may extend the standard domain (e.g. naturals) by adding $\perp$ and develop the theory of programs in that extended domain.

- *Decidable (recursive, computable) set.*

  A set is called decidable if there is a program which terminates after a finite amount of time and decides whether or not a given element belongs to the set or not. A set which is not decidable is called undecidable.

  Alternative names expressing the same notion are *recursive set* and *computable set*.

- *Semidecidable (recursively enumerable, semicomputable) set.*

  A set is called semidecidable if there is an algorithm which terminates after a finite amount of time (and gives a positive answer) if a given element belongs to the set. When the element is not in the set that algorithm may, in general, not terminate. A set which is not semidecidable is called unsemidecidable.

  Alternative names expressing the same notion are *recursively enumerable set* and *semicomputable set.*

  Every decidable set is semidecidable, but it is not true that every semidecidable set is decidable.

  A set is decidable if and only if it is semidecidable and its complement is also semidecidable.

  Alternative names expressing the same notion are *recursively enumerable set* and *semicomputable set.*

- *Domain of a function.*

  The domain of a function is the set of all inputs for which the function is defined. It is denoted by $dom[f]$.

  A set is semidecidable if and only if there exists a partial recursive function whose domain is that set.

- *Range of a function.*

  The range of a function is the set of all values that the function takes when the argument takes values in the domain. It is denoted by $range[f]$.

  The range of a computable function is semidecidable set.

- *Graph of a function.*

  The graph of a function f is the set of all pairs x, y, where x ranges over the domain of f and $f[x] = y$. It is denoted by $graph[f]$ or G_f.

  Note that from logical point of view there is no difference between functions and their graphs.

  The graph of a computable function is a semidecidable set.

  If the graph of a function is a semidecidable set, then the function is computable.

- *Restriction of a function.*

  The restriction of a function f to a set S, denoted by $f\ \lceil_S$, is the function which for any x, $x \in S,\ f\ \lceil_S [x] = f[x]$ and is not defined for arguments out of S.

- *Gödel numbering.*

  A Gödel numbering is a function that assigns to each symbol and formula of some formal language a unique natural number called its Gödel number. The concept was introduced by Kurt Gödel for the proof of his incompleteness theorem.

  In our case, Gödel numbering is interpreted as an encoding in which a number is assigned to each program.

  In more detail, for a fixed programming language, all the programs may be ordered in a list:

$$P_0,\ P_1,\ \dots\ P_k,\ \dots,$$

such that for any given program P, its Gödel number k may be computed, and, for any natural number k, the corresponding program P_k may be effectively obtained.

- *Enumeration of all the computable functions.*

  Based upon the idea of encoding programs, the same enumeration carries over to an enumeration of computable functions. To each program P_k (taken from the list of all programs) is associated a computable function φ_k. Hence the list:

  $$\varphi_0,\ \varphi_1,\ \cdots\ \varphi_k,\ \cdots,$$

  contains all the computable functions.

Note that any computable function is associated to infinitely many programs, and hence any computable function has infinitely many indexes.

This list of prerequisite is not (and cannot be) comprehensive—it briefly describes the basic notions which could be expected in order to fully understand the proofs in this chapter.

2.2 Operators

We have seen many times the word *operator* in the context of computer science or mathematics texts. In mathematics, *operator* is a function which operates on another function, namely it is a function which acts on functions to produce other functions. In linear algebra an *operator* is a linear operator. In analysis an *operator* may be a differential or integral operator.

In general, the term *operator* in computer programming languages has the same meaning as in mathematics. This is particularly a case in functional programming languages, where an operator is also a function.

For example, *if-then-else* is an operator. A common feature of them is that they perform transformations on programs, that is, transformations on computable functions.

The following example describes a typical recursive operator. Here we use *Pascal* as programming language.

```
type nat = 0 .. maxint ;                                    (2.1)
function G (function F : nat ; X : nat) : nat ;
    var Y, I : nat;
    begin
        Y := 0;
        for I := 0 to X do
            Y := Y + F(I);
        G := Y
    end;
```

From mathematical point of view, G is an operator, because it takes a function F as an argument and returns again a function. On the other hand, G is a function (higher order) which is described by the formula:

$$G[F] \;=\; \lambda X\;.\;\sum_{I=0}^{X} F[I]\;. \tag{2.2}$$

Note that G is defined on arbitrary unary functions, not necessarily computable ones. In order to be precise, we give here the definition of what we mean by operator.

Henceforth, by $\mathfrak{F}_n$ we denote the set of all partial functions on n arguments, $(n \geq 1)$.

Definition 2.1. *Any total function* Γ *from* $\mathfrak{F}_n$ *to* $\mathfrak{F}_r$ *is an operator. The pair* (n, r) *is the type of the operator* Γ *.*

In the example (2.1), G is an operator and its type is $(1, 1)$. Another interesting feature of G is its *compactness*, namely for a given function F, and argument X, for computing the new function $G[F]$ applied to the argument X, that is, $G[F][X]$, we need to know only a finite part of F, namely the values of F in the interval $[0, \ldots, X]$, but not the whole F. The behavior of F out of that interval does not influence the result of $G[F][X]$. Moreover, one may observe that all the operators in computer science have a *compact* behavior, because (intuitively) if the computation terminates, and gives the result, it does so in finitely many steps.

In order to compare functions, we consider the case when a partial function g *dominates* other partial function f (or alternatively, f is dominated by g). Intuitively, this means that g contains all the information f does and possibly some more.

Definition 2.2. *Let f and g be partial functions: $f, g \in \mathfrak{F}_n$ for some fixed n. We say f is dominated by g, and write $f \subseteq g$ if and only if*

$$(\forall x_1, \ldots, x_n)\;(f[x_1, \ldots, x_n]\downarrow \Longrightarrow\; f[x_1, \ldots, x_n] = g[x_1, \ldots, x_n]).$$

For any n, the relation $\subseteq$ defines a partial ordering in $\mathfrak{F}_n$ with a minimal element the nowhere defined function on n arguments Ω_n, and it is so, because for any $f \in \mathfrak{F}_n$, we have $\Omega \subseteq f$.

A partial function is *finite* if its domain is a finite set. For example, Ω is finite. It is clear that, every finite function is computable. Henceforth, for a fixed n and so $\mathfrak{F}_n$, by θ (sometimes with indexes) we denote *finite* functions.

We are now ready to give the precise definition of a *compact* operator.

Definition 2.3. *Let* Γ *be an* (n, r) *operator. We say* Γ *is compact if and only if for any* $f \in \mathfrak{F}_n$*,*

$$(\forall x_1, \ldots, x_r, y)\;(\Gamma[f][x_1, \ldots, x_r] = y \;\Longrightarrow\; (\exists \theta : \theta \subseteq f)\;(\Gamma[\theta][x_1, \ldots, x_r] = y)).$$

The next interesting feature of G is its *effectiveness*, that is, if F is computable function, then $G[F]$ is also computable function. Moreover, if we know the Gödel number a of F, that is, $F = \varphi_a$, we can find effectively the Gödel number b of $G[F]$, that is, $G[F] = \varphi_b$.

Definition 2.4. *Let Γ be an (n, r) operator. We say Γ is effective if and only if there exist a recursive function h, such that,*

$$(\forall a : a \in \mathbb{N})\ (\Gamma[\varphi_a] = \varphi_{h[a]}).$$

One may observe that all the operators in computer science are effective, because (intuitively) they are described by a (piece of) program, that is, the transformation corresponding to the operator is driven by a computable function.

In order to bring this two notions together, namely compactness and effectiveness, the notion of *recursive* operator is introduced.

Definition 2.5. *Let Γ be an (n, r) operator. We say Γ is recursive if and only if Γ is compact and effective.*

After having introduced the main definitions about operators, namely compact and effective ones, in the next sections we are going to study their behavior in more detail.

2.3 Compact Operators

In this section we consider closely the properties of compact operators in order to characterize their behavior. The reader may discover some similarities between compact operators and other compact objects, e.g., compact sets, compact definitions, etc, from other branches of mathematics, however, we do not presuppose any knowledge and results taken from that branches.

In order to make the point clearer, in this section we give an example of an operator which is not compact. Before being ready to do so, we need to have some more definitions.

Definition 2.6. *Let X be a set of partial functions on n arguments, that is, $X \subseteq \mathfrak{F}_n$, and $g \in \mathfrak{F}_n$. We say g is an upper bound of X if and only if $(\forall f : f \in X)\ (f \subseteq g)$.*

Note that *supremum* of X is the least upper bound of X, where least is with respect to $\subseteq$.

Also, we need to explicitly mention that there are subsets of $\mathfrak{F}_n$, which have no upper bound. For example, the set $\{\lambda x.\ x + 1,\ \lambda x.\ 0\}$ has no upper bound.

Lemma 2.7. *Given the ascending sequence $\{f_k\}_{k\in\mathbb{N}}$ of functions:*

$$f_0 \subseteq f_1 \subseteq \ldots \subseteq f_k \subseteq \ldots$$

$f_k \in \mathfrak{F}_n$ and the graph G:

$$G = \{\langle x, y\rangle \mid (\exists k)\ (f_k[x] = y)\}.$$

The function g, defined by its graph G is the supremum of the above sequence, denoted shortly as $\{f_k\}_{k\in\mathbb{N}}$.

Proof:
First we show that G is a graph of a function. Thus, let us take two elements from G: $\langle x, y_1\rangle, \langle x, y_2\rangle \in G$. From the definition of G we have that there are k_1 and k_2, such that, $f_{k_1}[x] = y_1$ and $f_{k_2}[x] = y_2$. Let $k_1 \le k_2$, which implies that $f_{k_1} \subseteq f_{k_2}$. This means that $f_{k_1}[x] = f_{k_2}[x] = y_1 = y_2$.

Second, we show that g is an upper bound of $\{f_k\}_{k\in\mathbb{N}}$. Let us take some x and assume that $f_k[x] = y$. This means that $\langle x, y\rangle \in G$, hence $g[x] = y$. Thus we conclude that $f_k \subseteq g$, that is, g is an upper bound.

Third, we show that g is the least upper bound of $\{f_k\}_{k\in\mathbb{N}}$. Let h be an upper bound of $\{f_k\}_{k\in\mathbb{N}}$. Assume that $g[x] = y$. From the definition of g follows that there exists k, such that, $f_k[x] = y$, and thus $h[x] = y$. From this follows that $g \subseteq h$, which completes the proof of the lemma.

As we saw in the course of the proof, the graph of the supremum is the union of all the graphs of the individual functions. This gives a good reason for using the notation.

Henceforth, by $\bigcup f_k$ we denote the supremum of an ascending sequence $\{f_k\}_{k\in\mathbb{N}}$.

Lemma 2.8. *Let $\{f_k\}_{k\in\mathbb{N}}$ be an ascending sequence of functions $f_k \in \mathfrak{F}_n$, and θ be a finite function dominated by the supremum, that is, $\theta \subseteq \bigcup f_k$. Then this finite function is dominated by some element of the sequence, that is, there exists m, such that $\theta \subseteq f_m$.*

Proof:

Let $dom[\theta] = \{x_1, \dots, x_s\}$, and $\theta[x_1] = y_1$, …, $\theta[x_s] = y_s$. From $\theta \subseteq \bigcup f_k$ follows that $\bigcup f_k[x_1] = y_1$, …, $\bigcup f_k[x_s] = y_s$. From here, by the previous lemma, we obtain that there are k_1, …, k_s, such that $f_{k_1}[x_1] = y_1$, …, $f_{k_s}[x_s] = y_s$. Now, let m be the maximum of these k_1, …, k_s, and thus $f_{k_1} \subseteq f_m$, …, $f_{k_s} \subseteq f_s$. This implies $\theta \subseteq f_m$, which completes the proof of the lemma.

After having these two lemmata, we go back to the compact operators. We are ready now to give an example for an operator, which is not compact.

$$\Gamma[f][x] = \begin{cases} 1 & \Leftarrow\ f\ is\ total\ function \\ \\ 0 & \Leftarrow\ otherwise \end{cases} \tag{2.3}$$

Let f be a total function, and thus, $\Gamma[f][0] = 1$. On the other hand, $\Gamma[\theta][0] = 0$ for any finite θ, and in particular when $\theta \subseteq f$, which contradicts definition (2.3).

Compact operators behave, in some sense, like continuous functions on reals. We know that, if two continuous functions coincide for each rational number, they do so for each real as well. The following lemma formalizes this statement.

Lemma 2.9. *Let Γ_1 and Γ_2 be compact operators of type (n, r). If $\Gamma_1[\theta] = \Gamma_2[\theta]$ for each finite θ, $\theta \in \mathfrak{F}_n$, then $\Gamma_1[f] = \Gamma_2[f]$ for each f, $f \in \mathfrak{F}_n$.*

Proof.

Let x and y be arbitrary but fixed, such that $\Gamma_1[f][x] = y$. From here, by the definition of compact operators, we derive the following:

$$\Gamma_1[f][x] = y \iff (\exists\theta : \theta \subseteq f)\ (\Gamma_1[\theta][x] = y) \iff$$

$$\iff (\exists\theta : \theta \subseteq f)\ (\Gamma_2[\theta][x] = y) \iff \Gamma_2[f][x] = y,$$

which completes the proof of the lemma.

There is one more characterization of operators, they can be monotonic. Here by monotonic, we will mean the standard definition:

Definition 2.10. *Let Γ be an (n, r) operator. We say Γ is monotonic if and only if for any $f, g \in \mathfrak{F}_n$,*

$$f \subseteq g \implies \Gamma[f] \subseteq \Gamma[g].$$

With the next statement, we are going to see the relationship between compact and monotonic operators.

Lemma 2.11. *Any compact operator is monotonic as well.*

Proof.

Assume that Γ is a compact operator. Let f, g be arbitrary functions $f, g \in \mathfrak{F}_n$, such that $f \subseteq g$. Let x and y be arbitrary but fixed, such that $\Gamma[f][x] = y$. From here, by the definition of compact operators, we derive that there exists θ, $\theta \subseteq f$, such that, $\Gamma[\theta][x] = y$. Since $\theta \subseteq f$ and $f \subseteq g$ we obtain $\theta \subseteq g$ and by the definition of compact operators, we obtain: $\Gamma[g][x] = y$.

Definition 2.12. *Let Γ be an (n, r) operator. We say Γ is continuous if and only if for any ascending sequence $\{f_k\}_{k\in\mathbb{N}}$, $\Gamma[\bigcup f_k]$ is the supremum of $\{\Gamma[f_k]\}_{k\in\mathbb{N}}$ in $\mathfrak{F}_r$.*

A bit further, we will see that any continuous operator is also monotonic. This fact allows us to modify the previous definition in the following way:

Definition 2.13. *Let Γ be an (n, r) operator. We say Γ is continuous if and only if Γ is monotonic, and for any ascending sequence $\{f_k\}_{k\in\mathbb{N}}$, $\Gamma[\bigcup f_k] = \bigcup \Gamma[f_k]$.*

With the next statement, we are going to see the relationship between compact and continuous operators.

Lemma 2.14. *Any compact operator is continuous as well.*

Proof.

Assume that Γ is a compact operator. Let $\{f_k\}_{k\in\mathbb{N}}$ be an ascending sequence.

Let h and g be the supremums of the sequences $\Gamma[f_k]$ and f_k respectively, that is: $h = \bigcup \Gamma[f_k]$ and $g = \bigcup f_k$. We need to show that $\Gamma[g] = h$.

Since Γ is monotonic, we have that $\Gamma[f_k] \subseteq \Gamma[g]$ for each k. From this we conclude that $\Gamma[g]$ is an upper bound for the set $\{\Gamma[f_k]\}_{k\in\mathbb{N}}$, which implies that $h \subseteq \Gamma[g]$.

Let x and y be arbitrary but fixed, such that $\Gamma[g][x] = y$. From here, by the definition of compact operators, we derive that there exists θ, $\theta \subseteq g$, such that, $\Gamma[\theta][x] = y$. From here, by lemma (2.8), we obtain that $\theta \subseteq f_m$ for some m, that is, $\Gamma[f_m][x] = y$. From here, by the definition of h, we obtain that $h[x] = y$, which implies that $\Gamma[g] \subseteq h$, and the proof of the lemma is completed.

Lemma 2.15. *Any continuous operator is compact as well.*

Remark: Within the proof of this lemma we explicitly use the fact that our functions may have not only single arguments (that is in case the functions are from $\mathfrak{F}_1$), but also multiple arguments. Thus we write here $\overline{x}$ instead of simply x.

Proof.

Assume that Γ is continuous operator. We first show Γ is monotonic.

Let f, g be arbitrary functions $f, g \in \mathfrak{F}_n$, such that $f \subseteq g$. Consider the sequence:

$$f \subseteq g \subseteq \ldots \subseteq g \subseteq \ldots$$

From here, by lemma (2.7), we obtain that the supremum of the sequence:

$$f, g, \dots, g, \dots$$

is g. Since we assumed that Γ is continuous, by its definition we obtain that $\Gamma[g]$ is an upper bound for the set $\{\Gamma[f], \Gamma[g]\}$, which implies that $\Gamma[f] \subseteq \Gamma[g]$. The latter one shows that Γ is monotonic.

Now we show that Γ is compact.

Let f be arbitrary function $f \in \mathfrak{F}_n$, such that f is not finite. Now we construct the finite restrictions

$$\theta_1, \theta_2, \dots, \theta_k, \dots$$

of f in the following way:

$$\theta_k = f \restriction \{\overline{0}, \dots, \overline{k-1}\},$$

for any k.

Remark: Here, by $\overline{0}$ we denote the zero vector of dimension n, moreover, by $\overline{i}$ we denote the vector $(i, i, \dots, i)$ of dimension n.

Remark: The set $\{\overline{0}, \dots, \overline{k-1}\}$ contains not only the diagonal elements, but all the $n * k$ possible vectors.

We now have the following:

- $\theta_0 = \Omega_n$, that is, the first finite approximation θ_0 is the nowhere defined function. This is so, because $\theta_0 = f \restriction \emptyset$.
- for any k, the domain $dom[\theta_k]$ of θ_k has k^n elements, namely $dom[\theta_k] = \{\overline{0}, \dots \overline{k-1}\}$.
- for any k, $\theta_k[\overline{i}] = f[\overline{i}]$, for each $i \in \{\overline{0}, \dots \overline{k-1}\}$

Now, we can see that:

$$\theta_0 \subseteq \theta_1 \subseteq \dots \subseteq \theta_k \subseteq \dots,$$

and f is the supremum of $\{\theta_k\}_{k \in \mathbb{N}}$, that is $f = \bigcup \theta_k$. From this follows that $\Gamma[f] = \Gamma[\bigcup \theta_k]$, and by knowing that Γ is continuous, by the transformed definition of continuous, we obtain that $\Gamma[\bigcup \theta_k] = \bigcup \Gamma[\theta_k]$.

Now we want to show that for any $\overline{x}$ and y, we have:

$$\Gamma[f][\overline{x}] = y \iff (\exists \theta : \theta \subseteq f)\ (\Gamma[\theta][\overline{x}] = y).$$

First we show this from left to right. Let $\overline{x}$ and y be arbitrary but fixed, such that $\Gamma[f][\overline{x}] = y$. This implies that $\Gamma[\bigcup \theta_k][\overline{x}] = y$, and by lemma (2.8) we obtain that there exists k, such that $\Gamma[\theta_k][\overline{x}] = y$.

Now we show the other direction—from right to left. Let $\overline{x}$, y and θ be arbitrary but fixed, such that $\theta \subseteq f$ and $\Gamma[\theta][\overline{x}] = y$. Now, since Γ is monotonic, we obtain that $\Gamma[f][\overline{x}] = y$, which completes the proof of the lemma.

Definition 2.16. *Let Γ be an (n, n) operator. We say f, $f \in \mathfrak{F}_n$ is the minimal fixpoint of Γ if and only if:*

$$(\forall g : g \in \mathfrak{F}_n)\ (\Gamma[g] = g \implies f \subseteq g),$$

that is minimal, and it is also a fixpoint

$$\Gamma[f] = f.$$

Lemma 2.17. *Let Γ be an (n, n) monotonic operator, and f, $f \in \mathfrak{F}_n$ be a minimal solution of the inequality $\Gamma[X] \subseteq X$, that is:*

$$\Gamma[f] \subseteq f,$$

and

$$(\forall g : g \in \mathfrak{F}_n)\ (\Gamma[g] \subseteq g \implies f \subseteq g).$$

Then f is a minimal fixpoint of Γ.

Proof.

Assume that Γ is (n, r) monotonic operator, and f, $f \in \mathfrak{F}_n$ is a minimal solution of the inequality $\Gamma[X] \subseteq X$. This implies that $\Gamma[f] \subseteq f$. From here, by applying Γ on both sides of the inequality (Γ is monotonic) we obtain $\Gamma[\Gamma[f]] \subseteq \Gamma[f]$. This now means that $\Gamma[f]$ is a solution of the inequality $\Gamma[X] \subseteq X$, and since f is a minimal solution, we conclude that $f \subseteq \Gamma[f]$. Thus, we have $f = \Gamma[f]$, that is, f is a fixpoint of Γ.

Now we show that f is a minimal fixpoint. Let g be arbitrary but fixed $g \in \mathfrak{F}_n$, such that $\Gamma[g] = g$. Then we have $g \subseteq \Gamma[g]$ and thus $g \subseteq f$, which completes the proof of the lemma.

We are now going to formulate a statement, which is an instance of the famous Knaster-Tarski theorem originally formulated for lattice theory [25]. There, the formulation is as follows:

Theorem. *Let L be a complete lattice and let G, $G : L \mapsto L$ be an order-preserving function. Then the set of fixed points of G in L is also a complete lattice.*

Since complete lattices cannot be empty, the theorem in particular guarantees the existence of at least one fixpoint of G, and even the existence of a least fixpoint. In many practical cases, this is the most important implication of the theorem.

In our context, the Knaster-Tarski theorem, actually the version we formulate here, shows that any continuous operator has a minimal fixpoint. The proof itself is also very interesting, because it shows how to construct a minimal fixpoint.

Theorem 2.18. *Any continuous operator Γ of type (n, n) has a minimal fixpoint, which is a minimal solution of the inequality $\Gamma[X] \subseteq X$.*

Proof.

Assume that Γ is continuous. Now we construct the functions:

$$f_0, f_1, \ldots, f_k, \ldots \in \mathfrak{F}_n,$$

in the following way:

$$f_0 = \Omega$$

$$\ldots$$

$$f_{k+1} = \Gamma[f_k]$$

$$\ldots.$$

Since Γ is continuous, and thus monotonic as well, we obtain the following:

$$f_0 \subseteq\ f_1 \subseteq\ \dots\ \subseteq f_k \subseteq \dots\ .$$

Let f be the supremum of the sequence, that is, $f\ =\ \bigcup f_k$. We show now that f is a minimal fixpoint of Γ.

Since Γ is continuous, we have:

$$\Gamma[f] = \Gamma[\bigcup f_k] = \bigcup \Gamma[f_k] = \bigcup f_{k+1}.$$

On the other hand, f is an upper bound for $\{f_k\}$, which implies that f is an upper bound for $\{f_{k+1}\}$. From here, we conclude that $\bigcup f_{k+1} \subseteq f$ and hence $\Gamma[f] \subseteq f$.

Let g, $g \in \mathfrak{F}_n$ be arbitrary but fixed, such that $\Gamma[g] \subseteq g$. Using induction on k, we show that $f_k \subseteq g$.

When $k = 0$ we have $f_0 = \Omega$ and hence $f_0 \subseteq g$.

Assume that $f_k \subseteq g$, for some k. Since Γ is monotonic, we obtain that $\Gamma[f_k] \subseteq \Gamma[g]$, that is $f_{k+1} \subseteq \Gamma[g]$ and since $\Gamma[g] \subseteq g$, we obtain that $f_{k+1} \subseteq g$.

We obtained that f is a minimal solution of the inequality $\Gamma[X] \subseteq X$, and by lemma (2.17), we conclude that f is a minimal fixpoint of Γ, which completes the proof of the theorem.

The original Knaster-Tarski theorem is proven in 1928 by Knaster and Tarski [37], and it was in the context of Set Theory. Later, Tarski extended the theorem to increasing functions on a complete lattice and gave some applications in set theory and topology.

The Knaster-Tarski theorem, also known as Knaster-Tarski principle has an impact in theories, which seem to be far from the original invention, such as metric spaces, recursive functions etc. An interesting and relatively recent development can be found at [34].

As we see in the last section, this theorem brings us the desired induction principle, which is then used for proving properties of fixpoints.

2.4 Effective and Recursive Operators

Effective operators are those operators which can be transformed into program code.

Recursive operators are the those operators which are effective and compact. Now, the following question arises: Does the compactness restrict the class of effective operators? As we see in this section, the answer is: No, it does not!

The Myhill-Shepherdson theorem (2.21) shows that for a given effective operator E, one can construct a recursive operator Γ, such that the two operators coincide over the computable functions, that is, for each computable function φ, we have $E[\varphi] = \Gamma[\varphi]$.

Before continuing with the lemmata about operators, we first formulate one well established result in recursive function theory, namely Rice-Shapiro theorem, which is used within the proof of the lemma.

Theorem 2.19. *Let $\mathcal{A}$ be a set of computable functions on n variables. If the set of Gödel numbers (index set) $I_{\mathcal{A}}$, $I_{\mathcal{A}} = \{a \mid \varphi_a \in \mathcal{A}\}$ is semidecidable, then for any function f,*

$$f \in \mathcal{A} \iff (\exists \theta : \theta \in \mathcal{A})\,(\theta \subseteq f\ \wedge\ \theta\ is\ finite).$$

We do not provide here a proof of this theorem, however, we point the reader to the precise proof in [57].

It is interesting to point out, that the halting problem itself is a well-known example of an undecidable problem. By reducing other problems to the halting problem, showing that they are as difficult, we find that many important computational questions are undecidable. In fact, according to the Rice-Shapiro theorem, all non-trivial properties of functions are undecidable. Here by non-trivial property we mean that there are functions obeying that property and also there are other functions not obeying it.

Lemma 2.20. *Let E be an (n, r) effective operator. Then for every computable function f, $f \in \mathfrak{F}_n$ the following equivalence holds:*

For any x and y,

$$E[f][x] = y \iff (\exists \theta : \theta \subseteq f \ \wedge \ \theta \ is \ finite) \ (E[\theta][x] = y).$$

Proof.

Let h be a recursive (total and computable) function, such that for each a, we have $E[\varphi_a] = \varphi_{h[a]}$. Let also x and y be arbitrary but fixed. Now, from these x and y we construct the set $\mathcal{A}$, where $\mathcal{A} \subseteq \mathfrak{F}_n$ in the following way:

$$\mathcal{A} = \{\varphi \mid \varphi \ \ is \ \ computable \ \wedge E[\varphi][x] = y\}.$$

The index set of $\mathcal{A}$ is $I_{\mathcal{A}} = \{a \mid \varphi_a \in \mathcal{A}\}$. Now we observe the following:

for any a,

$$a \in \mathcal{A} \iff E[\varphi_a][x] = y \iff \varphi_{h[a]}[x] = y.$$

Since x and y are fixed, the set $\{a \mid \varphi_{h[a]}[x] = y\}$ is recursively enumerable. Now, let φ, $\varphi \in \mathfrak{F}_n$ be arbitrary but fixed computable function. From the definition of $\mathcal{A}$, we have that

$$\varphi \in \mathcal{A} \iff E[\varphi][x] = y.$$

On the other hand, by the Rice-Shapiro theorem, we obtain that

$$\varphi \in \mathcal{A} \iff (\exists \theta : \theta \subseteq \varphi) \ (\theta \in \mathcal{A}),$$

and from the definition of $\mathcal{A}$ follows

$$\varphi \in \mathcal{A} \iff (\exists \theta : \theta \subseteq \varphi) \ (E[\theta][x] = y).$$

The latter one implies that:

$$E[\varphi][x] = y \iff (\exists \theta : \theta \subseteq \varphi) \ (E[\theta][x] = y),$$

which completes the proof of the lemma.

The next result we want to present is the Myhill-Shepherdson theorem, initially formulated in [51]. It shows that for a given effective operator E, one can construct a recursive operator Γ, such that the two operators coincide over the computable functions.

Here we give the formal statement of Myhill-Shepherdson theorem:

Theorem 2.21. *Let E be an (n, r) effective operator. Then there exists exactly one recursive operator Γ of type (n, r), such that for any computable function φ, where $\varphi \in \mathfrak{F}_n$, we have $E[\varphi] = \Gamma[\varphi]$.*

Proof.
Assume that E is (n, r) effective operator. Let us define Γ in the following way:
for any f, x, and y,

$$\Gamma[f][x] = y \iff (\exists\theta : \theta \subseteq f \ \wedge \ \theta \; is \; finite) \; (E[\theta][x] = y).$$

We need to show the following four properties of Γ:

- Γ is an operator, that is, the definition of Γ does not introduce a contradiction;
- Γ is effective;
- Γ is compact;
- Γ is unique.

Now we show the desired properties one-by-one.

- Let f, θ_1, θ_2, x, and y be arbitrary but fixed. Assume that $\theta_1 \subseteq \theta_2 \subseteq f$. From here, by lemma (2.20), applied from right to left, where $\varphi \leftarrow \theta_2$ and $\theta \leftarrow \theta_1$, we obtain that

  $$E[\theta_1][x] = y \implies E[\theta_2][x] = y.$$

  From this we see that for any x there exists exactly one y, such that

  $$(\exists\theta : \theta \subseteq f) \; (E[\theta][x] = y).$$

  Thus, Γ is an operator.

- Let φ be computable function. By the definition of Γ, we have

  $$\Gamma[\varphi][x] = y \iff (\exists\theta : \theta \subseteq \varphi) \; (E[\theta][x] = y).$$

  On the other hand, from lemma (2.20), we have

  $$((\exists\theta : \theta \subseteq \varphi) \; (E[\theta][x] = y)) \iff (E[\varphi][x] = y).$$

  This implies that

  $$\Gamma[\varphi][x] = y \iff E[\varphi][x] = y.$$

  In other words, E and Γ coincide for any computable function and thus we conclude that Γ is effective.

- Since any finite function is computable, by the previous observation we transform the definition of Γ in the following way:

$$\Gamma[f][x] = y \iff (\exists\theta : \theta \subseteq f)\ (\Gamma[\theta][x] = y).$$

  From this we conclude that Γ is compact.

- Assume that Δ is another (n, r) compact operator, such that for any computable function φ, we have

$$E[\varphi][x] = \Delta[\varphi][x].$$

  This implies that for any finite θ

$$E[\theta][x] = \Delta[\theta][x].$$

  From here, by lemma (2.9), we obtain that $E[f] = \Delta[f]$ for any function f, $f \in \mathfrak{F}_n$. Thus Γ is unique, which completes the proof of Myhill-Shepherdson theorem.

In general, a fixpoint theorem is a result saying that a function F will have at least one fixpoint under some conditions on F. Results of this kind are some of the most useful in mathematics [36].

We are now presenting (a version of) Kleene fixpoint theorem which, actually has a very important consequences in recursion theory and more generaly in computer science.

Theorem 2.22. *Let Γ be an (n, n) recursive operator. Then the minimal fixpoint of Γ is computable function.*

Proof.

Let f be the minimal fixpoint of Γ. (It exists, due to Knaster-Tarski theorem (2.18).) From the proof of that theorem, we know how f is constructed, namely:

$$f_0 = \Omega \tag{2.4}$$
$$\dots$$
$$f_{k+1} = \Gamma[f_k]$$
$$\dots$$
$$f = \bigcup f_k.$$

We used so far the fact that Γ is compact. Now, from the fact that Γ is effective, we know that there exists a recursive function h, such that, for any a $\Gamma[\varphi_a] = \varphi_{h[a]}$. Let a_0 be a Gödel number for Ω_n. We now define a function g, in the following way:

$$g[k] = \begin{cases} a_0 & \Leftarrow \quad k = 0 \\ \\ h[g[k-1]] & \Leftarrow \quad otherwise \end{cases} \tag{2.5}$$

From the definition of g we see that $g[k]$ is a Gödel number for the computable function f_k, that is $f_k = \varphi_{g[k]}$. In other words, we have:

$$f_0 = \varphi_{a_0} = \varphi_{g[0]}$$

$$\ldots$$

$$f_k = \varphi_{g[k]}$$

$$\ldots.$$

From here, by lemma (2.7), we obtain that for any x and y:

$$f[x] = y \iff \exists\, k\; (f_k[x] \downarrow\; \wedge\, f_k[x] = y) \iff$$

$$\iff \exists\, k\; (\varphi_{g[k]}[x] \downarrow \wedge\, \varphi_{g[k]}[x] = y)\ .$$

This implies that the graph of f is a semidecidable set and thus f is computable function, which completes the proof of the theorem.

From the programming point of view, Kleene fixpoint theorem shows that computable functions are closed with respect to recursive definitions. Moreover, it gives the description of the exact solution.

2.5 Fixpoint Induction

In this section we introduce a technique for proving properties of fixpoints of recursive operators. The association which we make between fixpoints of recursive operators and recursive programs is on purpose. The correctness of this correctness is guaranteed by Kleene fixpoint theorem.

Let Γ be a compact operator of type (n, n). Let f be the minimal fixpoint of Γ. From the proof of Knaster-Tarski theorem (2.18), we know that f is constructed like in (2.4).

Let P be a property of partial functions on n variables which we want to prove for the minimal fixpoint f, that is, we want to prove $P[f]$. Assume that we have:

$$P[f_0]$$

$$(\forall g : g \in \mathfrak{F}_n)\; (P[g] \implies P[\Gamma[g]])$$

From this, we see that for each k, $P[f_k]$ holds. Now, the following question arises: May we conclude $P[f]$ holds? The following example shows that this is not always a case.

Let Γ be an operator of type $(1, 1)$, defines as:

$$\Gamma[g] = \lambda x.\ \textbf{If } x = 0 \textbf{ then } 1 \textbf{ else } x * g[x-1]. \tag{2.6}$$

The operator Γ is compact, moreover, it is also recursive. Now, let us consider the finite approximations

$$f_0 \subseteq f_1 \subseteq\ \ldots, \subseteq f_k \subseteq\ \ldots.$$

Using induction on k, one may prove that:

$$f_k = \lambda x.\ \textbf{If } x < k \textbf{ then } x! \textbf{ else } \Omega[x].$$

From this, we obtain that the minimal fixpoint f of Γ is:

$$f = \bigcup f_k = \lambda x.\, x!\ .$$

Now, let us define a property P, saying that the function is not total. The precise definition of P is:

$$P[g] \iff (\exists x)\ \neg(g[x] \downarrow)\ .$$

As we can easily see, the property P holds for all the finite approximations, and in particular $P[\Omega]$. Now we show that,

$$(\forall g : g \in \mathfrak{F}_n)\ (P[g] \implies P[\Gamma[g]]).$$

Assume that $P[g]$ holds for some arbitrary but fixed g. Assume also that $\neg(g[x] \downarrow)$ for some x. From the definition of Γ we derive:

$$\Gamma[g][x+1] = (x+1) * g[x+1-1] = (x+1) * g[x],$$

and because $\neg(g[x] \downarrow)$ we obtain $\neg(\Gamma[g][x+1] \downarrow)$. Thus we have $P[\Gamma[g]]$.

On the other hand we have $\neg P[f]$, because, $f = \lambda x.x!$ is a total function.

This example shows, that in order to perform that induction like principle for proving properties of fixpoints, we need to make some restrictions on the possible properties which are allowed to be proven that way.

Definition 2.23. *Let P be a property of functions from $\mathfrak{F}_n$. We say P is continuous if and only if for any $f_0, f_1, \ldots f_k, \ldots$ from $\mathfrak{F}_n$, such that:*

$$f_0 \subseteq f_1 \subseteq \ldots f_k \subseteq \ldots$$

$P[f_k]$ holds for any k, then $P[\bigcup f_k]$ also holds.

Now we present (and prove) some sufficient conditions for properties in order for them to be continuous.

Intuitively, the idea here is to describe a property of a partial function g, and the predicates I and O serve as a specification for it. Then proving that a partial correctness condition P holds for a given partial function g would imply that g is partially correct with respect to the predicates I and O.

Definition 2.24. *Let P be a property of functions from $\mathfrak{F}_n$. We say P is a partial correctness condition if and only if there exist predicates I and O, such that for any function g from $\mathfrak{F}_n$, we have:*

$$P[g] \iff (\forall x)\ (I[x]\ \wedge\ g[x] \downarrow \implies O[x, g[x]]).$$

The partial correctness properties are weaker than total correctness properties.

Definition 2.25. *Let P be a property of functions from $\mathfrak{F}_n$. We say P is a total correctness condition if and only if there exist predicates I and O, such that for any function g from $\mathfrak{F}_n$, we have:*

$$P[g] \iff (\forall x)\,(I[x] \implies g[x]\downarrow\, \wedge O[x, g[x]]).$$

Proving partial correctness properties is, in general, easier than proving total correctness properties. By the end of this section we define a powerful method for proving partial correctness, however, not applicable in the case of total correctness.

Lemma 2.26. *Let P be a partial correctness condition on functions from $\mathfrak{F}_n$. Then P is continuous.*

Proof.
Assume that P is a partial correctness condition on functions from $\mathfrak{F}_n$. Assume also that:

- Predicates I and O are also given;
- for any function g from $\mathfrak{F}_n$, we have:

$$P[g] \iff (\forall x)\,(I[x] \implies g[x]\downarrow\, \wedge O[x, g[x]]);$$

- the function f is defined as $f = \bigcup f_k$
- if all finite approximations $f_0, f_1, \ldots f_k, \ldots$ from $\mathfrak{F}_n$, such that:

$$f_0 \subseteq f_1 \subseteq \ldots f_k \subseteq \ldots$$

  satisfy the condition P, that is, $P[f_k]$ holds for any k, then $P[\bigcup f_k]$ also holds.

We need to show that $P[f]$ holds.
Let x be arbitrary but fixed, such that $I[x]$ and $f[x]\downarrow$. From this, by lemma (2.7), we obtain that there exists k, such that $f[x] = f_k[x]$. This implies that $f_k[x]\downarrow$ and by having $P[f_k]$, we obtain $O[x, f_k[x]]$, and hence $P[f]$, which completes the proof of the lemma.

Lemma 2.27. *Let Γ_1 and Γ_2 be continuous operators of type (n, n) and P be defined such that for any function g from $\mathfrak{F}_n$, we have:*

$$P[g] \iff \Gamma_1[g] \subseteq \Gamma_2[g].$$

Then P is continuous.

Proof.
Assume that:

$$f_0, f_1, \ldots f_k, \ldots$$

from $\mathfrak{F}_n$, are such that:

$$f_0 \subseteq f_1 \subseteq \ldots f_k \subseteq \ldots$$

and

$$\Gamma_1[f_k] \subseteq \Gamma_2[f_k]$$

for any k.

We need to show that:

$$\Gamma_1[\bigcup f_k] \subseteq \Gamma_2[\bigcup f_k].$$

Since Γ_1 and Γ_2 are continuous, we obtain that:

$$\Gamma_1[\bigcup f_k] \subseteq \bigcup \Gamma_1[f_k]$$

and

$$\Gamma_2[\bigcup f_k] \subseteq \bigcup \Gamma_2[f_k],$$

and thus it suffices to show that:

$$\bigcup \Gamma_1[f_k] \subseteq \bigcup \Gamma_2[f_k].$$

Let x and y be arbitrary but fixed and assume that $\bigcup \Gamma_1[f_k][x] = y$. Since Γ_1 is continuous, it is also monotonic. From

$$f_0 \subseteq f_1 \subseteq \dots f_k \subseteq \dots$$

we obtain:

$$\Gamma_1[f_0] \ \subseteq \ \Gamma_1[f_1] \ \subseteq \dots \ \subseteq \ \Gamma_1[f_k] \ \subseteq \dots \ .$$

Knowing that $\bigcup \Gamma[f_k][x] = y$, by lemma (2.7), we obtain that there exists m, such that $\Gamma_1[f_m][x] = y$.

On the other hand, we have $\Gamma_1[f_m] \subseteq \Gamma_1[f_m]$ and thus $\Gamma_2[f_m][x] = y$. From here, we conclude that $\bigcup \Gamma_2[f_k][x] = y$, which completes the proof of the lemma.

Lemma 2.28. *Let P_1 and P_2 be continuous properties of functions and P be defined such that for any function g from $\mathfrak{F}_n$, we have:*

$$P[g] \iff P_1[g] \ \wedge \ P_2[g].$$

Then P is continuous.

Remark: Even though this lemma is never used, in our opinion stating it here is not redundant. It (hopefully) gives some additional clarification about the matters surrounding continuous properties.

Proof.

Assume that the functions:

$$f_0, f_1, \dots f_k, \dots$$

from $\mathfrak{F}_n$, are such that:

$$f_0 \subseteq f_1 \subseteq \dots f_k \subseteq \dots$$

and $P[f_k]$ for any k.

By the definition of P we obtain that $P_1[f_k]$ for any k and $P_2[f_k]$ for any k, and hence $\forall k \ P_1[f_k]$ and $\forall k \ P_2[f_k]$. Since P_1 and P_2 are continuous, we have that $P_1[\bigcup f_k]$ and $P_2[\bigcup f_k]$.

From here, by the definition of P we obtain that $P[\bigcup f_k]$, which completes the proof of the lemma.

We are finally ready with the preparation for defining the last lemma, known as Scott rule or Scott induction rule. It gives a common way for proving continuous properties of fixpoints of continuous operators.

Lemma 2.29. *Let Γ be an (n, n) a continuous operator and P be a continuous property.*
If the following assumptions hold:

- $P[\Omega]$
- *for any function g from $\mathfrak{F}_n$, $P[g] \implies P[\Gamma[g]]$,*

then $P[f]$ also holds, where f is the minimal fixpoint of Γ.

Proof.
Assume that Γ is (n, n) continuous operator and P is continuous property. Assume also that the two assumptions from the lemma hold, namely:

$$P[\Omega],$$

and,

$$(\forall g : g \in \mathfrak{F}_n)\ (P[g] \implies P[\Gamma[g]]).$$

Let f be the minimal fixpoint of Γ. From the proof of Knaster-Tarski theorem (2.18), we know how f is constructed, namely:

$$f_0 = \Omega$$

$$\dots$$

$$f_{k+1} = \Gamma[f_k]$$

$$\dots$$

$$f = \bigcup f_k.$$

Using induction on k one may prove that the property P holds for any finite approximation f_k, that is, $\forall k\ \ P[f_k]$. From here, by knowing that P is continuous property, we obtain that $P[\bigcup f_k]$, and hence $P[f]$, which completes the proof of the lemma.

There is a commonly expressed opinion that recursive programs are difficult for verifying. The goal of this chapter is to show that this is not a case. Moreover, the overall goal of this thesis is to show that recursive programs may be verified and debugged efficiently in fully automatic manner.

2.6 Verification Using Fixpoint Induction

After having developed all the apparatus for proving properties of least fixpoints and, in particular, recursive functions defined as such, we finally consider an example, presented in this section.

In most of the books on programming, the first example to start with is the factorial function, and, we shall not make an exception here.

Example 2.30. *Let* Γ *be an operator of type* $(1, 1)$*, defines as:*

$$\Gamma[g] = \lambda x.\ \textbf{If}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * g[x - 1], \tag{2.7}$$

and f_Γ *be the least fixpoint of* Γ.

Using the Scott rule (2.29), we show that for any x, $x \in Dom[f_\Gamma]$, we have $f_\Gamma[x] = x!$. First we define a partial correctness property P as:

$$(\forall g : g \in \mathfrak{F}_1)\ (P[g] \iff (\forall x\ (g[x] \downarrow \implies g[x] = x!))).$$

Obviously, $P[\Omega]$ holds. Assume that $P[g]$ holds for some g. We have to show that $P[\Gamma[g]]$ holds. Let x be arbitrary but fixed, such that $x \in Dom[f_\Gamma]$. We have the following two cases:

- Case: $x = 0$.

  In this case we have $\Gamma[g][x] = 1 = 0! = x!$.

- Case: $x \neq 0$.

  In this case we have $\Gamma[g][x] = x * g[x - 1]$.

  Since x was chosen such that $x \in Dom[f_\Gamma]$, we have $g[x - 1] \downarrow$, which is because $\Gamma[g][x] \downarrow$ and $x * g[x - 1] \downarrow$. From here, by the induction hypothesis $P[g]$, we obtain $g[x - 1] = (x - 1)!$ and hence:

  $$\Gamma[g][x] = x * (x - 1)! = x!.$$

In both cases we showed $P[\Gamma[g]]$ holds. From here, by the Scott rule, we conclude that $P[f_\Gamma]$ holds.

The association we make between least fixpoint of an operator and a definition of a computable function is justified by Kleene fixpoint theorem. The next step, and it is the most natural one, is making association between recursive programs and computable functions. Naturally, the computable function f_Γ from the example has the following definition:

$$f_\Gamma = \lambda x.\ \textbf{If}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * f_\Gamma[x - 1],$$

which is already very similar (even identical) to the syntax used in programming languages. In order to keep us closer to the common practice, henceforth we use a slightly different syntax, namely:

$$f_\Gamma[x] = \ \textbf{If}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * f_\Gamma[x - 1],$$

which in our opinion would be the most convenient for readers used to programming.

The theoretical material presented here is used for proving several statements presented in the next chapters. We saw here how concrete instances of recursive functional programs may be proven correct.

For the purpose of computer aided verification, based on the presented here fixpoint theory of programs, we are developing a framework where program analysis can be performed algorithmically. The correctness of that performance is, however, proven once and for all by using fixpoint theory of programs.

Chapter 3

Automation of the Verification

In this chapter we develop a theoretical framework whose results are then used for automatic verification.

In the literature, there is a variety of strategies for obtaining proof rules. However, some of them have been discovered to be unsound. Verification condition generators are broadly used for automating the verification of programs, however, these VCGs have in general not themselves been proven sound. This implies that any of the programs which were verified by the help of unsound VCG may, in fact, be incorrect.

In this chapter we define necessary and also sufficient conditions for a program (of certain kind) to be totaly correct. We then construct a VCG which generates these conditions.

In a series of theorems, we prove soundness and completeness of the respective verification conditions. This implies that the truth of the verification conditions is necessary and sufficient to verify the total correctness of the program under consideration.

These proofs of soundness and completeness form the basis of an implementation of the VCG that ensures the verification of concrete programs.

3.1 Program Schemata

Considering program schemata [45] instead of concrete programs has a relatively long traditions. Early surveys on the theory of program schemata can be found in [27], and in more general splitting programs into types of programs is well studied in [53].

More generally, the use of schemata (axiom schemata, proposition schemata, problem schemata, and algorithm schemata) plays a very important role for algorithm-supported mathematical theory exploration [13], [12], [20].

Program schemata are (almost) programs where the concrete constants, functions and predicates are replaced by symbolic expressions.

When investigating program schemata instead of concrete programs, one may derive properties which concern not just one concrete program, but many similar programs, more generally—a whole class of programs—those which fit to the schema.

Moreover, for a given schema, each concrete program can be obtained from it by an instantiation which gives concrete meanings to the constant, function and predicate symbols in the schema.

Smith proposed the use of schemata for synthesis of functional programs [60]. In fact, his work spans over more than two decades, and has produced some of the more important results in practical program synthesis.

A recent result on the application of program schemata to program synthesis is available at [13], [12]. There one may find how even non-trivial algorithms, e.g., Buchberger's algorithm for Gröbner bases [10], [11] may be synthesized fully automatically starting from the specification and the schema. Details and full implementation are available at [20].

Nowadays the theory of program schemata is often involved in program transformations and proving equivalence of different schemata [22], [21]. In this thesis we do not discuss the relevance of such equivalence transformations to our approach, however, an investigation into that research direction is planned as well.

We approach the problem of program verification by studying various program schemata. When deriving necessary (and also sufficient) conditions for program correctness, we actually prove at the meta-level that for any program of that class (defined by the schema) it suffices to check only the respective verification conditions. This is very important for the automation of the whole process, because the production of the verification conditions is not expensive from the computational point of view.

The following example will give more intuition on the notions of program schemata and concrete programs. Let us consider the schema defining *simple recursive programs* :

$$F[x] = \textbf{If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R[x]]], \tag{3.1}$$

where Q is a predicate and S, C, R are auxiliary functions.

Consider also, the program $Fact$ for computing the *factorial* function:

$$Fact[n] = \textbf{If } n = 0 \textbf{ then } 1 \textbf{ else } Fact[n-1]. \tag{3.2}$$

It is now obvious, that the program $Fact$ fits to the simple recursive program schema. In order to automate the process of reasoning about programs like $Fact$ we reason at the meta-level about their schemata.

3.2 Coherent Programs

In this section we state the general principles we use for writing coherent programs with the aim of building up a non-contradictory system of verified programs. Although, these principles are not our invention (similar ideas appear in [35]), we state them here because we want to emphasize on and later formalize them. Similar to these ideas appear also in software engineering—they are called there *Design by Contract* or *Programming by Contract* [50].

We build our system such that it preserves the modularity principle, that is, each concrete implementation of a program may be replaced by another one at any time.

Building up correct programs: Firstly, we want to ensure that our system of coherent programs would contain only correct (verified) programs. This we achieve, by:

- start from basic (trustful) functions e.g. addition, multiplication, etc.;
- define each new function in terms of already known (defined previously) functions by giving its source text, the specification (input and output predicates) and prove their total correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our system. The next we want to ensure is the easy exchange (mobility) of our program implementations. This principle is usually referred as:

Modularity: Once we define the new function and prove its correctness, we "forbid" using any knowledge concerning the concrete function definition. The only knowledge we may use is the specification. This gives the possibility of easy replacement of existing functions. For example we have a powering function P, with the following program definition (implementation):

$$P[x,n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } P[x,n-1] * x \tag{3.3}$$

The specification of P is:
The domain:

$$\mathbb{D} = \mathbb{R}^2,$$

the precondition:

$$I_P[x,n] \iff n \in \mathbb{N}$$

and the postcondition:

$$O_P[x,n,P[x,n]] \iff P[x,n] = x^n.$$

Additionally, we have proven the correctness of P. Later, after using the powering function P for defining other functions, we decide to replace its definition (implementation) by another one, however, keeping the same specification. In this situation, the only thing we should do (besides preserving the name) is to prove that the new definition (implementation) of P meets the old specification.

In order to achieve the modularity, we need to ensure that when defining a new program, all the calls made to the existing (already defined) programs obey the input restrictions of that programs—we call this: *Appropriate values for the function calls.*

We now define naturally the class of coherent programs as those which obey the *appropriate values to the function calls* principle. The general definition comes in two parts: for functions defined by superposition and for functions defined by *if-then-else*.

Definition 3.1. *Let F be obtained from H, G_1, …, G_n by superposition:*

$$F[x] = H[G_1[x], \ldots, G_n[x]]. \tag{3.4}$$

The program F with the specification (I_F and O_F) is coherent *with respect to its auxiliary functions H, G_i and their specifications (I_H and O_H), (I_{G_i} and O_{G_i})*

if and only if

$$(\forall x : I_F[x]) \implies I_{G_1}[x] \ \wedge \ \ldots \ \wedge \ I_{G_n}[x] \tag{3.5}$$

and

$$(\forall x : I_F[x])\, (\forall y_1 \ldots y_n)\, (O_{G_1}[x, y_1] \ \wedge \ \cdots \wedge \ O_{G_n}[x, y_n] \implies I_H[y_1, \ldots y_n]). \tag{3.6}$$

Definition 3.2. *Let F be obtained from H, G by if-then-else:*

$$F[x] = \textbf{ If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x]. \tag{3.7}$$

The program F with the specification (I_F and O_F) is coherent *with respect to its auxiliary functions H, G and their specifications (I_H and O_H), (I_G and O_G)*

if and only if

$$(\forall x : I_F[x])\, (Q[x] \implies I_H[x]) \tag{3.8}$$

$$\wedge$$

$$(\forall x : I_F[x])\, (\neg Q[x] \implies I_G[x]).$$

Throughout this thesis we deal manly with coherent functions. As a first step of the verification process, before going to the real verification, we check if the program is coherent. It is not that programs which are not coherent are necessarily not correct. However, if we want to achieve the modularity of our system, we need to restrict to dealing with coherent programs only.

3.3 Simple Recursive Programs

In this section we study the class of simple recursive programs and we extract the purely logical conditions which are sufficient for the program correctness. These are inferred using Scott induction and induction on natural numbers in the fixpoint theory of functions and constitute a meta-theorem which is proven once for the whole class. The concrete verification conditions for each program are then provable without having to use the fixpoint theory.

Our approach is incremental and experimental: starting from simpler examples, we improve our methods such that more and more interesting problems can be solved.

We approach the correctness problem by splitting it into two parts: *partial correctness* (prove that the program satisfies the specification provided it terminates), and *termination* (prove that the program always terminates).

Simple Recursive Programs are the most used in practice and at the same time the most elementary ones, namely we look at programs of the form:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R[x]]], \tag{3.9}$$

where Q is a predicate and S, C, R are auxiliary functions. Their names are chosen such that, $S[x]$ is a “simple” function, $C[x, y]$ is a “combinator” function, and $R[x]$ is a “reduction” function. We assume that the functions S, C, and R satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_C[x, y]$, $O_C[x, y, z]$, $I_R[x]$, $O_R[x, y]$.

As un important note, we point out that functions with multiple arguments also fall into this scheme, because the arguments x, y, z could be vectors (tuples).

In practice Q may also be implemented by a program, and it may also have an input condition, but we do not want to complicate the present discussion by including this aspect, which has a special flavor.

Type (or domain) information does not appear explicitly in this formulation, however it may be included in the input conditions.

Note that the “programming language” used here contains only the construct **If–then–else** in addition to the language of first order predicate logic.

One may also use some additional restrictions on the shape of the definitions of Q, S, C, and R (e. g. that they do not contain quantifiers) in order to make the program “easy” to execute. However, this depends on the complexity of the “interpreter” (“compiler”) and does not influence the actual generation of the verification conditions. In general, the auxiliary functions may be already defined in the underlying theory, or by other programs (that includes logical terms).

3.3.1 Coherent Simple Recursive Programs

In the course of verifying a real program, as we discussed in the previous section, we first check its coherence. Just to remind, coherent programs are the ones, which have the property that each function call is applied to arguments obeying the respective input specification.

In order to perform this check, we define here the relevant verification conditions, which are derived from the definition of coherent programs (3.1) and (3.2), namely:

Definition 3.3. *Let S, C, and R be functions which satisfy their specifications (I_S, O_S), (I_C, O_C), and (I_R, O_R). Then the simple recursive program F as defined in* (3.9) *with its specification (I_F, O_F)*

is coherent with respect to S, C, R, and their specifications, if and only if the following conditions hold:

$$(\forall x : I_F[x])\ (Q[x] \implies I_S[x]) \tag{3.10}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_F[R[x]]) \tag{3.11}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_R[x]) \tag{3.12}$$

$$(\forall x, y : I_F[x])\ (\neg Q[x] \wedge O_F[R[x], y] \implies I_C[x, y]) \tag{3.13}$$

As we can see, the above conditions correspond very much to our intuition about coherent programs, namely:

- (3.10) treats the special case, that is, $Q[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S.
- (3.11) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the new input $R[x]$ must fulfill the precondition of the main function F.
- (3.12) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the input x must fulfill the precondition of the reduction function R.
- (3.13) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the input x, together with any y (where y is a possible output $F[R[x]]$) must fulfill the precondition of the combinator function C.

3.3.2 Verification Conditions and their Soundness

As we already discussed, reasoning about programs is translated into proving logical conditions. After generating these verification conditions, one has to prove them as logical formulae in the theory of the domain on which the program is defined. If all of them hold, then the program is correct with respect to its specification. The latter statement we call *Soundness* theorem, and we are now ready to define it for the class of coherent simple recursive programs.

Theorem 3.4. *Let S, C, and R be functions which satisfy their specifications (I_S, O_S), (I_C, O_C), and (I_R, O_R). Let also the simple recursive program F as defined in* (3.9) *with its specification (I_F, O_F) be coherent with respect to S, C, R, and their specifications. Then F is totaly correct with respect to (I_F, O_F) if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]]) \tag{3.14}$$

$$(\forall x, y : I_F[x])\ (\neg Q[x]\ \wedge\ O_F[R[x], y]\ \implies\ O_F[x, C[x, y]]) \tag{3.15}$$

$$(\forall x : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.16}$$

where:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F'[R[x]]. \tag{3.17}$$

As we can see, the above conditions constitute the following principle:

- (3.14) prove that the base case is correct.
- (3.15) prove that the recursive expression is correct under the assumption that the reduced call is correct.
- (3.16) prove that a simplified version F' of the initial program F terminates. Here, $\mathbb{T}$ is the logical constant true, however, any constant may serve as well.

Proof:
The proof of the *Soundness* statement is split into two major parts:

- prove partial correctness using Scott induction;
- prove termination using induction on the number of recursive calls.

First we will see that F (3.9) terminates.
Indeed, from the assumption that S, C, and R are totally correct (with respect to I_S, I_C, and I_R) by the coherence F, namely, formulae (3.10), (3.11), (3.12) and (3.13) we ensure the *termination* of the calls to the auxiliary functions S, C, and R.
Take arbitrary but fixed x and assume $I_F[x]$. From (3.16), we obtain that $F'(x) = \mathbb{T}$. We first show that there must exist a number n such that after n steps of recursive calls, the predicate Q will be satisfied, that is,

$$F'(x) = \mathbb{T} \implies (\exists n \in \mathbb{N})\ (Q[R^n[x]]), \tag{3.18}$$

where

$$R^0[x] = x$$

and

$$R^{n+1}[x] = R[R^n[x]].$$

We prove this statement by contradiction, i.e. assume:

$$F'(x) = \mathbb{T} \wedge (\forall n \in \mathbb{N})(\neg Q[R^n[x]]).$$

Now, we look at the construction of F' as being the least fixpoint of the operator F' as defined in (3.17).
Let $f_0, f_1, \ldots f_m, \ldots$ be the finite approximations of F' obtained from the nowhere defined function Ω, in the following way:

$$f_0[x] = \Omega[x]$$

$$f_{m+1}[x] = \textbf{ If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } f_m[R[x]],$$

The computable function F', corresponding to (3.17) is defined as

$$F' = \bigcup_m f_m,$$

that is, the least fixpoint of (3.17).

Since for our particular x (it was taken arbitrary but fixed) we have $F'(x) = \mathbb{T}$, there must exist a finite approximation f_m, such that:

$$f_m[x] = \mathbb{T}.$$

If $m = 0$, then $f_0[x] = \mathbb{T}$, but on the other hand, by its definition, f_0 is the nowhere defined function $f_0 = \Omega$, thus this is not a case. Hence, we conclude that $m > 0$.

From

$$(\forall n \in \mathbb{N})\ (\neg Q[R^n[x]]),$$

and in particular $\neg Q[x]$, that is when $n = 0$, by the definition of f_m we obtain:

$$f_m[x] = f_{m-1}[R[x]].$$

By repeating the same kind of reasoning m times (in fact, formally it is done by induction), we obtain that:

$$f_m[x] = f_0[R^m[x]]$$

and by its definition ($f_0 = \Omega$) we obtain:

$$f_0[R^m[x]] = \bot,$$

which contradicts $f_m[x] = \mathbb{T}$. This is the desired contradiction, and hence, we have proven (3.18).

The proof of the termination of F will be completed by proving the following statement:

$$(\exists n \in \mathbb{N})\ Q[R^n[x]] \implies F[x] \downarrow. \tag{3.19}$$

Assume $Q[R^n[x]$ for our particular x (it was taken arbitrary but fixed).

Now we consider the following two cases:

- Case 1: $n = 0$.

  Now we have $Q[x]$, and by the definition of F, we have $F[x] = S[x]$. We chose x such that $I_F[x]$, and by (3.10) we obtain that $S[x] \downarrow$ and hence $F[x] \downarrow$.

- Case 2: $n > 0$.

  Let k be the smallest number such that $(\neg Q[R^k[x]])$ and $Q[R^{k+1}[x]]$. We see now that:

  $$(\forall i \leq k)\ (I_F[R^i[x]]\ \wedge\ \neg Q[R^i[x]] \implies I_F[R^{i+1}[x]]),$$

  which follows from (3.11),

  $$(\forall i \leq k)\ (I_F[R^i[x]]\ \wedge\ \neg Q[R^i[x]] \implies I_R[R^i[x]]),$$

which follows from (3.12), and

$$(\forall i \leq k)\ (I_F[R^{i+1}[x]]\ \wedge\ \neg Q[R^i[x]] \Longrightarrow I_C[R^i[x], y]),$$

which follows from (3.13).

Now we derive that:

$$I_F[R^{k+1}[x]]\ \wedge\ Q[R^{k+1}[x]] \Longrightarrow I_S[R^{k+1}[x]]),$$

and from the definition of F we obtain:

$$F[R^{k+1}[x]] = S[R^{k+1}[x]]),$$

thus $F[R^{k+1}[x]] \downarrow$. From this, by having $I_F[R^k[x]]$, $I_R[R^k[x]]$, and $I_C[R^{k+1}[x], F[R^k[x]]]$ by the definition of F (since $\neg Q[R^k[x]]$) we obtain:

$$F[R^k[x]] = C[R^k[x], F[R^{k+1}[x]]],$$

thus $F[R^k[x]] \downarrow$. Applying the same argument k-times, we obtain $F[R^0[x]] \downarrow$, that is $F[x] \downarrow$.

We proved so far that:

$$F'(x) = \mathbb{T} \implies (\exists n \in \mathbb{N})\ (Q[R^n[x]])$$

and

$$(\exists n \in \mathbb{N})\ Q[R^n[x]] \implies F[x] \downarrow,$$

which completes the proof of termination of F.

Secondly, using Scott induction, we will show that F is partially correct with respect to its specification, namely:

$$(\forall x : I_F[x])\ (F[x] \downarrow \Longrightarrow O_F[x, F[x]]). \tag{3.20}$$

As it was broadly discussed in chapter (2), not every property is admissible and may be proven by Scott induction. However, as we already saw, properties which express partial correctness are known to be admissible.

Let us remind the definition of these properties: A property ϕ is said to be a partial correctness property if and only if there are predicates I and O, such that:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge I[a] \Longrightarrow\ O[a, f[a]])). \tag{3.21}$$

We now consider the following partial correctness property ϕ:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge I_F[a] \implies O_F[a, f[a]])).$$

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω. By the definition of ϕ we obtain:

$$\phi[\Omega] \iff (\forall a)\ (\Omega[a] \downarrow\ \wedge I_F[a] \implies O_F[a, \Omega[a]])),$$

and so, $\phi[\Omega]$ holds, since $\Omega[a]\downarrow$ never holds.

In the second step of Scott induction, we assume $\phi[f]$ holds for some f:

$$(\forall a)\ (f[a]\downarrow\ \wedge I_F[a]\implies O_F[a,f[a]]), \tag{3.22}$$

and show $\phi[f_{new}]$, where f_{new} is obtained from f by the main program (3.9) as follows:

$$f_{new}=\ \textbf{If}\ Q[x]\ \textbf{then}\ S[x]\ \textbf{else}\ C[x,f[R[x]]].$$

Now, we need to show now that for an arbitrary a,

$$f_{new}[a]\downarrow\ \wedge I_F[a]\implies O_F[a,f_{new}[a]].$$

Assume $f_{new}[a]\downarrow$ and $I_F[a]$. We have now the following two cases:

- Case 1: $Q[a]$.

  By the definition of f_{new} we obtain $f_{new}[a]=S[a]$ and since $f_{new}[a]\downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a]\downarrow$. Now using verification condition (3.14) we may conclude $O_F[a,S[a]]$ and hence $O_F[a,f_{new}[a]]$.

- Case 2: $\neg Q[a]$.

  By the definition of f_{new} we obtain $f_{new}[a]=C[a,f[R[a]]]$ and since $f_{new}[a]\downarrow$, we conclude that all the others involved in this computation must also terminate, that is: $C[a,f[R[a]]]\downarrow$, $f[R[a]]\downarrow$, and $R[a]\downarrow$.

  From $I_F[a]$, by (3.11), we obtain $I_F[R[a]]$ and, knowing that: $f[R[a]]\downarrow$ by the induction hypothesis (3.22) we obtain $O_F[R[a],f[R[a]]]$.

  Concerning the verification condition (3.15), note that all the assumptions from the left part of the implication are at hand and thus we can conclude $O_F[a,f_{new}[a]]$.

Now we conclude that the property ϕ holds for the least fixpoint of (3.9) and hence, ϕ holds for the function computed by (3.9), which completes the proof of the soundness theorem (3.4).

In the next sections we will see how this theorem may easily be extend to a more general ones, by adding an appropriate treatment for programs defined by *Case*, that is, If–then–else with several cases, etc.

3.3.3 Completeness of the Verification Conditions

Completing the notion of *Soundness*, we introduce its dual—*Completeness*—which have been introduced for the first time by the author of this thesis in [40].

As we already mentioned in the introduction, the notion of *Completeness* of a verification condition generator is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on "what is wrong".

Indeed, most books about program verification present methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

As we already mentioned, after generating the verification conditions, one has to prove them as logical formulae. If all of them hold, then the program is correct with respect to its specification—*Soundness* theorem.

Now, we formulate the *Completeness* theorem for the class of coherent simple recursive programs.

Theorem 3.5. *Let S, C, and R be functions which satisfy their specifications (I_S, O_S), (I_C, O_C), and (I_R, O_R). Let also the simple recursive program F as defined in* (3.9) *with its specification (I_F, O_F) be coherent with respect to S, C, R, and their specifications, and the output specification of F, (O_F) is functional one.*

Then if F is totaly correct with respect to (I_F, O_F) then the following verification conditions hold:

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]]) \tag{3.23}$$

$$(\forall x, y : I_F[x])\ (\neg Q[x]\ \wedge\ O_F[R[x], y]\ \implies\ O_F[x, C[x, y]]) \tag{3.24}$$

$$(\forall x : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.25}$$

where:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F'[R[x]], \tag{3.26}$$

which are the same as (3.14)*,* (3.15)*,* (3.16)*, and* (3.17) *from the Soundness theorem* (3.4).

Proof:
We assume now that:

- The functions S, C, and R are totaly correct with respect to their specifications (I_S, O_S), (I_C, O_C), and (I_R, O_R).
- The simple recursive program F as defined in (3.9) with its specification (I_F, O_F) is coherent with respect to S, C, R, and their specifications.
- The output specification of F, (O_F) is functional one, that is:

$$(\forall x : I_F[x])\ (\exists! y)\ (O_F[x, y]).$$

- The simple recursive program F as defined in (3.9) is correct with respect to its specification, that is, the total correctness formula holds:

$$(\forall x : I_F[x])\ (F[x] \downarrow \wedge\ O_F[x, F[x]]). \tag{3.27}$$

We show that (3.23), (3.24), and (3.25) hold as logical formulae.

We start now with proving (3.23) and (3.24) simultaneously.
Take arbitrary but fixed x and assume $I_F[x]$. We consider the following two cases:

- Case 1: $Q[x]$

  By the definition of F, we have $F[x] = S[x]$, and by using the correctness formula (3.27) of F, we conclude (3.23) holds. The formula (3.24) holds, because we have $Q[x]$.

- Case 2: $\neg Q[x]$

  Now, (3.23) holds. Assume y is such that $O_F[R[x], y]$. Since F is correct, we obtain that $y = F[R[x]]$, because O_F is a functional predicate.

  On the other hand, by the definition of F, we have $F[x] = C[x, F[R[x]]]$ and hence $F[x] = C[x, y]$. Again, from the correctness of F, we obtain $O_F[x, C[x, y]]$, which had to be proven.

Now, we show that the simplified version F' of the initial function F terminates. Moreover, F' terminates if F terminates. In the course of the proof, one may notice that proving $F'[x] = \mathbb{T}$ is the same as proving that F' terminates.

Take arbitrary but fixed x and assume $I_F[x]$. Since $F[x]$ terminates, we denote $F(x) = a$, for some constant a. We first show that there must exist a number n such that after n steps of recursive calls, the predicate Q will be satisfied, that is:

$$F(x) = a \Longrightarrow (\exists n \in \mathbb{N})(Q[R^n[x]]). \tag{3.28}$$

We prove this statement by contradiction, i.e. assume:

$$F(x) = a \wedge (\forall n \in \mathbb{N})(\neg Q[R^n[x]]).$$

Now, we look at the construction of F as being the least fixpoint of the operator F as defined in (3.9).

Let $f_0, f_1, \ldots f_m, \ldots$ be the finite approximations of F obtained from the nowhere defined function Ω, in the following way:

$$f_0[x] = \Omega[x]$$

$$f_{m+1}[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, f_m[R[x]]],$$

The computable function F, corresponding to (3.9) is defined as

$$F = \bigcup_m f_m,$$

that is, the least fixpoint of (3.9).

Since for our particular x (it was taken arbitrary but fixed) we have $F(x) = a$, there must exist a finite approximation f_m, such that:

$$f_m[x] = a.$$

If $m = 0$, then $f_0[x] = a$, but on the other hand, by its definition, f_0 is the nowhere defined function $f_0 = \Omega$, thus this is not a case. Hence, we conclude that $m > 0$.

From

$$(\forall n \in \mathbb{N})\ (\neg Q[R^n[x]]),$$

and in particular $\neg Q[x]$, that is when $n = 0$, by the definition of f_m we obtain:

$$f_m[x] = f_{m-1}[R[x]].$$

By repeating the same kind of reasoning m times (in fact, formally it is done by induction), we obtain that:

$$f_m[x] = f_0[R^m[x]]$$

and by its definition ($f_0 = \Omega$) we obtain:

$$f_0[R^m[x]] = \bot,$$

which contradicts $f_m[x] = a$. This is the desired contradiction, and hence, we have proven (3.28).

The proof of the termination of F', or equivalently $F'[x] = \mathbb{T}$, will be completed by proving the following statement:

$$(\exists n \in \mathbb{N})\ Q[R^n[x]] \implies F'[x] \downarrow . \tag{3.29}$$

Assume $Q[R^n[x]$ for our particular x (it was taken arbitrary but fixed).
Now we consider the following two cases:

- Case 1: $n = 0$.

  Now we have $Q[x]$, and by the definition of F', we have $F'[x] = \mathbb{T}$, and hence $F'[x] \downarrow$.

- Case 2: $n > 0$.

  Let k be the smallest number such that $(\neg Q[R^k[x]])$ and $Q[R^{k+1}[x]]$. We see now that:

  $$F'[R^{k+1}[x]] = \mathbb{T},$$

  thus $F'[R^{k+1}[x]] \downarrow$. From this, by the definition of F' (since $\neg Q[R^k[x]]$) we obtain:

  $$F'[R^k[x]] = F'[R^{k+1}[x]] = \mathbb{T},$$

  thus $F'[R^k[x]] \downarrow$. Applying the same argument k-times, we obtain $F'[R^0[x]] = \mathbb{T}$ and $F'[R^0[x]] \downarrow$, that is $F'[x] \downarrow$.

We proved so far that:

$$F'(x) = \mathbb{T} \implies (\exists n \in \mathbb{N})\ (Q[R^n[x]])$$

and

$$(\exists n \in \mathbb{N})\ Q[R^n[x]] \implies F[x] = \mathbb{T},$$

which completes our proof of the Completeness theorem.

3.3.4 Discussion

After developing a theory, it is always recommendable to give some relevant examples. However, here we only point to them—they are situated at the end of this chapter, at the section Examples (3.10). There one can see how verification of concrete programs is handled. Moreover, there is a relatively big discussion on how debugging may be done, based on the completeness.

We want to draw the attention here to the Completeness statement, namely, it is not exactly the complement of the Soundness theorem. It has one more requirement, namely, the output specification O_F to be functional one, that is, the predicate O_F defines a function (for any x, there exists exactly one y, s.t. $O_F[x, y]$. The natural question now is: *Would it be possible to omit this requirement and still preserve the completeness?* The answer is: *no!*

In order to justify our answer, we give here an example of a program f, holding the following specific properties:

- f is simple recursive program;
- f together with its specification is coherent with respect to the auxiliary functions and their specifications;
- f is totaly correct with respect to its specification;
- the output specification of f is not functional one; and
- the generated verification conditions do not hold as logical formulae.

Here is the definition of f:

$$f[n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } n * f[n-1], \tag{3.30}$$

with the specification of f, Input:

$$\forall n\ (I_f[n] \Longleftrightarrow n \in \mathbb{N}) \tag{3.31}$$

and Output:

$$\forall n, m\ (O_f[n, m] \Longleftrightarrow\ m \neq 5). \tag{3.32}$$

First we are convinced that f is simple recursive program—it fits to the scheme defined in (3.9).

Second, in order to perform the check for coherence, we generate the conditions. They are as follows:

$$(\forall n : n \in \mathbb{N})\ (n = 0\ \Longrightarrow \mathbb{T}) \tag{3.33}$$

$$(\forall n : n \in \mathbb{N})\ (n \neq 0\ \Longrightarrow (n-1) \in \mathbb{N}) \tag{3.34}$$

$$(\forall n : n \in \mathbb{N})\ (n \neq 0\ \Longrightarrow \mathbb{T}) \tag{3.35}$$

$$(\forall n, m : n \in \mathbb{N})\ (n \neq 0 \wedge m \neq 5\ \Longrightarrow \mathbb{T}) \tag{3.36}$$

The condition (3.33) is trivially true—the true constant $\mathbb{T}$ comes from the precondition of the constant function $S[x] = 1$. The condition (3.34) is also true—it expresses that the reduced input $n - 1$ should also satisfy the input condition of f. The condition (3.35) is trivially true—the true constant $\mathbb{T}$ comes from the precondition of the reduction function $R[x] = x - 1$. The condition (3.36) is trivially true—the true constant $\mathbb{T}$ comes from the precondition of the combinator function $C[x, y] = x * y$. Thus, f is coherent.

Third, we need to show that f is totaly correct with respect to its specification. We do not give here a detailed proof, but only give some hints how this could be done, namely: prove that f computes the factorial function, that is, f is totaly correct with respect to the specification of the factorial function, and then, show that this output specification of f follows from the output specification of factorial, that is:

$$(\forall n, m : n \in \mathbb{N})\ (m = n! \implies m \neq 5).$$

Fourth, we need to show that the output specification of f is not functional one. Considering (3.32), it is obvious that for any n, there are more than one m, such that $m \neq 5$. Thus (3.32) is not functional specification.

Fifth, we need to show that the generated verification conditions do not hold as logical formulae, actually, one of them is violated. They are as follows:

$$(\forall n : n \in \mathbb{N})\ (n = 0 \implies 1 \neq 5) \tag{3.37}$$

$$(\forall n, m : n \in \mathbb{N})\ (n \neq 0 \wedge m \neq 5 \implies n + m \neq 5) \tag{3.38}$$

$$(\forall n : n \in \mathrm{N})\ (f'[n] = \mathbb{T}), \tag{3.39}$$

where:

$$f'[n] = \textbf{ If } n = 0 \textbf{ then } \mathbb{T} \textbf{ else } f'[n - 1]. \tag{3.40}$$

The formulae (3.37) and (3.39) hold, however, (3.38) does not hold, and this is what we wanted to show.

3.4 Simple Recursive Programs with Multiple *Else*

This section is dedicated to the class of simple recursive programs with multiple choice *if-then-else* with zero or one recursive calls on each else branch. Shortly, we call them simple multiple *else* programs. The section is a proper extension of the previous one, as the class under study here is a proper extension of the class of simple recursive programs.

As before, we extract the purely logical conditions which are sufficient, and also necessary, for the program correctness. The proofs are similar to that from the previous section, however, they are a bit longer.

We approach the correctness problem, again, by splitting it into two parts: *partial correctness*, and *termination*. However, first we check for coherence.

Simple recursive programs with multiple else are programs of the form:

$$F[x] = \textbf{If } Q_0[x] \textbf{ then } S[x] \tag{3.41}$$

$$\begin{aligned}
&\textbf{elseif } Q_1[x] \textbf{ then } C_1[x, F[R_1[x]]]\\
&\textbf{elseif } Q_2[x] \textbf{ then } C_2[x, F[R_2[x]]]\\
&\dots\\
&\textbf{elseif } Q_n[x] \textbf{ then } C_n[x, F[R_n[x]]],
\end{aligned}$$

where Q_i are predicates and S, C_i, R_i are auxiliary functions ($S[x]$ is a "simple" function (the bottom of the recursion), $C_i[x, y]$ are "combinator" functions, and $R_i[x]$ are "reduction" functions).

We assume that the functions S, C_i, and R_i satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_{C_i}[x, y]$, $O_{C_i}[x, y, z]$, $I_{R_i}[x]$, $O_{R_i}[x, y]$.

Additionally, assume that the Q_i predicates are noncontradictory and consistent, that is:

$$Q_1 \Rightarrow \neg Q_0$$

$$\dots$$

$$Q_n \Rightarrow \neg Q_{n-1}$$

and the last one is otherwise-like:

$$Q_n = \neg Q_0 \wedge \dots \wedge \neg Q_{n-1},$$

which we do only in order to simplify the presentation.

This is, in fact, the usual semantics of most programming languages (C, Java, Lisp).

3.4.1 Coherent Simple Recursive Multiple *Else* Programs

As already discussed, before going to the real verification process, we first check if the program is coherent, that is, all function call are applied to arguments obeying the respective input specifications.

The corresponding conditions for this class of programs, which are derived from the definition of coherent programs (3.1) and (3.2), are:

Definition 3.6. *Let for all i, the functions S, C_i, and R_i be such that they satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Then the simple program with multiple else F as defined in* (3.41) *with its specification (I_F, O_F) is coherent with respect to S, C_i, R_i, and their specifications, if and only if the following conditions hold:*

$$(\forall x : I_F[x])\ (Q_0[x] \implies I_S[x]) \tag{3.42}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_F[R_1[x]]) \tag{3.43}$$

$$\dots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_F[R_n[x]]) \tag{3.44}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_{R_1}[x]) \tag{3.45}$$

$$\dots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_{R_n}[x]) \tag{3.46}$$

$$(\forall x, y : I_F[x])(Q_1[x] \wedge O_F[R_1[x], y] \implies I_{C_1}[x, y]) \tag{3.47}$$

$$\dots$$

$$(\forall x, y : I_F[x])(Q_n[x] \wedge O_F[R_n[x], y] \implies I_{C_n}[x, y]). \tag{3.48}$$

Again we see that the respective conditions for coherence correspond very much to our intuition about coherent programs, namely:

- (3.42) treats the special case, that is, $Q_0[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S.
- (3.43), …, (3.44) treat the general case, that is, $\neg Q_0[x]$, and say $Q_i[x]$ holds and recursion is applied, thus the new input $R_i[x]$ must fulfill the precondition of the main function F.
- (3.45), …, (3.46) treat the general case, that is, $\neg Q_0[x]$, and say $Q_i[x]$ holds and recursion is applied, thus the input x must fulfill the precondition of the reduction function R_i.
- (3.47), …, (3.48) treat the general case, that is, $\neg Q_0[x]$, and say $Q_i[x]$ holds and recursion is applied, thus the input x, together with any y (where y is a possible output $F[R_i[x]]$) must fulfill the precondition of the combinator function C_i.

3.4.2 Verification Conditions and their Soundness

As we already discussed, in order to be sure that a program is correctly proven to be correct, one has to formally rely on the technique used for verification. Thus we formulate here a *Soundness* theorem, for the class of coherent simple programs with multiple else.

Theorem 3.7. *Let for each i S, C_i, and R_i be functions which satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Let also the simple program with multiple else F as defined in* (3.41) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R_i, and their specifications. Then F is totaly correct with respect to (I_F, O_F) if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{3.49}$$

$$(\forall x, y : I_F[x])\ (Q_1[x] \ \wedge\ O_F[R_1[x], y] \implies O_F[x, C_1[x, y]]) \tag{3.50}$$

$$\ldots$$

$$(\forall x, y : I_F[x])\ (Q_n[x] \ \wedge\ O_F[R_n[x], y] \implies O_F[x, C_n[x, y]]) \tag{3.51}$$

$$(\forall x : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.52}$$

where:

$$\begin{aligned} F'[x] = \ & \textbf{If } Q_0[x] \textbf{ then } \mathbb{T} \\ & \textbf{elseif } Q_1[x] \textbf{ then } F'[R_1[x]] \\ & \textbf{elseif } Q_2[x] \textbf{ then } F'[R_2[x]] \\ & \ldots \\ & \textbf{elseif } Q_n[x] \textbf{ then } F'[R_n[x]]. \end{aligned} \tag{3.53}$$

The above conditions constitute the following principle:

- (3.49) prove that the base case is correct.
- (3.50), ..., (3.51) for any *else* branch, prove that the recursive expression is correct under the assumption that the reduced call is correct.
- (3.52) prove that a simplified version F' of the initial program F terminates.

Proof:
The proof of the *Soundness* statement is split into two major parts:

- prove partial correctness using Scott induction;
- prove termination using induction on the number of recursive calls.

First we will see that F (3.41) terminates.

Indeed, from the assumption that for all i: S, C_i, and R_i are totally correct (with respect to I_S, I_{C_i}, and I_{R_i}) by the coherence of F, namely, formulae (3.42), (3.43), …, (3.44), (3.45), …, (3.46), and, (3.47), …, (3.48), we ensure the *termination* of the calls to the auxiliary functions S, C_i, and R_i.

Take arbitrary but fixed x and assume $I_F[x]$. From (3.52), we obtain that $F'[x] = \mathbb{T}$.

We now show that there must exist a sequence of l in number *else* branch indexes $i_1, \ldots, i_l$ (i_j is an index of an *else* branch, i.e., $1 \leq i_j \leq n$), such that finally the bottom of the recursion will be reached, that is:

$$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \ \wedge\ \ldots\ \wedge\ Q_{i_{l-1}}[R_{i_l}[x]] \ \wedge\ Q_{i_l}[x].$$

Since the predicates Q_i are consistent and noncontradictory, for the given x, there always exists i_l, such that $Q_{i_l}[x]$, and then for the $R_{i_l}[x]$, there exists again a i_{l-1}, such that $Q_{i_{l-1}}[R_{i_l}[x]]$, etc. What we need to show is that $i_1 = 0$, that is, the Q_0 predicate will finally be fulfilled.

We prove this by contradiction, i.e. assume that for any sequence of *else* branch indexes $i_1, \ldots, i_l$:

$$\neg Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]].$$

Now, we look at the construction of F' as being the least fixpoint of the operator F' as defined in (3.53).

Let $f_0, f_1, \ldots f_m, \ldots$ be the finite approximations of F' obtained from the nowhere defined function Ω, in the following way:

$$f_0[x] = \Omega[x]$$

$$\begin{aligned} f_{m+1}[x] = \ & \textbf{If } Q_0[x] \textbf{ then } \mathbb{T} \\ & \quad \textbf{elseif } Q_1[x] \textbf{ then } f_m[R_1[x]] \\ & \quad \textbf{elseif } Q_2[x] \textbf{ then } f_m[R_2[x]] \\ & \quad \ldots \\ & \quad \textbf{elseif } Q_n[x] \textbf{ then } f_m[R_n[x]]. \end{aligned}$$

The computable function F', corresponding to (3.53) is defined as

$$F' = \bigcup_m f_m,$$

that is, the least fixpoint of (3.53).

Since for our particular x (it was taken arbitrary but fixed) we have $F'(x) = \mathbb{T}$, there must exist a finite approximation f_m, such that:

$$f_m[x] = \mathbb{T}.$$

If $m = 0$, then $f_0[x] = \mathbb{T}$, but on the other hand, by its definition, f_0 is the nowhere defined function $f_0 = \Omega$, thus this is not a case. Hence, we conclude that $m > 0$.

Assume now, that for some m_0, $f_{m_0}[x] =\perp$ and $f_{m_0}[R_1[x]] =\perp, \ldots, f_{m_0}[R_n[x]] =\perp$.

We show that $f_{m_0+1}[x] =\perp$.

Indeed, by the definition of finite approximations, we have:

$$\begin{aligned} f_{m_0+1}[x] = \ & \textbf{If } Q_0[x] \textbf{ then } \mathbb{T} \\ & \qquad\qquad \textbf{elseif } Q_1[x] \textbf{ then } f_{m_0}[R_1[x]] \\ & \qquad\qquad \textbf{elseif } Q_2[x] \textbf{ then } f_{m_0}[R_2[x]] \\ & \qquad\qquad \ldots \\ & \qquad\qquad \textbf{elseif } Q_n[x] \textbf{ then } f_{m_0}[R_n[x]]. \end{aligned}$$

From here, by having that for any sequence of *else* branch indexes $i_1, \ldots, i_l$:

$$\neg Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]],$$

and in particular $\neg Q_0[x]$, we obtain that $f_{m_0+1}[x] = \perp$.
This leads us to the desired contradiction, namely $f_m[x] = \perp$, which contradicts $f_m[x] = \mathbb{T}$.

The proof of the termination of F will be completed by proving the following statement:
If there exists a sequence of l in number *else* branch indexes $i_1, \ldots, i_l$, such that:

$$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \ \wedge \ \ldots \ \wedge \ Q_{i_{l-1}}[R_{i_l}[x]] \ \wedge \ Q_{i_l}[x],$$

then $F[x] \downarrow$.
Assume that for our particular x, which was taken arbitrary but fixed, we have:

$$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \ \wedge \ \ldots \ \wedge \ Q_{i_{l-1}}[R_{i_l}[x]] \ \wedge \ Q_{i_l}[x].$$

Now we consider the following two cases:

- Case 1: $l = 0$.

  Now we have $Q_0[x]$, and by the definition of F, we have $F[x] = S[x]$. We chose x such that $I_F[x]$, and by (3.42) we obtain that $S[x] \downarrow$ and hence $F[x] \downarrow$.

- Case 2: $l > 0$.

  Now, by following the definition of F, we have $F[x] = C_{i_l}[x, F[R_{i_l}[x]]]$. Since F is coherent, we obtain that $I_{R_{i_l}}[x]$ and $I_{C_{i_l}}[x, y]$, and hence $F[x] \downarrow$ if and only if $F[R_{i_l}[x]] \downarrow$.

  Applying the same kind of reasoning l times, and by having

  $$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]],$$

  and

  $$I_S[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]]$$

  we obtain:

  $$S[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \downarrow .$$

  On the other hand, by the definition of F, we get that:

  $$F[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] = S[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]]$$

and hence

$$F[R_{i_1}[R_{i_2}[\dots R_{i_l}[x]]]] \downarrow .$$

We also have that

$$F[R_{i_1}[R_{i_2}[\dots R_{i_l}[x]]]] \downarrow \quad iff \quad F[x] \downarrow \ .$$

From here, we conclude that $F[x] \downarrow$, which completes the proof of termination of F.

Secondly, using Scott induction, we will show that F is partially correct with respect to its specification, namely:

$$(\forall x : I_F[x])\ (F[x] \downarrow \Longrightarrow O_F[x, F[x]]). \tag{3.54}$$

As it was broadly discussed in chapter (2), not every property is admissible and may be proven by Scott induction. However, as we already saw, properties which express partial correctness are known to be admissible.

Let us remind the definition of these properties: A property ϕ is said to be a partial correctness property if and only if there are predicates I and O, such that:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge I[a] \Longrightarrow O[a, f[a]])). \tag{3.55}$$

We now consider the following partial correctness property ϕ:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge I_F[a] \Longrightarrow O_F[a, f[a]])).$$

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω. By the definition of ϕ we obtain:

$$\phi[\Omega] \iff (\forall a)\ (\Omega[a] \downarrow\ \wedge I_F[a] \Longrightarrow O_F[a, \Omega[a]])),$$

and so, $\phi[\Omega]$ holds, since $\Omega[a] \downarrow$ never holds.

In the second step of Scott induction, we assume $\phi[f]$ holds for some f:

$$(\forall a)\ (f[a] \downarrow\ \wedge I_F[a] \Longrightarrow O_F[a, f[a]]), \tag{3.56}$$

and show $\phi[f_{new}]$, where f_{new} is obtained from f by the main program (3.41) as follows:

```
$f_{new}[x] =$ **If** $Q_0[x]$ **then** $S[x]$

                  **elseif** $Q_1[x]$ **then** $C_1[x, f[R_1[x]]]$
                  **elseif** $Q_2[x]$ **then** $C_2[x, f[R_2[x]]]$
                  ...
                  **elseif** $Q_n[x]$ **then** $C_n[x, f[R_n[x]]]$,
```

Now, we need to show now that for an arbitrary a,

$$f_{new}[a] \downarrow\ \wedge I_F[a] \Longrightarrow O_F[a, f_{new}[a]].$$

Assume $f_{new}[a] \downarrow$ and $I_F[a]$. We have now the following two cases:

- Case 1: $Q_0[a]$.

  By the definition of f_{new} we obtain $f_{new}[a] = S[a]$ and since $f_{new}[a] \downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a] \downarrow$. Now using verification condition (3.49) we may conclude $O_F[a, S[a]]$ and hence $O_F[a, f_{new}[a]]$.

- Case 2: $Q_i[a]$ for some i, $1 \leq i \leq n$.

  By the definition of f_{new} we obtain $f_{new}[a] = C_i[a, f[R_i[a]]]$ and since $f_{new}[a] \downarrow$, we conclude that all the others involved in this computation must also terminate, that is: $C_i[a, f[R_i[a]]] \downarrow$, $f[R_i[a]] \downarrow$, and $R_i[a] \downarrow$.

  From $I_F[a]$, by (3.43), we obtain $I_F[R_i[a]]$ and, knowing that: $f[R_i[a]] \downarrow$ by the induction hypothesis (3.56) we obtain $O_F[R_i[a], f[R_i[a]]]$.

  Concerning the verification condition (3.50), note that all the assumptions from the left part of the implication are at hand and thus we can conclude $O_F[a, f_{new}[a]]$.

Now we conclude that the property ϕ holds for the least fixpoint of (3.41) and hence, ϕ holds for the function computed by (3.41), which completes the proof of the soundness theorem (3.4).

3.4.3 Completeness of the Verification Conditions

Completing the notion of *Soundness*, we introduce its dual—*Completeness*.

As we already mentioned, after generating the verification conditions, one has to prove them as logical formulae. If all of them hold, then the program is correct with respect to its specification—*Soundness* theorem.

Now, we formulate the *Completeness* theorem for the class of coherent simple multiple *else* programs.

Theorem 3.8. *Let for any i, the functions S, C_i, and R_i satisfy their specifications (I_S, O_S), (I_C, O_C), and (I_R, O_R). Let also the simple program with multiple* else *F as defined in* (3.41) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R_i, and their specifications, and the output specification of F, (O_F) is functional one.*

Then if F is totaly correct with respect to (I_F, O_F) then the following verification conditions hold:

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{3.57}$$

$$(\forall x, y : I_F[x])\ (Q_1[x]\ \wedge\ O_F[R_1[x], y]\ \implies\ O_F[x, C_1[x, y]]) \tag{3.58}$$

$$\ldots$$

$$(\forall x, y : I_F[x])\ (Q_n[x]\ \wedge\ O_F[R_n[x], y]\ \implies\ O_F[x, C_n[x, y]]) \tag{3.59}$$

$$(\forall x : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.60}$$

where:

$$F'[x] = \textbf{ If } Q_0[x] \textbf{ then } \mathbb{T} \tag{3.61}$$

```
        elseif $Q_1[x]$ then $F'[R_1[x]]$
        elseif $Q_2[x]$ then $F'[R_2[x]]$
        ...
        elseif $Q_n[x]$ then $F'[R_n[x]]$.
```

which are the same as (3.49)*,* (3.50)*,* (3.52)*, and* (3.53) *from the Soundness theorem* (3.4).

Proof:
We assume now that:

- For all i, the functions S, C_i, and R_i are totaly correct with respect to their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}).
- The program F as defined in (3.41) with its specification (I_F, O_F) is coherent with respect to S, C_i, R_i, and their specifications.
- The output specification of F, (O_F) is functional one, that is:

$$(\forall x : I_F[x])\ (\exists! y)\ (O_F[x, y]).$$

- The program F as defined in (3.41) is correct with respect to its specification, that is, the total correctness formula holds:

$$(\forall x : I_F[x])\ (F[x]\downarrow \wedge O_F[x, F[x]]). \tag{3.62}$$

We show that (3.57), (3.58), ..., (3.59), and (3.60) hold as logical formulae.

We start now with proving (3.57) and (3.58), ..., (3.59) simultaneously.
Take arbitrary but fixed x and assume $I_F[x]$. We consider the following two cases:

- Case 1: $Q_0[x]$

  By the definition of F, we have $F[x] = S[x]$, and by using the correctness formula (3.62) of F, we conclude (3.57) holds. The formulae (3.58), ..., (3.59) are trivial to prove, because we predicates Q are consistent and noncontradictory, and hence $\neg Q_i[x]$ for all i, $1 \leq i \leq n$.

- Case 2: $Q_i[x]$ for some i, $1 \leq i \leq n$.

  Now, the formulae (3.57) and all except one of (3.58), ..., (3.59) hold. Assume y is such that $O_F[R_i[x], y]$. Since F is correct, we obtain that $y = F[R_i[x]]$, because O_F is a functional predicate.

  On the other hand, by the definition of F, we have $F[x] = C_i[x, F[R_i[x]]]$ and hence $F[x] = C_i[x, y]$. Again, from the correctness of F, we obtain $O_F[x, C_i[x, y]]$, which had to be proven.

Now, we show that the simplified version F' of the initial function F terminates. Moreover, F' terminates if F terminates. In the course of the proof, one may notice that proving $F'[x] = \mathbb{T}$ is the same as proving that F' terminates.

Take arbitrary but fixed x and assume $I_F[x]$. Since $F[x]$ terminates, we denote $F[x] = a$, for some constant a.

We now show that there must exist a sequence of l in number *else* branch indexes $i_1, \dots, i_l$, such that finally the bottom of the recursion will be reached, that is:

$$Q_0[R_{i_1}[R_{i_2}[\dots R_{i_l}[x]]]] \ \wedge\ \dots\ \wedge\ Q_{i_{l-1}}[R_{i_l}[x]]\ \wedge\ Q_{i_l}[x].$$

Since the predicates Q_i are consistent and noncontradictory, for the given x, there always exists i_l, such that $Q_{i_l}[x]$, and then for the $R_{i_l}[x]$, there exists again a i_{l-1}, such that $Q_{i_{l-1}}[R_{i_l}[x]]$, etc. What we need to show is that $i_1 = 0$, that is, the Q_0 predicate will finally be fulfilled.

We prove this by contradiction, i.e. assume that for any sequence of *else* branch indexes $i_1, \dots, i_l$:

$$\neg Q_0[R_{i_1}[R_{i_2}[\dots R_{i_l}[x]]]].$$

Now, we look at the construction of F as being the least fixpoint of the operator F as defined in (3.41).

Let $f_0, f_1, \dots f_m, \dots$ be the finite approximations of F obtained from the nowhere defined function Ω, in the following way:

$$f_0[x] = \Omega[x]$$

$$\begin{aligned} f_{m+1}[x] = \ & \textbf{If } Q_0[x] \textbf{ then } S[x] \\ & \qquad \textbf{elseif } Q_1[x] \textbf{ then } C_1[x, f_m[R_1[x]]] \\ & \qquad \textbf{elseif } Q_2[x] \textbf{ then } C_2[x, f_m[R_2[x]]] \\ & \qquad \dots \\ & \qquad \textbf{elseif } Q_n[x] \textbf{ then } C_n[x, f_m[R_n[x]]]. \end{aligned}$$

The computable function F, corresponding to (3.41) is defined as

$$F = \bigcup_m f_m,$$

that is, the least fixpoint of (3.41).

Since for our particular x (it was taken arbitrary but fixed) we have $F(x) = a$, there must exist a finite approximation f_m, such that:

$$f_m[x] = a.$$

If $m = 0$, then $f_0[x] = a$, but on the other hand, by its definition, f_0 is the nowhere defined function $f_0 = \Omega$, thus this is not a case. Hence, we conclude that $m > 0$.

Assume now, that for some m_0, $f_{m_0}[x] =\perp$ and $f_{m_0}[R_1[x]] =\perp, \dots, f_{m_0}[R_n[x]] =\perp$.

We show that $f_{m_0+1}[x] =\perp$.

Indeed, by the definition of finite approximations, we have:

$$\begin{aligned} f_{m_0+1}[x] = \ & \textbf{If } Q_0[x] \textbf{ then } S[x] \\ & \qquad \textbf{elseif } Q_1[x] \textbf{ then } C_1[x, f_{m_0}[R_1[x]]] \\ & \qquad \textbf{elseif } Q_2[x] \textbf{ then } C_2[x, f_{m_0}[R_2[x]]] \\ & \qquad \dots \\ & \qquad \textbf{elseif } Q_n[x] \textbf{ then } C_n[x, f_{m_0}[R_n[x]]]. \end{aligned}$$

From here, by having that for any sequence of *else* branch indexes $i_1, \ldots, i_l$:

$$\neg Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]],$$

and in particular $\neg Q_0[x]$, we obtain that $f_{m_0+1}[x] = \perp$.

This leads us to the desired contradiction, namely $f_m[x] = \perp$, which contradicts $f_m[x] = a$.

The proof of the termination of F' will be completed by proving the following statement:

If there exists a sequence of l in number *else* branch indexes $i_1, \ldots, i_l$, such that:

$$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \ \wedge\ \ldots\ \wedge\ Q_{i_{l-1}}[R_{i_l}[x]] \ \wedge\ Q_{i_l}[x],$$

then $F'[x] \downarrow$.

Assume that for our particular x, which was taken arbitrary but fixed, we have:

$$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \ \wedge\ \ldots\ \wedge\ Q_{i_{l-1}}[R_{i_l}[x]] \ \wedge\ Q_{i_l}[x].$$

Now we consider the following two cases:

- Case 1: $l = 0$.

  Now we have $Q_0[x]$, and by the definition of F', we have $F[x] = \mathbb{T}$ and hence $F[x] \downarrow$.

- Case 2: $l > 0$.

  Now, by following the definition of F', we have $F'[x] = F'[R_{i_l}[x]]$. Applying the same kind of reasoning l times, we obtain:

  $$F'[x] = F'[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]],$$

  and by having:

  $$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]],$$

  we conclude that:

  $$F'[x] = F'[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] = \mathbb{T},$$

  hence, $F'[x] \downarrow$, which completes the proof of termination of F'.

We proved so far that:

If $F[x]$ terminates, then there must exist a sequence of l in number *else* branch indexes $i_1, \ldots, i_l$, such that finally the bottom of the recursion is reached, that is:

$$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \ \wedge\ \ldots\ \wedge\ Q_{i_{l-1}}[R_{i_l}[x]] \ \wedge\ Q_{i_l}[x],$$

and If there exist a sequence of l in number *else* branch indexes $i_1, \ldots, i_l$, such that:

$$Q_0[R_{i_1}[R_{i_2}[\ldots R_{i_l}[x]]]] \ \wedge\ \ldots\ \wedge\ Q_{i_{l-1}}[R_{i_l}[x]] \ \wedge\ Q_{i_l}[x],$$

then $F'[x]$ terminates.

By this we completed our proof of the Completeness theorem.

In order to illustrate the *Soundness* and the *Completeness* theorems, and the class of simple programs with multiple *else*, we refer to the example of the binary powering function (3.10.7). There, the reader finds also wrongly written versions of that program and may follow up the respective verification conditions. Moreover, there is a relatively big discussion on how debugging may be done, based on the completeness.

3.5 Fibonacci-like Schemata

In this section we study the class of Fibonacci-like recursive programs. They are like simple recursive programs, but on the *else* branch several recursive calls may appear simultaneously, and thus the Fibonacci program may be treated.

This schema is a proper extension of the simple recursive programs schema, thus this section itself is a proper extension of section (3.3).

Here we again extract the purely logical conditions which are sufficient and also necessary for the program correctness. The proofs of the *Soundness* and the *Completeness* theorems are similar to the ones presented in (3.3), however, they are a bit more complicated, due to the extension of the class of programs they treat.

Fibonacci-like Recursive Programs are programs of the form:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R_1[x]], \ldots, F[R_k[x]]]. \tag{3.63}$$

We assume that the functions S, C, and R_1, …, R_k satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_C[x, y]$, $O_C[x, y_1, \ldots, y_k, z]$, $I_{R_i}[x]$, $O_{R_i}[x, y]$.

We must admit, that this kind of programs is not the most popular in practice, however, it has specific features. Programs of this kind are considered to be (in some cases) more efficient then others, however, this is a discussion which is not relevant to our research. Note, that the so called *Divide and conquer algorithms* fit well to the Fibonacci-like schema.

3.5.1 Coherent Fibonacci-like Recursive Programs

We start up with instantiating the definitions for coherent programs (3.1) and (3.2), namely:

Definition 3.9. *Let for all i, the functions S, C, and R_i satisfy their specifications (I_S, O_S), (I_C, O_C), and (I_{R_i}, O_{R_i}). Then the program F as defined in* (3.63) *with its specification (I_F, O_F) is coherent with respect to S, C, R_i, and their specifications, if and only if the following conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \implies I_S[x]) \tag{3.64}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_F[R_1[x]]) \tag{3.65}$$

$$\ldots$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_F[R_k[x]]) \tag{3.66}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_{R_1}[x]) \tag{3.67}$$

$$\ldots$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_{R_k}[x]) \tag{3.68}$$

$$(\forall x, y_1, \ldots, y_k : I_F[x]) \tag{3.69}$$
$$(\neg Q[x] \wedge O_F[R_1[x], y] \;\wedge\; \ldots \;\wedge\; O_F[R_k[x], y] \implies I_C[x, y_1, \ldots, y_k])$$

When looking closer to the definitions, we see that our intuition about coherent programs is met again, namely:

- (3.64) treats the special case, that is, $Q[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S.
- (3.65), ..., (3.66) treat the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the new inputs $R_i[x]$ must fulfill the precondition of the main function F.
- (3.67), ..., (3.68) treat the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the input x must fulfill the precondition of all the reduction functions R_i.
- (3.69) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the input x, together with any $y_1, \ldots, y_n$ (where $y_1, \ldots, y_n$ are possible outputs from $F[R_1[x]]$, ..., $F[R_k[x]]$) must fulfill the precondition of the combinator function C.

After having defined the coherence verification conditions, we go towards defining the verification conditions for ensuring total correctness.

3.5.2 Verification Conditions and their Soundness

We introduce the verification conditions for the class of Fibonacci-like recursive programs, by providing the relevant *Soundness* theorem. The statement itself and the proof are generalization of the similar theorem (3.4) for the class of simple recursive programs.

Theorem 3.10. *Let for all i, the functions S, C, and R_i satisfy their specifications (I_S, O_S), (I_C, O_C), and (I_{R_i}, O_{R_i}). Let also the program F as defined in* (3.63) *with its specification (I_F, O_F) be coherent with respect to S, C, R_i, and their specifications. Then F is totaly correct with respect to (I_F, O_F) if the following verification conditions hold:*

$$(\forall x : I_F[x]) \; (Q[x] \implies O_F[x, S[x]]) \tag{3.70}$$

$$(\forall x, y_1, \ldots, y_k : I_F[x]) \tag{3.71}$$
$$(\neg Q[x] \;\wedge\; O_F[R_1[x], y_1] \;\wedge\; \ldots \;\wedge\; O_F[R_k[x], y_k] \implies O_F[x, C[x, y_1, \ldots, y_k]])$$

$$(\forall x : I_F[x]) \; (F'[x] = \mathbb{T}) \tag{3.72}$$

where:

$$F'[x] = \textbf{If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F'[R_1[x]] \wedge F'[R_2[x]] \;\wedge\; \ldots \;\wedge\; F'[R_k[x]]. \tag{3.73}$$

The above conditions constitute the following principle:

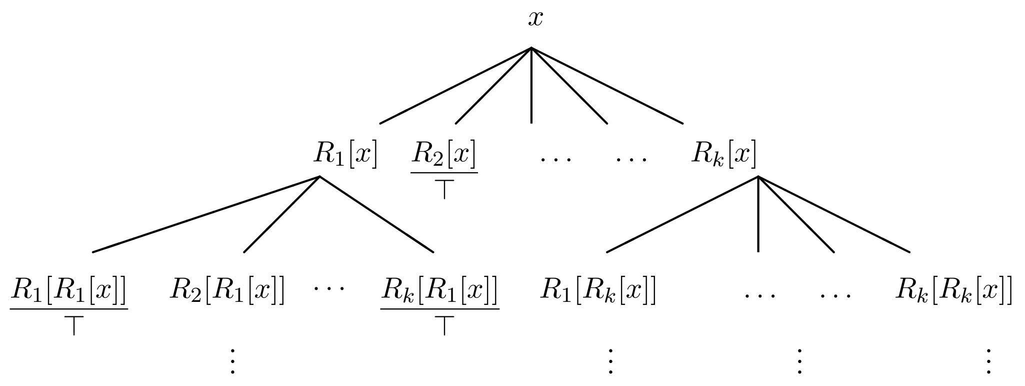
- (3.70) prove that the base case is correct.
- (3.71) prove that the recursive expression is correct under the assumption that all the reduced calls are correct.
- (3.72) prove that a simplified version F' of the initial program F terminates.

Proof:
As usual, the proof of the *Soundness* statement is split into two major parts:

- prove partial correctness using Scott induction;
- prove termination.

First we will see that F (3.63) terminates.
Indeed, from the assumption that for all i: S, C, and R_i are totally correct (with respect to I_S, I_C, and I_{R_i}) by the coherence of F, namely, formulae (3.64), (3.65), …, (3.66), (3.67), …, (3.68) and (3.69) we ensure the *termination* of the calls to the auxiliary functions S, C, and R_i.
Take arbitrary but fixed x and assume $I_F[x]$. From (3.72), we obtain that $F'(x) = \mathbb{T}$. Now we construct the recursive tree of F', starting form x, $RT_{F'}[x]$ in the following way:

- x is the root of the tree, that is, the uppermost node;
- for any node u, if $Q[u]$ holds, then stop further construction on that branch, and put the symbol $\overline{\top}$;
- for any node u, if $\neg Q[u]$ holds, then construct all the k descendent nodes $R_1[u]$, …, $R_k[u]$.



$$\neg Q[R_1[x]] \qquad Q[R_2[x]] \rightsquigarrow \overline{\top} \qquad \neg Q[R_k[x]]$$

$$Q[R_1[R_1[x]]] \rightsquigarrow \overline{\top} \quad \neg Q[R_2[R_1[x]]] \quad Q[R_k[R_1[x]]] \rightsquigarrow \overline{\top} \quad \neg Q[R_1[R_k[x]]] \quad \neg Q[R_k[R_k[x]]]$$

We first show that $RT_{F'}[x]$ is finite.

We prove this statement by contradiction, i.e. assume $RT_{F'}[x]$ is infinite. Hence, there exists an infinite path $(i_1, i_2, \ldots, i_l, \ldots)$, such that:

$$\neg Q[x] \ \wedge\ \neg Q[R_{i_1}[x]] \ \wedge \ldots \ \wedge\ \neg Q[R_{i_l}[\ldots[R_{i_1}[x]]]] \ \wedge \ldots \tag{3.74}$$

Now, we look at the construction of F' as being the least fixpoint of the operator F' as defined in (3.73).

Let $f_0, f_1, \ldots f_m, \ldots$ be the finite approximations of F' obtained from the nowhere defined function Ω, in the following way:

$$f_0[x] = \Omega[x]$$

$$f_{m+1}[x] = \ \textbf{If}\ Q[x]\ \textbf{then}\ \mathbb{T}\ \textbf{else}\ f_m[R_1[x]] \ \wedge\ \ldots\ \wedge\ f_m[R_k[x]].$$

The computable function F', corresponding to (3.73) is defined as

$$F' = \bigcup_m f_m,$$

that is, the least fixpoint of (3.73).

Since for our particular x (it was taken arbitrary but fixed) we have $F'(x) = \mathbb{T}$, there must exist a finite approximation f_m, such that:

$$f_m[x] = \mathbb{T}.$$

If $m = 0$, then $f_0[x] = \mathbb{T}$, but on the other hand, by its definition, f_0 is the nowhere defined function $f_0 = \Omega$, thus this is not a case. Hence, we conclude that $m > 0$.

From the assumption (3.74), and in particular $\neg Q[x]$, by the definition of f_m we obtain:

$$f_m[x] = f_{m-1}[R_1[x]] \ \wedge\ \ldots\ \wedge\ f_{m-1}[R_k[x]].$$

From here, and $f_m[x] = \mathbb{T}$ we obtain that:

$$f_{m-1}[R_1[x]] = \mathbb{T} \ \wedge\ \ldots\ \wedge\ f_{m-1}[R_k[x]] = \mathbb{T},$$

and hence $f_{m-1}[R_{i_1}[x]] = \mathbb{T}$.

By repeating the same kind of reasoning m times (in fact, formally it is done by induction), we obtain that:

$$f_0[R_{i_m}[\ldots[R_{i_1}[x]]]] = \mathbb{T}$$

and by its definition ($f_0 = \Omega$) we obtain:

$$f_0[R_{i_m}[\ldots[R_{i_1}[x]]]] = \perp.$$

This is the desired contradiction, and hence, we have proven that the recursive tree $RT_{F'}[x]$ is finite.

Now we continue the proof of the termination of F. For our particular x (it was taken arbitrary but fixed, $I_F[x]$), we consider the following two cases:

- Case 1: $Q[x]$.

  Now by the definition of F, we have $F[x] = S[x]$. We chose x such that $I_F[x]$, and by (3.64) we obtain that $S[x]\downarrow$ and hence $F[x]\downarrow$.

- Case 2: $\neg Q[x]$. Now, by following the definition of F, we have,

  $$F[x] = C[x, F[R_1[x], \ldots, R_k[x]]],$$

  and since F is coherent, we have $I_{R_1}[x]$, $I_{R_2}[x]$, and $I_{R_k}[x]$, and $I_C[x, y_1, \ldots, y_k]$, and thus, proving $F[x]\downarrow$, reduces to proving:

  $$F[R_1[x]\downarrow\ \wedge F[R_2[x]\downarrow\ \wedge\ \ldots\ F[R_k[x]\downarrow .$$

  We need to apply the same kind of reasoning to all the nodes of the recursive tree $RT_{F'}[x]$, and since it is finite, after unfolding finitely many times we reach the leaves where for each leaf we arrive at the Case 1. Thus, $F[x]\downarrow$.

Secondly, using Scott induction, we will show that F is partially correct with respect to its specification, namely:

$$(\forall x : I_F[x])\ (F[x]\downarrow \Longrightarrow O_F[x, F[x]]). \tag{3.75}$$

As it was broadly discussed in chapter (2), not every property is admissible and may be proven by Scott induction. However, as we already saw, properties which express partial correctness are known to be admissible.

Let us remind the definition of these properties: A property ϕ is said to be a partial correctness property if and only if there are predicates I and O, such that:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a]\downarrow\ \wedge I[a] \Longrightarrow\ O[a, f[a]])). \tag{3.76}$$

We now consider the following partial correctness property ϕ:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a]\downarrow\ \wedge I_F[a]\ \Longrightarrow\ O_F[a, f[a]])).$$

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω. By the definition of ϕ we obtain:

$$\phi[\Omega] \iff (\forall a)\ (\Omega[a]\downarrow\ \wedge I_F[a]\ \Longrightarrow\ O_F[a, \Omega[a]])),$$

and so, $\phi[\Omega]$ holds, since $\Omega[a]\downarrow$ never holds.

In the second step of Scott induction, we assume $\phi[f]$ holds for some f:

$$(\forall a)\ (f[a]\downarrow\ \wedge I_F[a]\ \Longrightarrow\ O_F[a, f[a]]), \tag{3.77}$$

and show $\phi[f_{new}]$, where f_{new} is obtained from f by the main program (3.63) as follows:

$$f_{new} = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, f[R_1[x]], f[R_2[x]], \ldots, f[R_k[x]]].$$

Now, we need to show that for an arbitrary a,

$$f_{new}[a] \downarrow \ \wedge I_F[a] \implies O_F[a, f_{new}[a]].$$

Assume $f_{new}[a] \downarrow$ and $I_F[a]$. We have now the following two cases:

- Case 1: $Q[a]$.

  By the definition of f_{new} we obtain $f_{new}[a] = S[a]$ and since $f_{new}[a] \downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a] \downarrow$. Now using verification condition (3.70) we may conclude $O_F[a, S[a]]$ and hence $O_F[a, f_{new}[a]]$.

- Case 2: $\neg Q[a]$.

  By the definition of f_{new} we obtain:

  $$f_{new}[a] = C[a, f[R_1[x]], f[R_2[x]], \ldots, f[R_k[x]]]$$

  and since $f_{new}[a] \downarrow$, we conclude that all the others involved in this computation must also terminate, that is:

  $$C[a, f[R_1[x]], f[R_2[x]], \ldots, f[R_k[x]]] \downarrow,$$

  $$f[R_1[x]] \downarrow, \ f[R_2[x]] \downarrow, \ \ldots, \ f[R_k[x]] \downarrow,$$

  $$R_1[a] \downarrow, \ R_2[a] \downarrow, \ \ldots, R_k[a] \downarrow .$$

  From $I_F[a]$, by (3.65), $\ldots$, (3.66) we obtain $I_F[R_1[a]]$, $\ldots$, $I_F[R_k[a]]$ and, knowing that: $f[R_1[a]] \downarrow, \ldots, f[R_k[a]] \downarrow$ by the induction hypothesis (3.77) we obtain:

  $$O_F[R_1[a], f[R_1[a]]], \ O_F[R_2[a], f[R_2[a]]], \ldots, O_F[R_k[a], f[R_k[a]]].$$

  Concerning the verification condition (3.71), note that all the assumptions from the left part of the implication are at hand and thus we can conclude:

  $$O_F[a, C[a, f[R_1[x]], f[R_2[x]], \ldots, f[R_k[x]]]],$$

  and thus $O_F[a, f_{new}[a]]$.

Now we conclude that the property ϕ holds for the least fixpoint of (3.63) and hence, ϕ holds for the function computed by (3.63), which completes the proof of the soundness theorem (3.10).

This completes the proof of the *Soundness* theorem. Now we proceed towards its complement, namely, the *Completeness* theorem.

3.5.3 Completeness of the Verification Conditions

Now, we formulate the *Completeness* theorem for the class of Fibonacci-like recursive programs.

Theorem 3.11. *Let for all i the functions S, C, and R_i satisfy their specifications (I_S, O_S), (I_C, O_C), and (I_{R_i}, O_{R_i}). Let also the program F as defined in* (3.63) *with its specification (I_F, O_F) be coherent with respect to S, C, R_i, and their specifications, and the output specification of F, (O_F) is functional one.*

Then if F is totaly correct with respect to (I_F, O_F) then the following verification conditions hold:

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]]) \tag{3.78}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_k : I_F[x]) \\ (\neg Q[x]\ \wedge\ O_F[R_1[x], y_1]\ \wedge\ \ldots\ \wedge\ O_F[R_k[x], y_k]\ \implies\ O_F[x, C[x, y_1, \ldots, y_k]])\end{gathered} \tag{3.79}$$

$$(\forall x : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.80}$$

where:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F'[R_1[x]]\ \wedge F'[R_2[x]]\ \wedge\ \ldots\ \wedge\ F'[R_k[x]]. \tag{3.81}$$

which are the same as (3.70), (3.71), (3.72), *and* (3.73) *from the Soundness theorem* (3.10).

Proof:
We assume now that:

- The functions S, C, and R_i are totaly correct with respect to their specifications (I_S, O_S), (I_C, O_C), and (I_{R_i}, O_{R_i}).
- The program F as defined in (3.63) with its specification (I_F, O_F) is coherent with respect to S, C, R_i, and their specifications.
- The output specification of F, (O_F) is functional one, that is:

$$(\forall x : I_F[x])\ (\exists! y)\ (O_F[x, y]).$$

- The program F as defined in (3.63) is correct with respect to its specification, that is, the total correctness formula holds:

$$(\forall x : I_F[x])\ (F[x] \downarrow \wedge\, O_F[x, F[x]]). \tag{3.82}$$

We show that (3.78), (3.79), and (3.80) hold as logical formulae.

We start now with proving (3.78) and (3.79) simultaneously.
Take arbitrary but fixed x and assume $I_F[x]$. We consider the following two cases:

- Case 1: $Q[x]$

  By the definition of F, we have $F[x] = S[x]$, and by using the correctness formula (3.82) of F, we conclude (3.78) holds. The formula (3.79) holds, because we have $Q[x]$.

- Case 2: $\neg Q[x]$

  Now, (3.78) holds. Assume $y_1, \ldots, y_k$ are such that:

$$O_F[R_1[x], y_1],\ \ldots,\ O_F[R_1[x], y_k].$$

  Since F is correct, we obtain that:

$$y_1 = F[R_1[x]],\ \ldots,\ y_k = F[R_k[x]]$$

  because O_F is a functional predicate.

  On the other hand, by the definition of F, we have:

$$F[x] = C[x, F[R_1[x]], \ldots, F[R_k[x]]]$$

  and hence $F[x] = C[x, y_1, \ldots, y_k]$. Again, from the correctness of F, we obtain:

$$O_F[x, C[x, y_1, \ldots, y_k]],$$

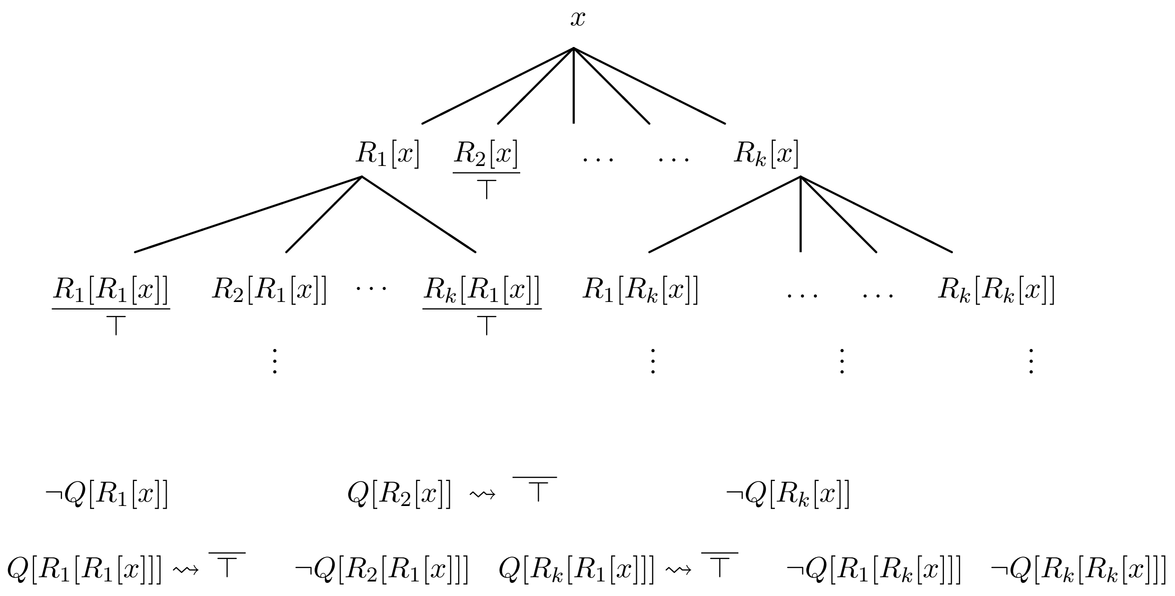
  which had to be proven.

Now, we show that the simplified version F' of the initial function F terminates. In the course of the proof, one may notice that proving $F'[x] = \mathbb{T}$ is the same as proving that F' terminates.

Take arbitrary but fixed x and assume $I_F[x]$. Since $F[x]$ terminates, we denote $F[x] = a$, for some constant a.

Now we construct the recursive tree of F, starting form x, $RT_F[x]$ in the following way:

- x is the root of the tree, that is, the uppermost node;
- for any node u, if $Q[u]$ holds, then stop further construction on that branch, and put the symbol $\top$;
- for any node u, if $\neg Q[u]$ holds, then construct all the k descendent nodes $R_1[u], \ldots, R_k[u]$.



Note that the recursive tree of F, $RT_F[x]$ is the same as the recursive tree of F', $RT_{F'}[x]$. Thus $RT_F[x]$ is finite.

Now we need to show termination of F'. For our particular x (it was taken arbitrary but fixed, $I_F[x]$), we consider the following two cases:

- Case 1: $Q[x]$.

  Now by the definition of F', we have $F[x] = \mathbb{T}$ and hence $F'[x] \downarrow$.

- Case 2: $\neg Q[x]$. Now, by following the definition of F', we have,

$$F'[x] = F'[R_1[x]] \;\wedge\; \ldots \;\wedge\; F'[R_k[x]].$$

  We need to apply the same kind of reasoning to all the nodes of the recursive tree $RT_F[x]$, and since it is finite, after unfolding finitely many times we reach the leaves where for each leaf we arrive at the Case 1. Thus, $F'[x] \downarrow$.

By this we completed our proof of the *Completeness* theorem.

In order to illustrate the *Soundness* and the *Completeness* theorems, and the class of Fibonacci-like programs, we refer to the example of Neville's algorithm, (3.264), which is broadly discussed in section (3.10.8). There, the reader is introduced to a *nonclassical* proof of a *classical* theorem from the area of numerical analysis.

3.5.4 A Note on the Termination of Fibonacci-like Programs

In this section we share some thoughts about proving termination of Fibonacci-like programs. In fact, we show that certain simplification is not possible by constructing a counterexample.

Consider the following program (already a simplified version) for computing F:

$$F[x] = \textbf{If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F[R_1] \ \wedge \ F[R_2]. \tag{3.83}$$

The question we want to ask is the following: In order to prove termination of F, would it be sufficient to prove termination of a split of F, namely to prove termination of F_1 and F_2, where:

$$F_1[x] = \textbf{If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F[R_1], \tag{3.84}$$

$$F_2[x] = \textbf{If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F[R_2]. \tag{3.85}$$

We do not want to go into discussions on *if this were so*. We give an example in order to show that this is not a case. Let us have:

$$Q[x] \Longleftrightarrow x = 0$$

$$R_1[0] = 1, \ \ R_1[1] = 2, \ \ R_1[2] = 0$$

$$R_2[0] = 2, \ \ R_2[1] = 0, \ \ R_2[2] = 1$$

$$I_F[x] \ \Longleftrightarrow \ I_{R_1}[x] \Longleftrightarrow \ I_{R_2}[x] \Longleftrightarrow \ x = 0 \ \vee \ x = 1 \ \vee \ x = 2.$$

First check if the program F is coherent. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x : x = 0 \ \vee \ x = 1 \ \vee \ x = 2) \ (x = 0 \ \Longrightarrow x = 0 \ \vee \ x = 1 \ \vee \ x = 2)$$

$$(\forall x : x = 0 \ \vee \ x = 1 \ \vee \ x = 2) \ (x \neq 0 \ \Longrightarrow R_1[x] = 0 \ \vee \ R_1[x] = 1 \ \vee \ R_1[x] = 2)$$

$$(\forall x : x = 0 \ \vee \ x = 1 \ \vee \ x = 2) \ (x \neq 0 \ \Longrightarrow R_2[x] = 0 \ \vee \ R_2[x] = 1 \ \vee \ R_2[x] = 2)$$

$$(\forall x : x = 0 \ \vee \ x = 1 \ \vee \ x = 2) \ (x \neq 0 \ \Longrightarrow x = 0 \ \vee \ x = 1 \ \vee \ x = 2)$$

$$(\forall x : x = 0 \ \vee \ x = 1 \ \vee \ x = 2) \ (x \neq 0 \ \Longrightarrow x = 0 \ \vee \ x = 1 \ \vee \ x = 2)$$

$$(\forall x : x = 0 \ \vee \ x = 1 \ \vee \ x = 2) \ (x \neq 0 \ \wedge \ \ldots \ \Longrightarrow \mathbb{T}).$$

After we are convinced that F is coherent, we first observe that its split F_1 and F_2 both terminate. For F_1, all the possibilities are:

$$F_1[0] = \mathbb{T}$$

$$F_1[1] = F_1[2] = F_1[0] = \mathbb{T}$$

$$F_1[2] = F_1[0] = \mathbb{T},$$

and for F_2:

$$F_2[0] = \mathbb{T}$$

$$F_2[1] = F_2[0] = \mathbb{T}$$

$$F_2[2] = F_2[1] = F_2[0] = \mathbb{T}.$$

Now we will show that $F[1]$ does not terminate. Indeed:

$$F[1] = F[R_1[1]] \ \wedge\ F[R_2[1]] = F[2] \ \wedge\ F[0] = F[R_1[2]] \ \wedge\ F[R_2[2]] =$$

$$= F[0] \ \wedge\ F[1] = \mathbb{T} \ \wedge\ F[1].$$

Thus, proving termination of Fibonacci-like programs requires proving termination of the whole simplified version, and, in general, no split into parts is possible.

3.6 Generalization of Fibonacci-like Schemata

This section is dedicated to the most general class of recursive programs which have no nested recursion. It is defined similarly to Fibonacci-like programs, however, with multiple choice *if-then-else* with zero or more recursive calls on each else branch.

Shortly, we call them general multiple *else* programs.

The section is a proper extension of the previous ones, as the class under study here is a proper extension of the classes of simple recursive programs, simple recursive multiple *else* programs and Fibonacci-like programs.

As before, we extract the purely logical conditions which are sufficient, and also necessary, for the program correctness. The proofs are similar to that from the previous sections, however, they are a bit longer.

We approach the correctness problem, again, by splitting it into two parts: *partial correctness*, and *termination*. However, first we check for coherence.

General recursive programs with multiple else are programs of the form:

$$F[x] = \textbf{ If } Q_0[x] \textbf{ then } S[x] \tag{3.86}$$

$$\begin{aligned}
&\textbf{elseif } Q_1[x] \textbf{ then } C_1[x, F[R_{1,1}[x]], \ldots, F[R_{1,k_1}[x]]] \\
&\textbf{elseif } Q_2[x] \textbf{ then } C_2[x, F[R_{2,1}[x]], \ldots, F[R_{2,k_2}[x]]] \\
&\ldots \\
&\textbf{elseif } Q_n[x] \textbf{ then } C_n[x, F[R_{n,1}[x]], \ldots, F[R_{n,k_n}[x]]],
\end{aligned}$$

where Q_i are predicates and $S, C_i, R_{i,j}$ are auxiliary functions ($S[x]$ is a "simple" function (the bottom of the recursion), $C_i[x, y_1, \ldots, y_{k_i}]$ are "combinator" functions, and $R_{i,j}[x]$ are "reduction" functions).

We assume that the functions S, C_i, and $R_{i,j}$ satisfy their specifications given by $I_S[x]$, $O_S[x,y]$, $I_{C_i}[x, y_1, \ldots, y_{k_i}]$, $O_{C_i}[x, y_1, \ldots, y_{k_i}, z]$, $I_{R_{i,j}}[x]$, $O_{R_{i,j}}[x,y]$.

Additionally, assume that the Q_i predicates are consistent and noncontradictory, that is:

$$Q_1 \Rightarrow \neg Q_0$$

$$\ldots$$

$$Q_n \Rightarrow \neg Q_{n-1}$$

and the last one is otherwise-like:

$$Q_n = \neg Q_0 \wedge \cdots \wedge \neg Q_{n-1},$$

which we do only in order to simplify the presentation.

3.6.1 Coherent General Recursive Multiple *Else* Programs

As already discussed, before going to the real verification process, we first check if the program is coherent, that is, all function call are applied to arguments obeying the respective input specifications.

The corresponding conditions for this class of programs, which are derived from the definition of coherent programs (3.1) and (3.2), are:

Definition 3.12. *Let for all i, j, the functions S, C_i, and $R_{i,j}$ be such that they satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and $(I_{R_{i,j}}, O_{R_{i,j}})$. Then the simple program with multiple else F as defined in* (3.86) *with its specification (I_F, O_F) is coherent with respect to S, C_i, $R_{i,j}$, and their specifications, if and only if the following conditions hold:*

$$(\forall x : I_F[x])\,(Q_0[x] \implies I_S[x]) \tag{3.87}$$

$$(\forall x : I_F[x])\,(Q_1[x] \implies I_F[R_{1,1}[x]] \,\wedge\, \ldots \,\wedge\, I_F[R_{1,k_1}[x]]) \tag{3.88}$$

$$\ldots$$

$$(\forall x : I_F[x])\,(Q_n[x] \implies I_F[R_{n,1}[x]] \,\wedge\, \ldots \,\wedge\, I_F[R_{n,k_n}[x]]) \tag{3.89}$$

$$(\forall x : I_F[x])\,(Q_1[x] \implies I_{R_{1,1}}[x] \,\wedge\, \ldots \,\wedge\, I_{R_{1,k_1}}[x]) \tag{3.90}$$

$$\ldots$$

$$(\forall x : I_F[x])\,(Q_n[x] \implies I_{R_{n,1}}[x] \,\wedge\, \ldots \,\wedge\, I_{R_{n,k_n}}[x]) \tag{3.91}$$

$$(\forall x, y_1, \ldots, y_{k_1} : I_F[x])\,(Q_1[x] \wedge O_F[R_{1,1}[x], y_1] \,\wedge\, \ldots \,\wedge O_F[R_{1,k_1}[x], y_{k_1}] \tag{3.92}$$

$$\implies$$

$$I_{C_1}[x, y_1, \ldots, y_{k_1}])$$

$$\ldots$$

$$(\forall x, y_1, \ldots, y_{k_n} : I_F[x])\,(Q_n[x] \wedge O_F[R_{n,1}[x], y_1] \,\wedge\, \ldots \,\wedge O_F[R_{n,k_n}[x], y_{k_n}] \tag{3.93}$$

$$\implies$$

$$I_{C_1}[x, y_1, \ldots, y_{k_1}])$$

Again we see that the respective conditions for coherence correspond very much to our intuition about coherent programs, namely:

- (3.87) treats the special case, that is, $Q_0[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S.
- (3.88), …, (3.89) treat the general case, that is, $\neg Q_0[x]$, and say $Q_i[x]$ holds and recursion is applied, thus all the new inputs $R_{i,1}[x]$, …, $R_{i,k_i}[x]$ must fulfill the precondition of the main function F.

- (3.90), ..., (3.91) treat the general case, that is, $\neg Q_0[x]$, and say $Q_i[x]$ holds and recursion is applied, thus the input x must fulfill the preconditions of the reduction functions $R_{i,1}, \ldots, R_{i,k_i}$.
- (3.92), ..., (3.93) treat the general case, that is, $\neg Q_0[x]$, and say $Q_i[x]$ holds and recursion is applied, thus the input x, together with any $y_1, \ldots, y_{k_i}$ (where for each j, y_j is a possible output $F[R_{i,j}[x]]$) must fulfill the precondition of the combinator function C_i.

3.6.2 Verification Conditions and their Soundness

As we already discussed, in order to be sure that a program is correctly proven to be correct, one has to formally rely on the technique used for verification. Thus we formulate here a *Soundness* theorem, for the class of coherent general programs with multiple else.

Theorem 3.13. *Let for each i, j: S, C_i, and $R_{i,j}$ be functions which satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and $(I_{R_{i,j}}, O_{R_{i,j}})$. Let also the general program with multiple else F as defined in* (3.86) *with its specification (I_F, O_F) be coherent with respect to S, C_i, $R_{i,j}$, and their specifications. Then F is totaly correct with respect to (I_F, O_F) if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{3.94}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{k_1} : I_F[x])\ (Q_1[x] \wedge O_F[R_{1,1}[x], y_1]\ \wedge\ \ldots\ \wedge O_F[R_{1,k_1}[x], y_{k_1}] \\ \implies \\ O_F[x, C_1[x, y_1, \ldots, y_{k_1}]])\end{gathered} \tag{3.95}$$

$$\ldots$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{k_n} : I_F[x])\ (Q_n[x]\ \wedge\ O_F[R_{n,1}[x], y_1]\ \wedge\ \ldots\ \wedge O_F[R_{n,k_n}[x], y_{k_n}] \\ \implies \\ O_F[x, C_n[x, y_1, \ldots, y_{k_n}]])\end{gathered} \tag{3.96}$$

$$(\forall x : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.97}$$

where:

$$\begin{aligned} F'[x] = \ & \textbf{If } Q_0[x] \textbf{ then } \mathbb{T} \\ & \textbf{elseif } Q_1[x] \textbf{ then } F'[R_{1,1}[x]]\ \wedge\ \ldots\ \wedge\ F'[R_{1,k_1}[x]] \\ & \textbf{elseif } Q_2[x] \textbf{ then } F'[R_{2,1}[x]]\ \wedge\ \ldots\ \wedge\ F'[R_{2,k_2}[x]] \\ & \ldots \\ & \textbf{elseif } Q_n[x] \textbf{ then } F'[R_{n,1}[x]]\ \wedge\ \ldots\ \wedge\ F'[R_{n,k_n}[x]]. \end{aligned} \tag{3.98}$$

The above conditions constitute the following principle:

- (3.94) prove that the base case is correct.
- (3.95), …, (3.96) for any *else* branch, prove that the recursive expression is correct under the assumption that the reduced calls are correct.
- (3.97) prove that a simplified version F' of the initial program F terminates.

Proof:

The proof of the *Soundness* statement is split into two major parts:

- prove partial correctness using Scott induction;
- prove termination.

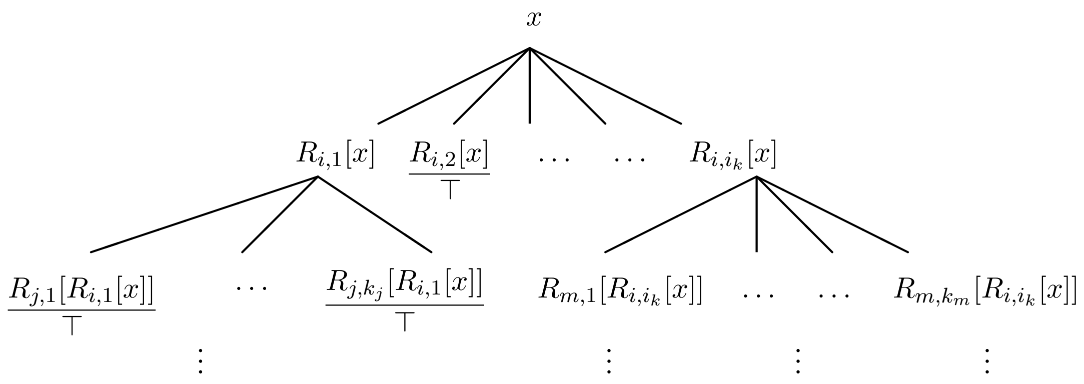
First we will see that F (3.86) terminates.

Indeed, from the assumption that for all i: S, C_i, and $R_{i,j}$ are totally correct (with respect to I_S, I_{C_i}, and $I_{R_{i,j}}$) by the coherence of F, namely, formulae (3.87), (3.88), …, (3.89), (3.90), …, (3.91), and, (3.92), …, (3.93), we ensure the *termination* of the calls to the auxiliary functions S, C_i, and $R_{i,j}$.

Take arbitrary but fixed x and assume $I_F[x]$. From (3.97), we obtain that $F'[x] = \mathbb{T}$.

Now we construct the recursive tree of F', starting form x, $RT_{F'}[x]$ in the following way:

- x is the root of the tree, that is, the uppermost node;
- for any node u, if $Q_0[u]$ holds, then stop further construction on that branch, and put the symbol $\overline{\top}$;
- for any node u, if $Q_i[u]$ holds, for some $i \neq 0$, then construct all the k_i descendent nodes $R_{i,1}[u], \ldots, R_{i,k_i}[u]$.


x
$R_{i,1}[x]$
$\frac{R_{i,2}[x]}{\top}$
$\dots \quad \dots$
$R_{i,i_k}[x]$
$\frac{R_{j,1}[R_{i,1}[x]]}{\top}$
$\dots$
$\frac{R_{j,k_j}[R_{i,1}[x]]}{\top}$
$R_{m,1}[R_{i,i_k}[x]]$
$\dots \quad \dots$
$R_{m,k_m}[R_{i,i_k}[x]]$
$\vdots$

$Q_i[x]$

$Q_j[R_{i,1}[x]] \qquad Q_0[R_{i,2}[x]] \rightsquigarrow \overline{\top} \qquad Q_m[R_{i,i_k}[x]]$

$Q_0[R_{j,1}[R_{i,1}[x]]] \rightsquigarrow \overline{\top} \qquad Q_0[R_{j,k_j}[R_{i,1}[x]]] \rightsquigarrow \overline{\top}$

We first show that $RT_{F'}[x]$ is finite.

We prove this statement by contradiction, i.e. assume $RT_{F'}[x]$ is infinite. Hence, there exists an infinite path $(\langle i_1, j_1\rangle, \langle i_2, j_2\rangle, \dots, \langle i_l, j_l\rangle \dots)$, such that:

$$
\begin{aligned}
&\neg Q_0[x] \quad \text{but:} \quad Q_{i_1}[x] \\
&\neg Q_0[R_{i_1,j_1}[x]] \quad \text{but:} \quad Q_{i_2}[R_{i_1,j_1}[x]] \\
&\dots \\
&\neg Q_0[R_{i_l,j_l}[\dots[R_{i_1,j_1}[x]]]] \quad \text{but:} \quad Q_{i_{l+1}}[R_{i_l,j_l}[\dots[R_{i_1,j_1}[x]]]] \\
&\dots .
\end{aligned}
\tag{3.99}
$$

Now, we look at the construction of F' as being the least fixpoint of the operator F' as defined in (3.98).

Let $f_0, f_1, \dots f_m, \dots$ be the finite approximations of F' obtained from the nowhere defined function Ω, in the following way:

$$f_0[x] = \Omega[x]$$

$$
\begin{aligned}
f_{m+1}[x] = \ &\textbf{If } Q_0[x] \textbf{ then } \mathbb{T} \\
&\quad \textbf{elseif } Q_1[x] \textbf{ then } f_m[R_{1,1}[x]] \wedge \dots \wedge f_m[R_{1,k_1}[x]] \\
&\quad \textbf{elseif } Q_2[x] \textbf{ then } f_m[R_{2,1}[x]] \wedge \dots \wedge f_m[R_{2,k_2}[x]] \\
&\quad \dots \\
&\quad \textbf{elseif } Q_n[x] \textbf{ then } f_m[R_{n,1}[x]] \wedge \dots \wedge f_m[R_{n,k_n}[x]].
\end{aligned}
$$

The computable function F', corresponding to (3.98) is defined as

$$F' = \bigcup_m f_m,$$

that is, the least fixpoint of (3.98).

Since for our particular x (it was taken arbitrary but fixed) we have $F'(x) = \mathbb{T}$, there must exist a finite approximation f_m, such that:

$$f_m[x] = \mathbb{T}.$$

If $m = 0$, then $f_0[x] = \mathbb{T}$, but on the other hand, by its definition, f_0 is the nowhere defined function $f_0 = \Omega$, thus this is not a case. Hence, we conclude that $m > 0$.

From the assumption (3.99), and in particular $\neg Q_0[x]$ and $Q_{i_1}[x]$, by the definition of f_m we obtain:

$$f_m[x] = f_{m-1}[R_{i_1,1}[x]] \ \wedge \ \dots \ \wedge \ f_{m-1}[R_{i_1,k_{i_1}}[x]].$$

From here, and $f_m[x] = \mathbb{T}$ we obtain that:

$$f_{m-1}[R_{i_1,1}[x]] = \mathbb{T} \ \wedge \ \dots \ \wedge \ f_{m-1}[R_{i_1,k_{i_1}}[x]] = \mathbb{T},$$

and hence $f_{m-1}[R_{i_1,j_1}[x]] = \mathbb{T}$.

By repeating the same kind of reasoning m times (in fact, formally it is done by induction), we obtain that:

$$f_0[R_{i_m,j_m}[\dots[R_{i_1,j_m}[x]]]] = \mathbb{T}$$

and by its definition ($f_0 = \Omega$) we obtain:

$$f_0[R_{i_m,j_m}[\dots[R_{i_1,j_1}[x]]]] = \bot.$$

This is the desired contradiction, and hence, we have proven that the recursive tree $RT_{F'}[x]$ is finite.

Now we continue the proof of the termination of F. We prove this statement by contradiction, i.e. assume $RT_{F'}[x]$ is finite and $F[x] = \bot$.

For our particular x (it was taken arbitrary but fixed, $I_F[x]$), we consider the following two cases:

- Case 1: $Q_0[x]$.

  Now by the definition of F, we have $F[x] = S[x]$. We chose x such that $I_F[x]$, and by (3.87) we obtain that $S[x] \downarrow$ and hence $F[x] \downarrow$ and thus we obtain a contradiction.

- Case 2: $\neg Q_0[x]$, and assume $Q_{i_1}[x]$. Now, by following the definition of F, we have,

  $$F[x] = C_{i_1}[x, F[R_{i_1,1}[x], \dots, R_{i_1,k_{i_1}}[x]]],$$

  and since F is coherent, we have $I_{R_{i_1,1}}[x]$, $I_{R_{i_1,2}}[x]$, and $I_{R_{i_1,k_{i_1}}}[x]$, and $I_C[x, y_1, \dots, y_{k_{i_1}}]$, and thus, there exist j_1, such that $R_{i_1,j_1}[x] = \bot$.

Applying the same kind of reasoning we obtain the infinite path $(\langle i_1, j_1\rangle, \langle i_2, j_2\rangle, \ldots, \langle i_l, j_l\rangle \ldots)$, (we already considered the same in (3.99)), that is:

$\neg Q_0[x]$ but: $Q_{i_1}[x]$

$\neg Q_0[R_{i_1,j_1}[x]]$ but: $Q_{i_2}[R_{i_1,j_1}[x]]$

$\ldots$

$\neg Q_0[R_{i_l,j_l}[\ldots [R_{i_1,j_1}[x]]]]$ but: $Q_{i_{l+1}}[R_{i_l,j_l}[\ldots [R_{i_1,j_1}[x]]]]$

$\ldots$.

This implies that the three $RT_{F'}[x]$ is infinite, which is the desired contradiction.

Secondly, using Scott induction, we will show that F is partially correct with respect to its specification, namely:

$$(\forall x : I_F[x])\ (F[x] \downarrow \Longrightarrow O_F[x, F[x]]). \tag{3.100}$$

As it was broadly discussed in chapter (2), not every property is admissible and may be proven by Scott induction. However, as we already saw, properties which express partial correctness are known to be admissible.

Let us remind the definition of these properties: A property ϕ is said to be a partial correctness property if and only if there are predicates I and O, such that:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge I[a] \Longrightarrow O[a, f[a]])). \tag{3.101}$$

We now consider the following partial correctness property ϕ:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge I_F[a] \Longrightarrow O_F[a, f[a]])).$$

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω. By the definition of ϕ we obtain:

$$\phi[\Omega] \iff (\forall a)\ (\Omega[a] \downarrow\ \wedge I_F[a] \Longrightarrow O_F[a, \Omega[a]])),$$

and so, $\phi[\Omega]$ holds, since $\Omega[a] \downarrow$ never holds.

In the second step of Scott induction, we assume $\phi[f]$ holds for some f:

$$(\forall a)\ (f[a] \downarrow\ \wedge I_F[a] \Longrightarrow O_F[a, f[a]]), \tag{3.102}$$

and show $\phi[f_{new}]$, where f_{new} is obtained from f by the main program (3.86) as follows:

```
$f_{new}[x] =$ **If** $Q_0[x]$ **then** $S[x]$

              **elseif** $Q_1[x]$ **then** $C_1[x, f[R_{1,1}[x]], \ldots, f[R_{1,k_1}[x]]]$
              **elseif** $Q_2[x]$ **then** $C_2[x, f[R_{2,1}[x]], \ldots, f[R_{2,k_2}[x]]]$
              $\ldots$
              **elseif** $Q_n[x]$ **then** $C_n[x, f[R_{n,1}[x]], \ldots, f[R_{n,k_n}[x]]]$,
```

Now, we need to show now that for an arbitrary a,

$$f_{new}[a] \downarrow \ \wedge I_F[a] \implies O_F[a, f_{new}[a]].$$

Assume $f_{new}[a] \downarrow$ and $I_F[a]$. We have now the following two cases:

- Case 1: $Q_0[a]$.

  By the definition of f_{new} we obtain $f_{new}[a] = S[a]$ and since $f_{new}[a] \downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a] \downarrow$. Now using verification condition (3.94) we may conclude $O_F[a, S[a]]$ and hence $O_F[a, f_{new}[a]]$.

- Case 2: $Q_i[a]$ for some i, $1 \leq i \leq n$.

  By the definition of f_{new} we obtain:

  $$f_{new}[a] = C_i[a, f[R_{i,1}[a]],\ \ldots,\ f[R_{i,k_i}[a]]]$$

  and since $f_{new}[a] \downarrow$, we conclude that all the others involved in this computation must also terminate, that is:

  $$C_i[a, f[R_{i,1}[a]],\ \ldots,\ f[R_{i,k_i}[a]]] \downarrow,$$

  $$f[R_{i,1}[a]] \downarrow,\ \ldots, f[R_{i,k_i}[a]] \downarrow$$

  and

  $$R_{i,1}[a] \downarrow,\ \ldots, R_{i,k_i}[a] \downarrow .$$

  Since F is coherent, namely from $I_F[a]$, by (3.88)–(3.89), we obtain:

  $$I_F[R_{i,1}[x]] \ \wedge\ \ldots\ \wedge\ I_F[R_{i,k_1}[x]]$$

  and, knowing that for each j: $f[R_{i,j}[a]] \downarrow$, by the induction hypothesis (3.102) we obtain $O_F[R_{i,j}[a], f[R_{i,j}[a]]]$.

  Considering the appropriate i^{th} verification condition (3.95)–(3.96), note that all the assumptions from the left part of the implication are at hand and thus we can conclude:

  $$O_F[a, C_i[a, f[R_{i,1}[a]],\ \ldots,\ f[R_{i,k_i}[a]]]],$$

  which is

  $$O_F[a, f_{new}[a]].$$

Now we conclude that the property ϕ holds for the least fixpoint of (3.86) and hence, ϕ holds for the function computed by (3.86), which completes the proof of the soundness theorem (3.13).

3.6.3 Completeness of the Verification Conditions

Completing the notion of *Soundness*, we introduce its dual—*Completeness*.

As we already mentioned, after generating the verification conditions, one has to prove them as logical formulae. If all of them hold, then the program is correct with respect to its specification—*Soundness* theorem.

Now, we formulate the *Completeness* theorem for the class of coherent general multiple *else* recursive programs.

Theorem 3.14. *Let for any* i, j *the functions* S*,* C_i*, and* $R_{i,j}$ *satisfy their specifications* (I_S, O_S)*,* (I_C, O_C)*, and* $(I_{R_{i,j}}, O_{R_{i,j}})$*. Let also the general program with multiple* else F *as defined in* (3.86) *with its specification* (I_F, O_F) *be coherent with respect to* S*,* C_i*,* $R_{i,j}$*, and their specifications, and the output specification of* F*,* (O_F) *is functional one.*

Then if F is totaly correct with respect to (I_F, O_F) *then the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{3.103}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{k_1} : I_F[x])\ (Q_1[x] \wedge O_F[R_{1,1}[x], y_1] \ \wedge\ \ldots\ \wedge O_F[R_{1,k_1}[x], y_{k_1}] \\ \implies \\ O_F[x, C_1[x, y_1, \ldots, y_{k_1}]])\end{gathered} \tag{3.104}$$

$$\ldots$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{k_n} : I_F[x])\ (Q_n[x] \ \wedge\ O_F[R_{n,1}[x], y_1] \ \wedge\ \ldots\ \wedge O_F[R_{n,k_n}[x], y_{k_n}] \\ \implies \\ O_F[x, C_n[x, y_1, \ldots, y_{k_n}]])\end{gathered} \tag{3.105}$$

$$(\forall x : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.106}$$

where:

$$\begin{aligned} F'[x] = \ & \textbf{If } Q_0[x] \textbf{ then } \mathbb{T} \\ & \quad \textbf{elseif } Q_1[x] \textbf{ then } F'[R_{1,1}[x]] \ \wedge\ \ldots\ \wedge\ F'[R_{1,k_1}[x]] \\ & \quad \textbf{elseif } Q_2[x] \textbf{ then } F'[R_{2,1}[x]] \ \wedge\ \ldots\ \wedge\ F'[R_{2,k_2}[x]] \\ & \quad \ldots \\ & \quad \textbf{elseif } Q_n[x] \textbf{ then } F'[R_{n,1}[x]] \ \wedge\ \ldots\ \wedge\ F'[R_{n,k_n}[x]], \end{aligned} \tag{3.107}$$

which are the same as (3.94)*,* (3.95)–(3.96)*,* (3.97)*, and* (3.98) *from the Soundness theorem* (3.13).

Proof:
We assume now that:

- For all i, j the functions S, C_i, and $R_{i,j}$ are totaly correct with respect to their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and $(I_{R_{i,j}}, O_{R_{i,j}})$.
- The program F as defined in (3.86) with its specification (I_F, O_F) is coherent with respect to S, C_i, $R_{i,j}$, and their specifications.
- The output specification of F, (O_F) is functional one, that is:

$$(\forall x : I_F[x])\ (\exists! y)\ (O_F[x, y]).$$

- The program F as defined in (3.86) is correct with respect to its specification, that is, the total correctness formula holds:

$$(\forall x : I_F[x])\ (F[x] \downarrow \wedge O_F[x, F[x]]). \tag{3.108}$$

We show that (3.103), (3.104), …, (3.105), and (3.106) hold as logical formulae.

We start now with proving (3.103) and (3.104), …, (3.105) simultaneously.
Take arbitrary but fixed x and assume $I_F[x]$. We consider the following two cases:

- Case 1: $Q_0[x]$

  By the definition of F, we have $F[x] = S[x]$, and by using the correctness formula (3.108) of F, we conclude (3.103) holds. The formulae (3.104), …, (3.105) hold, because the predicates Q are consistent and noncontradictory, and hence $\neg Q_i[x]$ for all i, $1 \le i \le n$.

- Case 2: $Q_i[x]$ for some i, $1 \le i \le n$.

  Now, the formulae (3.103) and all except one of (3.104), …, (3.105) hold trivially, because at the left hand side of the implication we have $\neg Q_i[x]$.

  Assume $y_1, \ \ldots\ , y_{k_i}$ are such that:

$$O_F[R_{i,1}[x], y_1],\ \ldots\ ,\ O_F[R_{i,k_i}[x], y_{k_i}].$$

  Since F is correct, we obtain that:

$$y_1 = F[R_{i,1}[x]],\ \ldots\ ,\ y_{k_i} = F[R_{i,k_i}[x]]$$

  because O_F is a functional predicate.

  On the other hand, by the definition of F, we have:

  $F[x] = C_i[a, F[R_{i,1}[a]],\ \ldots,\ F[R_{i,k_i}[a]]]$

  and hence $F[x] = C_i[x, y_1,\ \ldots\ , y_{k_i}]$.

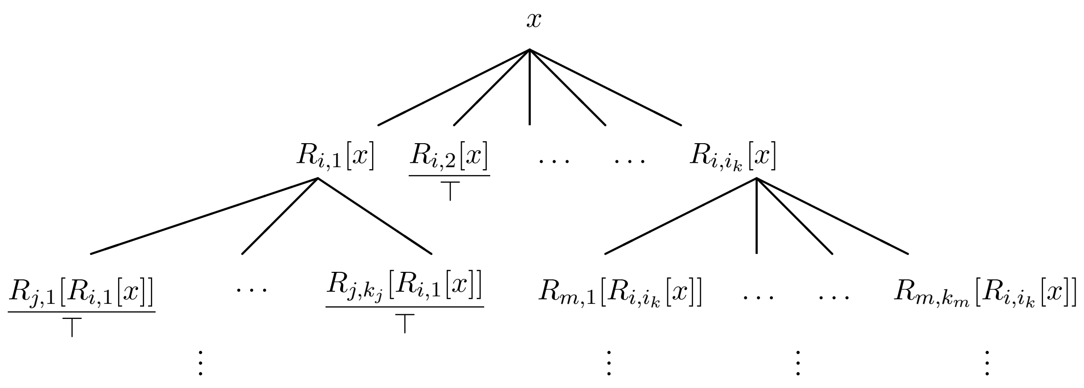
  Again, from the correctness of F, we obtain $O_F[x, C_i[x, y_1,\ \ldots\ , y_{k_i}]]$, which had to be proven.

Now, we show that the simplified version F' of the initial function F terminates. Moreover, F' terminates if F terminates. In the course of the proof, one may notice that proving $F'[x] = \mathbb{T}$ is the same as proving that F' terminates.

Take arbitrary but fixed x and assume $I_F[x]$.

Now we construct the recursive tree of F, starting form x, $RT_F[x]$ in the following way:

- x is the root of the tree, that is, the uppermost node;
- for any node u, if $Q_0[u]$ holds, then stop further construction on that branch, and put the symbol $\overline{\top}$;
- for any node u, if $Q_i[u]$ holds, for some $i \neq 0$, then construct all the k_i descendent nodes $R_{i,1}[u], \dots, R_{i,k_i}[u]$.


x
$R_{i,1}[x]$
$\frac{R_{i,2}[x]}{\top}$
$\dots \quad \dots$
$R_{i,i_k}[x]$
$\frac{R_{j,1}[R_{i,1}[x]]}{\top}$
$\dots$
$\frac{R_{j,k_j}[R_{i,1}[x]]}{\top}$
$R_{m,1}[R_{i,i_k}[x]]$
$\dots \quad \dots$
$R_{m,k_m}[R_{i,i_k}[x]]$

$Q_i[x]$

$Q_j[R_{i,1}[x]]$ $\qquad$ $Q_0[R_{i,2}[x]] \rightsquigarrow \overline{\top}$ $\qquad$ $Q_m[R_{i,i_k}[x]]$

$Q_0[R_{j,1}[R_{i,1}[x]]] \rightsquigarrow \overline{\top}$ $\qquad$ $Q_0[R_{j,k_j}[R_{i,1}[x]]] \rightsquigarrow \overline{\top}$

Note that the recursive tree of F, $RT_F[x]$ is the same as the recursive tree of F', $RT_{F'}[x]$. Thus $RT_F[x]$ is finite.

Now we need to show termination of F'. For our particular x (it was taken arbitrary but fixed, $I_F[x]$), we consider the following two cases:

- Case 1: $Q_0[x]$.

  Now by the definition of F', we have $F[x] = \mathbb{T}$ and hence $F'[x] \downarrow$.

- Case 2: $\neg Q_0[x]$, and say $Q_i[x]$. Now, by following the definition of F', we have,

  $$F'[x] = F'[R_{i,1}[x]] \wedge \dots \wedge F'[R_{i,k_i}[x]].$$

  We need to apply the same kind of reasoning to all the nodes of the recursive tree $RT_F[x]$, and since it is finite, after unfolding finitely many times we reach the leaves where for each leaf we arrive at the Case 1. Thus, $F'[x] \downarrow$.

By this we completed our proof of the *Completeness* theorem.

In order to illustrate the *Soundness* and the *Completeness* theorems, we may take any of the examples presented in this thesis—the only one which is not suitable for this schema is the "McCarthy 91", because it contains nested recursion.

3.7 Mutual Recursion

Mutual recursion is a special form of recursion where two programs are defined in terms of each other. Moreover, the two programs form a system, and its solution is the computable function which is defined by the programs.

We study the class of simple mutual recursive programs and we extract the purely logical conditions which are sufficient for the program correctness. As in the other sections, they are inferred using Scott induction and induction on natural numbers in the fixpoint theory of functions and constitute a meta-theorem.

Simple Mutual Recursive Programs are the simplest mutual recursive programs, however their study gives an impression how one can perform verification in more general setting.

We look at programs F, defined by the system F_1, F_2:

$$F[x] = F_1[x], \tag{3.109}$$

where:

$$F_1[x] = \textbf{ If } Q_1[x] \textbf{ then } S_1[x] \textbf{ else } C_1[x, F_2[R_1[x]]], \tag{3.110}$$

and

$$F_2[x] = \textbf{ If } Q_2[x] \textbf{ then } S_2[x] \textbf{ else } C_2[x, F_1[R_2[x]]], \tag{3.111}$$

with Q_1 and Q_2 are predicates and $S_1, S_2, C_1, C_2, R_1, R_2$ are auxiliary functions. Their names are chosen such that, $S_i[x]$ is a "simple" function, $C_i[x, y]$ is a "combinator" function, and $R_i[x]$ is a "reduction" function. We assume that the functions S_i, C_i, and R_i satisfy their specifications given by $I_{S_i}[x]$, $O_{S_i}[x, y]$, $I_{C_i}[x, y]$, $O_{C_i}[x, y, z]$, $I_{R_i}[x]$, $O_{R_i}[x, y]$.

The specifications of the two functions F_1 and F_2 is given, that is, $I_{F_1}[x]$, $O_{F_1}[x, y]$ and $I_{F_2}[x]$, $O_{F_2}[x, y]$. The specification of the main function F is (by definition) the same as the specification of F_1.

3.7.1 Coherent Simple Mutual Recursive Programs

In order to perform the coherence check, we define here the relevant verification conditions, which are derived from the definition of coherent programs (3.1) and (3.2), namely:

Definition 3.15. *Let S_i, C_i, and R_i (for $i = 1, 2$) be functions which satisfy their specifications (I_{S_i}, O_{S_i}), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Then the program F as defined in* (3.109) *with its specification (I_{F_1}, O_{F_1}) is coherent if F_1, F_2 are coherent with respect to S_i, C_i, R_i, and their specifications, if and only if the following conditions hold:*

$$(\forall x : I_{F_1}[x])\ (Q_1[x] \implies I_{S_1}[x]) \tag{3.112}$$

$$(\forall x : I_{F_1}[x])\ (\neg Q_1[x] \implies I_{F_2}[R_1[x]]) \tag{3.113}$$

$$(\forall x : I_{F_1}[x])\ (\neg Q_1[x] \implies I_{R_1}[x]) \tag{3.114}$$

$$(\forall x, y : I_{F_1}[x])\ (\neg Q_1[x] \wedge O_{F_2}[R_1[x], y] \implies I_{C_1}[x, y]) \tag{3.115}$$

$$(\forall x : I_{F_2}[x])\ (Q_2[x] \implies I_{S_2}[x]) \tag{3.116}$$

$$(\forall x : I_{F_2}[x])\ (\neg Q_2[x] \implies I_{F_1}[R_2[x]]) \tag{3.117}$$

$$(\forall x : I_{F_2}[x])\ (\neg Q_2[x] \implies I_{R_2}[x]) \tag{3.118}$$

$$(\forall x, y : I_{F_2}[x])\ (\neg Q_2[x] \wedge O_{F_1}[R_2[x], y] \implies I_{C_2}[x, y]) \tag{3.119}$$

As we can see, the above conditions correspond very much to our intuition about coherent programs, namely:

- (3.112) treats the special case in F_1, that is, $Q_1[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S_1.
- (3.113) treats the general case in F_1, that is, $\neg Q_1[x]$ holds and a call to F_2 is applied, thus the new input $R_1[x]$ must fulfill the precondition of F_2.
- (3.114) treats the general case in F_1, that is, $\neg Q_1[x]$ holds and a call to F_2 is applied, thus the input x must fulfill the precondition of the reduction function R_2.
- (3.115) treats the general case in F_1, that is, $\neg Q_1[x]$ holds and a call to F_2 is applied, thus the input x, together with any y (where y is a possible output $F_2[R_1[x]]$) must fulfill the precondition of the combinator function C_1.
- (3.116) treats the special case in F_2, that is, $Q_2[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S_2.
- (3.117) treats the general case in F_2, that is, $\neg Q_2[x]$ holds and a call to F_1 is applied, thus the new input $R_2[x]$ must fulfill the precondition of F_1.
- (3.118) treats the general case in F_2, that is, $\neg Q_2[x]$ holds and a call to F_1 is applied, thus the input x must fulfill the precondition of the reduction function R_1.
- (3.119) treats the general case in F_2, that is, $\neg Q_2[x]$ holds and a call to F_1 is applied, thus the input x, together with any y (where y is a possible output $F_1[R_2[x]]$) must fulfill the precondition of the combinator function C_2.

3.7.2 Verification Conditions and their Soundness

As we already discussed, reasoning about programs is translated into proving logical conditions. After generating these verification conditions, one has to prove them as logical formulae in the theory of the domain on which the program is defined. If all of them hold, then the program is correct with respect to its specification. The latter statement we call *Soundness* theorem, and we are now ready to define it for the class of coherent simple mutual recursive programs.

Theorem 3.16. *Let S_i, C_i, and R_i (for $i = 1, 2$) be functions which satisfy their specifications (I_{S_i}, O_{S_i}), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Let also the simple mutual recursive program F as defined in* (3.109) *with its specification (I_{F_1}, O_{F_1}) be coherent, that is, F_1, F_2 be coherent with respect to S_i, C_i, R_i, and their specifications. Then F is totaly correct with respect to (I_{F_1}, O_{F_1}) if the following verification conditions hold:*

$$(\forall x : I_{F_1}[x])\ (Q_1[x] \implies O_{F_1}[x, S_1[x]]) \tag{3.120}$$

$$(\forall x, y : I_{F_1}[x])\ (\neg Q_1[x]\ \wedge\ O_{F_2}[R_1[x], y] \implies O_{F_1}[x, C_1[x, y]]) \tag{3.121}$$

$$(\forall x : I_{F_2}[x])\ (Q_2[x] \implies O_{F_2}[x, S_2[x]]) \tag{3.122}$$

$$(\forall x, y : I_{F_2}[x])\ (\neg Q_2[x]\ \wedge\ O_{F_1}[R_2[x], y] \implies O_{F_2}[x, C_2[x, y]]) \tag{3.123}$$

$$(\forall x : I_{F_1}[x])\ (F_1'[x] = \mathbb{T}) \tag{3.124}$$

where:

$$F_1'[x] = \textbf{ If } Q_1[x] \textbf{ then } \mathbb{T} \textbf{ else } F_2'[R_1[x]] \tag{3.125}$$

$$F_2'[x] = \textbf{ If } Q_2[x] \textbf{ then } \mathbb{T} \textbf{ else } F_1'[R_2[x]]. \tag{3.126}$$

As we can see, the above conditions constitute the following principle:

- (3.120), (3.122) prove that the base cases for F_1 and F_2 are correct.
- (3.121), (3.123) prove that the recursive expressions for F_1 and F_2 are correct under the assumption that the reduced calls are correct.
- (3.124) prove that a simplified version F_1' of F_1, (whose definition also involves a simplified version F_2' of F_2) terminates for all possible inputs x: $I_{F_1}[x]$.

Proof:
The proof of the *Soundness* statement is split into two major parts:

- prove partial correctness using Scott induction;
- prove termination.

First we will see that F (3.109) terminates.

Indeed, from the assumption that: S_i, C_i, and R_i are totally correct (with respect to I_{S_i}, I_{C_i}, and I_{R_i}) by the coherence of F_1 and F_2, namely, formulae (3.112), (3.113), (3.114), (3.115), and, (3.116), (3.117), (3.118), (3.119) we ensure the *termination* of the calls to the auxiliary functions S_i, C_i, and R_i.

Take arbitrary but fixed x and assume $I_{F_1}[x]$. From (3.124), we obtain that $F'[x] = \mathbb{T}$.

Now we construct the recursive tree of the system F_1', F_2' starting form x, $RT_{F_1',F_2'}[x]$ in the following way:

- x is the root of the tree, that is, the uppermost node;
- x is an even node;
- for any even node u, if $Q_1[u]$ holds, then stop further construction, and put the symbol $\overline{\top}$;
- for any even node u, if $\neg Q_1[u]$ holds, construct an odd node $R_1[u]$;
- for any odd node u, if $Q_2[u]$ holds, then stop further construction, and put the symbol $\overline{\top}$;
- for any odd node u, if $\neg Q_2[u]$ holds, construct an even node $R_2[u]$.

$$\begin{tikzcd}
\textit{even} & x \arrow[d, no head, "\neg Q_1[x]"] \\
\textit{odd} & R_1[x] \arrow[d, no head, "\neg Q_2[R_1[x]]"] \\
\textit{even} & R_2[R_1[x]] \arrow[d, no head, "\neg Q_1[R_2[R_1[x]]]"] \\
\textit{odd} & R_1[R_2[R_1[x]]] \\
 & \vdots \\
 & \vdots \\
\textit{even} & \underline{R_2[\ldots\ R_2[R_1[x]]]} \\
 & \overline{\overline{\top}} & Q_1[R_2[\ldots\ R_2[R_1[x]]]]
\end{tikzcd}$$

$Q_1[R_2[\ldots\ R_2[R_1[x]]]] \rightsquigarrow \overline{\top}$

We first show that $RT_{F'_1,F'_2}[x]$ is finite.

We prove this statement by contradiction, i.e. assume $RT_{F'_1,F'_2}[x]$ is infinite. Hence, for any even node u there will be $\neg Q_1[u]$, and for any odd node u — $\neg Q_2[u]$, respectively.

Now, we look at the construction of the system F'_1, F'_2 as being the least fixpoint of (3.125), (3.126).

Let $f_0, f_1, \dots f_m, \dots$ be the finite approximations of F'_1, and $g_0, g_1, \dots g_m, \dots$ be the finite approximations of F'_2, obtained from the nowhere defined function Ω, in the following way:

$$f_0[x] = \Omega[x]$$

$$f_{m+1}[x] = \textbf{ If } Q_1[x] \textbf{ then } \mathbb{T} \textbf{ else } g_m[R_1[x]]$$

and

$$g_0[x] = \Omega[x]$$

$$g_{m+1}[x] = \textbf{ If } Q_2[x] \textbf{ then } \mathbb{T} \textbf{ else } f_m[R_2[x]].$$

The computable function F'_1, corresponding to (3.125) is defined as:

$$F'_1 = \bigcup_m f_m, \quad \text{and} \quad F'_2 = \bigcup_m g_m.$$

Since for our particular x (it was taken arbitrary but fixed) we have $F'_1(x) = \mathbb{T}$, there must exist a finite approximation f_m, such that:

$$f_m[x] = \mathbb{T}.$$

If $m = 0$, then $f_0[x] = \mathbb{T}$, but on the other hand, by its definition, f_0 is the nowhere defined function $f_0 = \Omega$, thus this is not a case. Hence, we conclude that $m > 0$.

From the assumption, and in particular $\neg Q_1[x]$, by the definition of f_m we obtain:

$$f_m[x] = g_{m-1}[R_1[x]].$$

From here, and $f_m[x] = \mathbb{T}$ we obtain that:

$$g_{m-1}[R_1[x]] = \mathbb{T}.$$

Now, from the assumption $\neg Q_2[R_1[x]]$, by the definition of g_{m-1} we obtain:

$$g_{m-1}[R_1[x]] = f_{m-2}[R_2[R_1[x]]] = \mathbb{T}.$$

By repeating the same kind of reasoning m times (in fact, formally it is done by induction), we obtain that:

$$f_0[R_2[\dots[R_1[x]]]] = \mathbb{T}$$

and by its definition ($f_0 = \Omega$) we obtain:

$$f_0[R_2[\dots[R_1[x]]]] = \bot.$$

Remark: If m is an odd number, we obtain:

$$g_0[R_1[\dots[R_1[x]]]] = \mathbb{T},$$

however, the observation is the same.
This is the desired contradiction, and hence, we have proven that the recursive tree $RT_{F_1',F_2'}[x]$ is finite.

Now we continue the proof of the termination of F_1. We prove this statement by contradiction, i.e. assume $RT_{F_1',F_2'}[x]$ is finite and $F_1[x] = \bot$.
For our particular x (it was taken arbitrary but fixed, $I_{F_1}[x]$), we consider the following two cases:

- Case 1: $Q_1[x]$.

  Now by the definition of F_1, we have $F_1[x] = S_1[x]$. We chose x such that $I_{F_1}[x]$, and by (3.112) we obtain that $S_1[x] \downarrow$ and hence $F_1[x] \downarrow$ and thus we obtain a contradiction.

- Case 2: $\neg Q_1[x]$. Now, by following the definition of F_1, we have,

  $$F_1[x] = C_1[x, F_2[R_1[x]]],$$

  and since F_1 is coherent, we have $I_{R_1}[x]$, and $I_{C_1}[x, y]$, and thus, in order to have $F_1[x] = \bot$, we have $F_2[R_1[x]] = \bot$.

  By following the definition of F_2, we see that if $Q_2[R_1[x]]$ holds, $F_2[R_1[x]]$ terminates (which contradicts $F_1[x] = \bot$) and thus we conclude $\neg Q_2[R_1[x]]$.

  Applying the same kind of reasoning we obtain an infinite sequence:

  $$\neg Q_1[x],\ \neg Q_2[R_1[x]],\ \dots\ ,\ \neg Q_2[R_1\ \dots\ [R_1[x]]]\ \dots\ ,$$

  which implies that the three $RT_{F_1',F_2'}[x]$ is infinite, and this is the desired contradiction.

Secondly, using Scott induction, we will show that F_1 is partially correct:

$$(\forall x : I_{F_1}[x])\ (F_1[x] \downarrow \Longrightarrow O_{F_1}[x, F_1[x]]). \tag{3.127}$$

In fact, we will show that the minimal fixpoint f, g of the system (3.110), (3.111), satisfies:

$$(\forall x : I_{F_1}[x])\ (f[x] \downarrow \Longrightarrow O_{F_1}[x, f[x]])$$

$$(\forall x : I_{F_2}[x])\ (g[x] \downarrow \Longrightarrow O_{F_2}[x, g[x]]).$$

As it was broadly discussed in chapter (2), not every property is admissible and may be proven by Scott induction. However, as we already saw, properties which express partial correctness are known to be admissible.

Let us remind the definition of these properties: A property ϕ is said to be a partial correctness property if and only if there are predicates I and O, such that:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a] \downarrow\ \wedge I[a] \Longrightarrow\ O[a, f[a]])). \tag{3.128}$$

We now consider the following partial correctness properties ϕ_1 and ϕ_2:

$$(\forall f,g)\ (\phi_1[f,g] \iff (\forall a)\ (f[a]\downarrow\ \wedge I_{F_1}[a] \implies O_{F_1}[a,f[a]]))$$

$$(\forall f,g)\ (\phi_2[f,g] \iff (\forall a)\ (g[a]\downarrow\ \wedge I_{F_2}[a] \implies O_{F_2}[a,g[a]])).$$

Since the properties ϕ_1 and ϕ_2 are continuous, by Lemma (2.28), we obtain that their conjunction is continuous property as well. Namely, we construct the property ϕ as:

$$(\forall f,g)\ (\phi[f,g] \iff$$
$$(\forall a)\ (f[a]\downarrow\ \wedge I_{F_1}[a] \implies O_{F_1}[a,f[a]]) \wedge$$
$$\wedge\ (\forall a)\ (g[a]\downarrow\ \wedge I_{F_2}[a] \implies O_{F_2}[a,g[a]])).$$

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω. By the definition of ϕ we obtain:

$$\phi[\Omega,\Omega] \iff$$
$$(\forall a)\ (\Omega[a]\downarrow\ \wedge I_{F_1}[a] \implies O_{F_1}[a,\Omega[a]]) \wedge$$
$$\wedge\ (\forall a)\ (\Omega[a]\downarrow\ \wedge I_{F_2}[a] \implies O_{F_2}[a,\Omega[a]]),$$

and so, $\phi[\Omega,\Omega]$ holds, since $\Omega[a]\downarrow$ never holds.

In the second step of Scott induction, we assume $\phi[f,g]$ holds for some f,g:

$$(\forall a)\ (f[a]\downarrow\ \wedge I_{F_1}[a] \implies O_{F_1}[a,f[a]])$$
$$(\forall a)\ (g[a]\downarrow\ \wedge I_{F_2}[a] \implies O_{F_2}[a,g[a]])$$

and show $\phi[f_{new},g_{new}]$, where f_{new},g_{new} are obtained from f,g by the system (3.110), (3.111) as follows:

$$f_{new} = \textbf{If } Q_1[x] \textbf{ then } S_1[x] \textbf{ else } C_1[x,g[R_1[x]]]$$

$$g_{new} = \textbf{If } Q_2[x] \textbf{ then } S_2[x] \textbf{ else } C_2[x,f[R_2[x]]].$$

First we show $\phi_1[f_{new},g_{new}]$, namely we need to show now that for an arbitrary a:

$$f_{new}[a]\downarrow\ \wedge I_{F_1}[a] \implies O_{F_1}[a,f_{new}[a]].$$

Assume $f_{new}[a]\downarrow$ and $I_{F_1}[a]$. We have now the following two cases:

- Case 1: $Q_1[a]$.

  By the definition of f_{new} we obtain $f_{new}[a] = S_1[a]$ and since $f_{new}[a]\downarrow$, we obtain that $S_1[a]$ must terminate as well, that is $S_1[a]\downarrow$. Now using verification condition (3.120) we may conclude $O_{F_1}[a,S_1[a]]$ and hence $O_{F_1}[a,f_{new}[a]]$.

- Case 2: $\neg Q_1[a]$.

  By the definition of f_{new} we obtain $f_{new}[a] = C_1[a, g[R_1[a]]]$ and since $f_{new}[a] \downarrow$, we conclude that all the others involved in this computation must also terminate, that is: $C_1[a, g[R_1[a]]] \downarrow$, $g[R_1[a]] \downarrow$, and $R_1[a] \downarrow$.

  From $I_{F_1}[a]$, by (3.113), we obtain $I_{F_2}[R_1[a]]$ and, knowing that: $g[R_1[a]] \downarrow$ by the induction hypothesis we obtain $O_{F_2}[R_1[a], g[R_1[a]]]$.

  Concerning the verification condition (3.121), note that all the assumptions from the left part of the implication are at hand and thus we can conclude:

  $$O_{F_1}[a, C_1[a, g[R_1[a]]]].$$

  Since we have:

  $$f_{new}[a] = C_1[a, g[R_1[a]]],$$

  we finally obtain that $O_{F_1}[a, f_{new}[a]]$.

It remains to show $\phi_2[f_{new}, g_{new}]$, namely we need to show now that for an arbitrary a:

$$g_{new}[a] \downarrow \ \wedge I_{F_2}[a] \implies O_{F_2}[a, g_{new}[a]].$$

Assume $g_{new}[a] \downarrow$ and $I_{F_2}[a]$. We have now the following two cases:

- Case 1: $Q_2[a]$.

  By the definition of g_{new} we obtain $g_{new}[a] = S_2[a]$ and since $g_{new}[a] \downarrow$, we obtain that $S_2[a]$ must terminate as well, that is $S_2[a] \downarrow$. Now using verification condition (3.122) we may conclude $O_{F_2}[a, S_2[a]]$ and hence $O_{F_2}[a, g_{new}[a]]$.

- Case 2: $\neg Q_2[a]$.

  By the definition of g_{new} we obtain $g_{new}[a] = C_2[a, f[R_2[a]]]$ and since $g_{new}[a] \downarrow$, we conclude that all the others involved in this computation must also terminate, that is: $C_2[a, f[R_2[a]]] \downarrow$, $f[R_2[a]] \downarrow$, and $R_2[a] \downarrow$.

  From $I_{F_2}[a]$, by (3.117), we obtain $I_{F_1}[R_2[a]]$ and, knowing that: $f[R_2[a]] \downarrow$ by the induction hypothesis we obtain $O_{F_1}[R_2[a], f[R_2[a]]]$.

  Concerning the verification condition (3.123), note that all the assumptions from the left part of the implication are at hand and thus we can conclude:

  $$O_{F_2}[a, C_2[a, f[R_2[a]]]].$$

  Since we have:

  $$g_{new}[a] = C_2[a, f[R_2[a]]],$$

  we finally obtain that $O_{F_2}[a, g_{new}[a]]$.

Now we conclude that the property ϕ holds for the least fixpoint of the system (3.110), (3.111) and hence, ϕ holds for the function computed by this system, which completes the proof of the soundness theorem (3.16).

3.7.3 Completeness of the Verification Conditions

As we already mentioned in the introduction, the notion of *Completeness* of a verification condition generator is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on "what is wrong".

Now, we formulate the *Completeness* theorem for the class of coherent simple mutual recursive programs.

Theorem 3.17. *Let S_i, C_i, and R_i (for $i = 1, 2$) be functions which satisfy their specifications (I_{S_i}, O_{S_i}), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Let also the simple mutual recursive program F as defined in* (3.109) *with its specification (I_{F_1}, O_{F_1}) be coherent, that is, F_1, F_2 be coherent with respect to S_i, C_i, R_i, and their specifications, and the output specifications of F_i, (O_{F_i}) are functional ones.*

Then if F_1 and F_2 are totaly correct with respect to (I_{F_i}, O_{F_i}) then the following verification conditions hold:

$$(\forall x : I_{F_1}[x])\ (Q_1[x] \implies O_{F_1}[x, S_1[x]]) \tag{3.129}$$

$$(\forall x, y : I_{F_1}[x])\ (\neg Q_1[x]\ \wedge\ O_{F_2}[R_1[x], y] \implies O_{F_1}[x, C_1[x, y]]) \tag{3.130}$$

$$(\forall x : I_{F_2}[x])\ (Q_2[x] \implies O_{F_2}[x, S_2[x]]) \tag{3.131}$$

$$(\forall x, y : I_{F_2}[x])\ (\neg Q_2[x]\ \wedge\ O_{F_1}[R_2[x], y] \implies O_{F_2}[x, C_2[x, y]]) \tag{3.132}$$

$$(\forall x : I_{F_1}[x])\ ({F_1}'[x] = \mathbb{T}) \tag{3.133}$$

where:

$$F_1'[x] = \textbf{ If } Q_1[x] \textbf{ then } \mathbb{T} \textbf{ else } F_2'[R_1[x]] \tag{3.134}$$

$$F_2'[x] = \textbf{ If } Q_2[x] \textbf{ then } \mathbb{T} \textbf{ else } F_1'[R_2[x]]. \tag{3.135}$$

which are the same as (3.120), (3.121), (3.122), (3.123), (3.124), *and* (3.125), (3.126) *from the Soundness theorem* (3.16).

Proof:
We assume now that:

- The functions S_i, C_i, and R_i (for $i = 1, 2$) are totaly correct with respect to their specifications (I_{S_i}, O_{S_i}), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}).
- The simple mutual recursive program F as defined in (3.109) with its specification (I_{F_1}, O_{F_1}) is coherent, that is, F_1, F_2 are coherent with respect to S_i, C_i, R_i, and their specifications.

- The output specifications of F_i, (O_{F_i}) are functional ones, that is:

$$(\forall x : I_{F_i}[x])\ (\exists! y)\ (O_{F_i}[x, y]).$$

- Each of the components of the system F_1 and F_2 as defined in (3.110), and (3.111) are correct with respect to their specifications, that is, the total correctness formulae hold:

$$(\forall x : I_{F_i}[x])\ (F_i[x] \downarrow \wedge O_{F_i}[x, F_i[x]]). \tag{3.136}$$

We show that (3.129),(3.130), (3.131), (3.132), and (3.133) hold as logical formulae.

We start now with proving (3.129) and (3.130) simultaneously.
Take arbitrary but fixed x and assume $I_{F_1}[x]$. We consider the following two cases:

- Case 1: $Q_1[x]$

  By the definition of F_1, we have $F_1[x] = S_1[x]$, and by using the correctness formula (3.136) of F_1, we conclude (3.129) holds. The formula (3.130) holds, because we have $Q_1[x]$.

- Case 2: $\neg Q_1[x]$

  Now, (3.129) holds. Assume y is such that $O_{F_2}[R_1[x], y]$. Since F_2 is correct, we obtain that $y = F_2[R_1[x]]$, because O_{F_2} is a functional predicate.

  On the other hand, by the definition of F_1, we have $F_1[x] = C_1[x, F_2[R_1[x]]]$ and hence $F_1[x] = C_1[x, y]$. From the correctness of F_1, we obtain $O_{F_1}[x, C_1[x, y]]$, which completes the proof of (3.130).

We continue with proving (3.131) and (3.132) simultaneously, which is very similar to the previous proof. Take arbitrary but fixed x and assume $I_{F_2}[x]$. We consider the following two cases:

- Case 1: $Q_2[x]$

  By the definition of F_2, we have $F_2[x] = S_2[x]$, and by using the correctness formula (3.136) of F_2, we conclude (3.131) holds. The formula (3.132) holds, because we have $Q_2[x]$.

- Case 2: $\neg Q_2[x]$

  Now, (3.131) holds. Assume y is such that $O_{F_1}[R_2[x], y]$. Since F_1 is correct, we obtain that $y = F_1[R_2[x]]$, because O_{F_1} is a functional predicate.

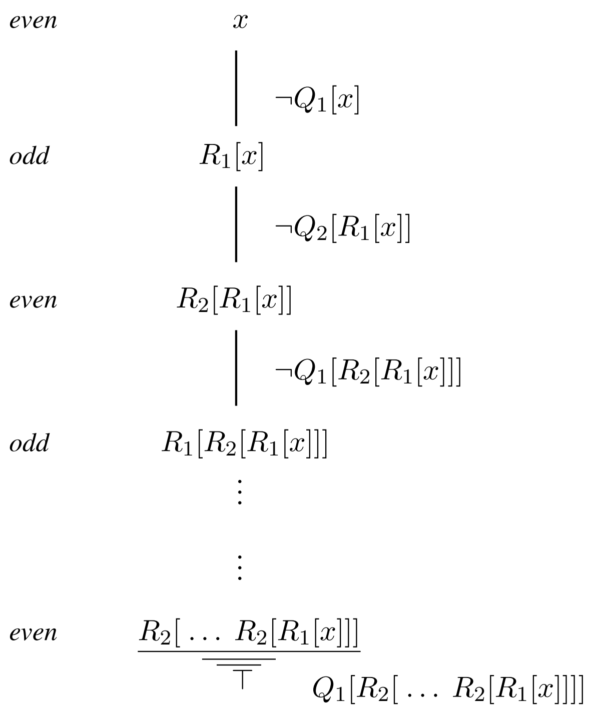
  On the other hand, by the definition of F_2, we have $F_2[x] = C_2[x, F_1[R_2[x]]]$ and hence $F_2[x] = C_2[x, y]$. From the correctness of F_2, we obtain $O_{F_2}[x, C_2[x, y]]$, which completes the proof of (3.132).

Now, we show that the simplified versions F_1' and F_2' of the components of the system F_1, F_2 terminate. Take arbitrary but fixed x and assume $I_{F_1}[x]$.

Now we construct the recursive tree of the system F_1', F_2' starting form x, $RT_{F_1', F_2'}[x]$ in the following way:

- x is the root of the tree, that is, the uppermost node;

- x is an even node;
- for any even node u, if $Q_1[u]$ holds, then stop further construction, and put the symbol $\overline{\top}$;
- for any even node u, if $\neg Q_1[u]$ holds, construct an odd node $R_1[u]$;
- for any odd node u, if $Q_2[u]$ holds, then stop further construction, and put the symbol $\overline{\top}$;
- for any odd node u, if $\neg Q_2[u]$ holds, construct an even node $R_2[u]$.



$Q_1[R_2[\ldots\ R_2[R_1[x]]]] \rightsquigarrow \overline{\top}$

Note that the recursive tree of F_1, F_2, $RT_{F_1,F_2}[x]$ is the same as the recursive tree of F_1', F_2', $RT_{F_1',F_2'}[x]$. Thus $RT_{F_1,F_2}[x]$ is finite.

Now we need to show termination of F_1'. In fact, we proof termination of F_1' and F_2' simultaneously. For our particular x (it was taken arbitrary but fixed, $I_{F_1}[x]$), we consider the following two cases:

- Case 1: $Q_1[x]$.

  Now by the definition of F_1', we have $F_1'[x] = \mathbb{T}$ and hence $F_1'[x] \downarrow$. In this case, F_2' terminates trivially, because no execution is made at all.

- Case 2: $\neg Q_1[x]$. Now, by following the definition of F_1', we have, $F_1'[x] = F_2'[R_1[x]]$.

  By unfolding the definitions of F_1' and F_2', we see that we actually cover all the nodes of the recursive tree $RT_{F_1,F_2}[x]$, and since it is finite, after unfolding finitely many times we reach the bottom. Thus, $F_1'[x] \downarrow$ and $F_2' \downarrow$.

By this we completed our proof of the *Completeness* theorem.

In order to illustrate the class of simple mutual recursive programs, and, actually what are the necessary and sufficient conditions for the program to be correct, we point to an example (3.10.6) for checking whether a given natural number is even or not.

3.8 Nested Recursion

In this section we study a class of recursive functions may contain nested recursive definitions. The programs here are like simple recursive programs, but on the *else* branch several nested recursive calls may appear, that is, an argument to a recursive call of the program may contain another invocation of the program itself. For example:

$$f[x] = \textbf{ If } x = 0 \textbf{ then } 0 \textbf{ else } f[f[x-1]].$$

Now, for this kind of programs we will not attempt proving termination but only partial correctness. The proofs of the *Soundness* and the *Completeness* theorems (in their parts concerning termination)are very complicated and presenting them would not be worth reading.

The reason of doing so we summarize as follows: Throughout the previous sections we conveyed the idea that, in order to prove termination, we construct a simplified version F' of the initial program F and we state that proving termination of F is equivalent to proving termination of F'. This idea worked well for programs fitting to "popular" schemata, e.g. $factorial$, and proving termination of many examples would be reduced to matching against simplified versions. However, programs containing nested recursion are not popular at all, and finding examples having the same simplified version is now hopeless.

The reader would loose any motivation, if finally in the examples, the initial program is the same as its simplified version as it would be with the McCarthy 91 function [48], [47].

In order to prove partial correctness, we again extract the purely logical conditions which are sufficient and also necessary for the program to be partially correct. The proofs of the *Soundness* and the *Completeness* theorems are similar to the ones presented in (3.3), however, they treat partial correctness.

We concentrate here on the essence of nested recursion, thus we consider programs of the form:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C_1[x, F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]]]. \tag{3.137}$$

We assume that the functions S, C_1, $\ldots$, C_k and R satisfy their specifications given by $I_S[x]$, $O_S[x,y]$, $I_{C_i}[x,y]$, $O_{C_i}[x,y,z]$, $I_R[x]$, $O_R[x,y]$.

It is well agreed that functions having nested recursion are difficult to reason about [59], especially in an automatic manner. The purpose of our study is to generate adequate verification conditions, which describe the program definition and the specification as first order predicate logic formulae.

We demonstrate our method on the McCarthy 91 function, which is viewed as a "challenge problem" for automated program verification.

3.8.1 Coherent Nested Recursive Programs

We start up with instantiating the definitions for coherent programs (3.1) and (3.2), namely:

Definition 3.18. *Let for all i, the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Then the program F as defined in* (3.137) *with its specification (I_F, O_F) is coherent with respect to S, C_i, R, and their specifications, if and only if the following conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \implies I_S[x]) \tag{3.138}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_F[R[x]]) \tag{3.139}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_R[x]) \tag{3.140}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{2k} : I_F[x]) \\ (\neg Q[x]\ \wedge\ O_F[R[x], y_1]\ \wedge\ O_F[y_2, y_3]\ \wedge\ \ldots\ \wedge\ O_F[y_{2k-2}, y_{2k-1}] \wedge \\ \wedge\, O_{C_k}[x, y_1, y_2]\ \wedge\ O_{C_{k-1}}[x, y_3, y_4]\ \wedge\ \ldots\ \wedge\ O_{C_1}[x, y_{2k-1}, y_{2k}] \\ \implies \\ I_{C_1}[x, y_1]\ \wedge\ I_{C_2}[x, y_3]\ \wedge\ \ldots\ \wedge\ I_{C_k}[x, y_{2k-1}] \wedge \\ \wedge\, I_F[y_2]\ \wedge\ I_F[y_4]\ \wedge\ \ldots\ \wedge I_F[y_{2k-2}])\end{gathered} \tag{3.141}$$

Now the conditions for coherence look a bit more complicated, however, in the example we will see that this is not a case. We see again that our intuition about coherent programs is met, namely:

- (3.138) treats the special case, that is, $Q[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S.
- (3.139) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the first new input $R[x]$ must fulfill the precondition of the main function F.
- (3.140) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the input x must fulfill the precondition of the reduction function R.
- (3.141) treats the general case, and expresses in a cascade manner, that all the inputs to the combinator functions $C_1, \ldots, C_k$ must be appropriate and also all the intermediate inputs to the main function F must be appropriate as well.

After having defined the coherence verification conditions, we go towards defining the verification conditions for ensuring total correctness.

3.8.2 Verification Conditions and their Soundness

We introduce the verification conditions for the class of programs with nested recursion, by providing the relevant *Soundness* theorem. The statement itself and the proof are generalization of the similar theorem (3.4) in that part where partial correctness is concerned.

Theorem 3.19. *Let for all i, the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Let also the program F as defined in* (3.137) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R, and their specifications. Then F is partially correct with respect to (I_F, O_F) if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]]) \tag{3.142}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{2k} : I_F[x]) \\ (\neg Q[x] \ \wedge\ O_F[R[x], y_1] \ \wedge\ O_F[y_2, y_3] \ \wedge\ \ldots\ \wedge\ O_F[y_{2k-2}, y_{2k-1}] \wedge \\ \wedge\, O_{C_k}[x, y_1, y_2] \ \wedge\ O_{C_{k-1}}[x, y_3, y_4] \ \wedge\ \ldots\ \wedge\ O_{C_1}[x, y_{2k-1}, y_{2k}] \\ \Longrightarrow \\ O_F[x, y_{2k}])\end{gathered} \tag{3.143}$$

The above conditions constitute the following principle:

- (3.142) prove that the base case is correct.
- (3.143) prove that the recursive expression is correct under the assumption that all the reduced calls are correct.

Proof:

Using Scott induction, we will show that F is partially correct with respect to its specification, namely:

$$(\forall x : I_F[x])\ (F[x]\downarrow \Longrightarrow O_F[x, F[x]]). \tag{3.144}$$

We now consider the following partial correctness property ϕ:

$$(\forall f)\ (\phi[f] \iff (\forall a)\ (f[a]\downarrow\ \wedge I_F[a] \implies O_F[a, f[a]])).$$

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω. By the definition of ϕ we obtain:

$$\phi[\Omega] \iff (\forall a)\ (\Omega[a]\downarrow\ \wedge I_F[a] \implies O_F[a, \Omega[a]])),$$

and so, $\phi[\Omega]$ holds, since $\Omega[a]\downarrow$ never holds.

In the second step of Scott induction, we assume $\phi[f]$ holds for some f:

$$(\forall a)\ (f[a]\downarrow\ \wedge I_F[a] \implies O_F[a, f[a]]), \tag{3.145}$$

and show $\phi[f_{new}]$, where f_{new} is obtained from f by the main program (3.137) as follows:

$$f_{new} = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]].$$

Now, we need to show that for an arbitrary a,

$$f_{new}[a]\downarrow\ \wedge I_F[a] \implies O_F[a, f_{new}[a]].$$

Assume $f_{new}[a]\downarrow$ and $I_F[a]$. We have now the following two cases:

- Case 1: $Q[a]$.

  By the definition of f_{new} we obtain $f_{new}[a] = S[a]$ and since $f_{new}[a]\downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a]\downarrow$. Now using verification condition (3.142) we may conclude $O_F[a, S[a]]$ and hence $O_F[a, f_{new}[a]]$.

- Case 2: $\neg Q[a]$.

  By the definition of f_{new} we obtain:

$$f_{new}[a] = C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]]$$

and since $f_{new}[a] \downarrow$, we obtain that all the programs involved in this computation also terminate, that is:

$$C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]] \downarrow$$

and say: $C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]] = y_{2k}$,

$$f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]] \downarrow$$

and say: $f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]] = y_{2k-1}$,

$$C_2[a, f[\ldots C_k[a, f[R[a]]]]] \downarrow$$

and say: $C_2[a, f[\ldots C_k[a, f[R[a]]]]] = y_{2k-2}$,

$$f[C_3[a, f[\ldots C_k[a, f[R[a]]]]]] \downarrow$$

and say: $f[C_3[a, f[\ldots C_k[a, f[R[a]]]]]] = y_{2k-3}$,

$$\ldots$$

$$f[C_k[a, f[R[a]]]] \downarrow$$

and say: $f[C_k[a, f[R[a]]]] = y_3$,

$$C_k[a, f[R[a]]] \downarrow$$

and say: $C_k[a, f[R[a]]] = y_2$,

$$f[R[a]] \downarrow$$

and say: $f[R[a]] = y_1$, and

$$R[a] \downarrow .$$

From here, by the induction hypothesis, we obtain that:

$$O_F[R[a], y_1] \ \wedge \ O_F[y_2, y_3] \ \wedge \ \ldots \ \wedge \ O_F[y_{2k-2}, y_{2k-1}].$$

On the other hand, by knowing that all the programs C_1, C_2, ..., C_k are partially correct with respect to their specifications, we obtain that:

$$O_{C_1}[a, y_{2k-1}, y_{2k}] \ \wedge \ O_{C_2}[a, y_{2k-3}, y_{2k-2}] \ \wedge \ \ldots \ \wedge \ O_{C_{k-1}}[a, y_3, y_4] \ \wedge \ O_{C_k}[a, y_1, y_2].$$

Concerning the verification condition (3.143), note that all the assumptions from the left part of the implication are at hand and thus we can conclude:

$$O_F[a, y_{2k}],$$

and thus $O_F[a, f_{new}[a]]$.

Now we conclude that the property ϕ holds for the least fixpoint of (3.137) and hence, ϕ holds for the function computed by (3.137), which completes the proof of the soundness theorem (3.19).

This completes the proof of the *Soundness* theorem. Now we proceed towards its complement, namely, the *Completeness* theorem.

3.8.3 Completeness of the Verification Conditions

Now, we formulate the *Completeness* theorem for the class of programs with nested recursion.

Theorem 3.20. *Let for all i the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Let also the program F as defined in* (3.137) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R, and their specifications, and the output specifications of F, C_i: (O_F), $(O_{C_I}$ are functional ones.*

Then if F is partially correct with respect to (I_F, O_F) then the following verification conditions hold:

$$(\forall x : I_F[x]) \ (Q[x] \implies O_F[x, S[x]]) \tag{3.146}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{2k} : I_F[x]) \\ (\neg Q[x] \wedge \ O_F[R[x], y_1] \ \wedge \ O_F[y_2, y_3] \ \wedge \ \ldots \ \wedge \ O_F[y_{2k-2}, y_{2k-1}] \ \wedge \\ \wedge \ O_{C_k}[x, y_1, y_2] \ \wedge \ O_{C_{k-1}}[x, y_3, y_4] \ \wedge \ \ldots \ \wedge \ O_{C_1}[x, y_{2k-1}, y_{2k}] \\ \implies \\ O_F[x, y_{2k}])\end{gathered} \tag{3.147}$$

which are the same as (3.142) *and* (3.143) *from the Soundness theorem* (3.19).

Proof:
We assume now that:

- The functions S, C_i, and R are partially correct with respect to their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R).

- The program F as defined in (3.137) with its specification (I_F, O_F) is coherent with respect to S, C_i, R, and their specifications.
- The output specifications of F, C_i: O_F, O_{C_i} are functional ones, that is:

$$(\forall x : I_F[x])\ (\exists! y)\ (O_F[x, y]),$$

$$(\forall x, y : I_{C_i}[x, y])\ (\exists! z)\ (O_{C_i}[x, y, z]).$$

- The program F as defined in (3.137) is partially correct with respect to its specification, that is, the partial correctness formula holds:

$$(\forall x : I_F[x])\ (F[x] \downarrow \Longrightarrow O_F[x, F[x]]). \tag{3.148}$$

We show that (3.146) and (3.147) hold as logical formulae by proving them simultaneously. Take arbitrary but fixed x and assume $I_F[x]$ and $F[x] \downarrow$. We consider the following two cases:

- Case 1: $Q[x]$

  By the definition of F, we have $F[x] = S[x]$, and by using the partial correctness formula (3.148) of F, we conclude (3.146) holds. The formula (3.147) holds, because we have $Q[x]$.

- Case 2: $\neg Q[x]$

  Now, (3.146) holds. Assume $y_1, \ldots, y_{2k}$ are such that:

$$O_F[R[x], y_1]\ \wedge\ O_F[y_2, y_3]\ \wedge\ \ldots\ \wedge\ O_F[y_{2k-2}, y_{2k-1}] \wedge$$
$$\wedge\ O_{C_k}[x, y_1, y_2]\ \wedge\ O_{C_{k-1}}[x, y_3, y_4]\ \wedge\ \ldots\ \wedge\ O_{C_1}[x, y_{2k-1}, y_{2k}].$$

  Since F is partially correct and $F[x] \downarrow$, we obtain that:

$$C_1[x, F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]]] \downarrow$$

$$F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]] \downarrow$$

$$C_2[x, F[\ldots C_k[x, F[R[x]]]]] \downarrow$$

$$F[C_3[x, F[\ldots C_k[x, F[R[x]]]]]] \downarrow$$

$$\ldots$$

$$F[C_k[x, F[R[x]]]] \downarrow$$

$$C_k[x, F[R[x]]] \downarrow$$

$$F[R[x]] \downarrow$$

$$R[x] \downarrow .$$

Since the output specifications of F, C_i: O_F, O_{C_i} are functional predicates, we obtain that:

$$C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]] = y_{2k},$$

$$f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]] = y_{2k-1},$$

$$C_2[x, f[\ldots C_k[x, f[R[x]]]]] = y_{2k-2},$$

$$f[C_3[x, f[\ldots C_k[x, f[R[x]]]]]] = y_{2k-3},$$

$$\ldots$$

$$f[C_k[x, f[R[x]]]] = y_3,$$

$$C_k[x, f[R[x]]] = y_2,$$

$$f[R[x]] = y_1.$$

On the other hand, by the definition of F, we have:

$$F[x] = C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]]$$

and hence $F[x] = y_{2k}$. Again, from the correctness of F, we obtain:

$$O_F[x, y_{2k}],$$

which had to be proven.

By this we completed our proof of the *Completeness* theorem.

3.8.4 Example and Discussion

In order to illustrate the *Soundness* and the *Completeness* theorems, and the class of recursive functions which may contain nested recursive definitions, we consider the McCarthy 91 function, which is viewed as a "challenge problem" for automated program verification.

The program itself is defined as follows:

$$M[x] = \textbf{ If } x \geq 101 \textbf{ then } x - 10 \textbf{ else } M[M[x+11]], \tag{3.149}$$

with the specification:

$$(\forall x)\ (I_M[x] \iff x \in \mathbb{N}) \tag{3.150}$$

and

$$(\forall x, y)\ (O_M[x, y] \iff (x < 101\ \wedge\ y = 91) \vee (x \geq 101\ \wedge\ y = x - 10)). \tag{3.151}$$

The (automatically generated) conditions for **coherence** are:

$$(\forall x : x \in \mathbb{N})\ (x \geq 101\ \Rightarrow\ \mathbb{T}) \tag{3.152}$$

$$(\forall x : x \in \mathbb{N})\ (x \ngeq 101\ \Rightarrow\ x + 11 \in \mathbb{N}) \tag{3.153}$$

$$(\forall x : x \in \mathbb{N})\ (x \ngeq 101\ \Rightarrow\ \mathbb{T}) \tag{3.154}$$

$$(\forall x, y_1, y_2, y_3, y_4 : x \in \mathbb{N}) \tag{3.155}$$

$$(x \ngeq 101\ \wedge\ ((x + 11 < 101\ \wedge\ y_1 = 91) \vee (x + 11 \geq 101\ \wedge\ y_1 = x + 11 - 10))\ \wedge$$
$$\wedge\ ((y_2 < 101\ \wedge\ y_3 = 91) \vee (y_2 \geq 101\ \wedge\ y_3 = y_2 - 10))\ \wedge\ y_1 = y_2\ \wedge\ y_3 = y_4$$
$$\Longrightarrow$$
$$\mathbb{T}\ \wedge\ \mathbb{T}\ \wedge\ y_2 \in \mathbb{N}\ \wedge\ y_4 \in \mathbb{N})$$

One sees that the formulae (3.152) and (3.154) hold, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the $x - 10$ ($S[x] = x - 10$) and the projection functions ($C_1[x, y] = y$ and $C_2[x, y] = y$).

The formulae (3.153) and (3.155) are easy consequences of the elementary theory of naturals.

For the further check of **correctness** the generated conditions are:

$$(\forall x : x \in \mathbb{N}) \tag{3.156}$$
$$(x \geq 101 \Longrightarrow (x < 101\ \wedge\ x - 10 = 91) \vee (x \geq 101\ \wedge\ x - 10 = x - 10)).$$

$$(\forall x, y_1, y_2, y_3, y_4 : x \in \mathbb{N}) \tag{3.157}$$
$$(n \ngeq 101\ \wedge\ ((x + 11 < 101\ \wedge\ y_1 = 91) \vee (x + 11 \geq 101\ \wedge\ y_1 = x + 11 - 10))\ \wedge$$
$$\wedge\ ((y_2 < 101\ \wedge\ y_3 = 91) \vee (y_2 \geq 101\ \wedge\ y_3 = y_2 - 10))\ \wedge\ y_1 = y_2\ \wedge\ y_3 = y_4$$
$$\Longrightarrow$$
$$(x < 101\ \wedge\ y_4 = 91) \vee (x \geq 101\ \wedge\ y_4 = x - 10)).$$

The proofs of these verification conditions are straightforward, and thus the program (3.149) is partially correct with respect to the specification (3.150), (3.151).

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program M is now almost the same as the previous one, but in the base case (when $x \geq 101$) the return value is $x - 11$. The new (wrong) definition of M is:

$$M[x] = \textbf{ If } x \geq 101 \textbf{ then } x - 11 \textbf{ else } M[M[x + 11]], \tag{3.158}$$

After generating the verification conditions, we see that all but one hold, namely:

$$(\forall x : x \in \mathbb{N}) \tag{3.159}$$
$$(x \geq 101 \implies (x < 101 \ \wedge \ x - 11 = 91) \vee (x \geq 101 \ \wedge \ x - 11 = x - 10)).$$

which reduces to proving:

$$x - 11 = x - 10.$$

Therefore, according to the completeness of the method, we conclude that the program M does not satisfy its specification. Moreover, the failed proof gives a hint for "debugging": we need to change the return value in the case $x \geq 101$ to $x - 10$.

A similar experiment shows, that in fact, the input condition (3.151), that is $x \in \mathbb{N}$ is too strong, and could be successfully replaced by $x \in \mathbb{Z}$.

3.9 Termination

In this section we present a specialized strategy for proving termination of recursive functional programs. The detailed termination proofs may in many cases be skipped, because the termination conditions are reusable and thus collected in specialized libraries. Enlargement of the libraries is possible by proving termination of each candidate, but also by taking new elements directly from existing libraries.

It is well agreed, that proving correctness of recursive programs is still challenging, especially when by correctness is meant total correctness. There are various approaches, however, there is no (and cannot be) general recipe. Termination proofs exposed in classical books (e.g., [44]) are very comprehensive, however, their orientation is theoretical rather than practical. On the other hand there are various tools for proving program correctness automatically or semiautomatically, (see, e.g., [28],[4]), and this is where the contribution of this section falls into.

Termination proofs of individual programs are, in general, expensive from the automatic theorem proving point of view—they normally involve induction and thus an induction prover must be applied. In some cases, program termination, however, may be ensured—and this is the main contribution of this chapter—by matching against *simplified versions* (of programs) collected in specialized libraries.

As we already saw, proving total correctness of a program is split into three distinct steps: first—proving coherence, second—proving partial correctness, and third—proving termination.

Furthermore, partial correctness and termination, expressed as verification conditions which themselves may be proven without taking into account their order. Moreover, as we have shown in the previous sections, a coherent program (of a certain recursive type) is totaly correct if and only if its verification conditions hold as logical formulae.

Proving any of the three kinds of verification conditions has its own difficulty, however, our experience shows that proving coherence is relatively easy, proving partial correctness is more difficult and proving the termination verification condition (it is only one condition) is in general the most difficult one.

The proof typically needs an induction prover and the induction step may sometimes be difficult to find. Fortunately, due to the specific structure, the proof is not always necessary, and this is what we discuss here.

3.9.1 Libraries of Terminating Programs

In this subsection we describe the idea of proving termination of recursive programs by creating and exploring libraries of terminating programs, and thus avoiding redundancy of induction proofs. The core idea is that different recursive programs may have the same *simplified version*.

Let us reconsider the following very simple recursive program for computing the factorial function:

$$Fact[n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } n * Fact[n-1], \tag{3.160}$$

with the specification of $Fact$, Input:

$$\forall n\ (I_{Fact}[n] \Longleftrightarrow n \in \mathbb{N}) \tag{3.161}$$

and Output:

$$\forall n, m\ (O_{Fact}[n,m] \Longleftrightarrow n! = m). \tag{3.162}$$

The verification condition for the termination of $Fact$ is expressed using a *simplified version* of the initial function:

$$Fact'[n] = \textbf{ If } n = 0 \textbf{ then } \mathbb{T} \textbf{ else } Fact'[n-1], \tag{3.163}$$

namely, the verification condition is

$$(\forall n : n \in \mathbb{N})\ (Fact'[n] = \mathbb{T}), \tag{3.164}$$

where $\mathbb{T}$ expresses the logical constant *true*.

More generally, when having a recursive program which may fit to the schema:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R_1[x]], \ldots, F[R_k[x]]], \tag{3.165}$$

where Q is a predicate and S, C, R_1, $\ldots$, R_k are auxiliary functions whose total correctness is assumed, the corresponding *simplified version* of F is:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } \mathbb{T} \textbf{ else } F'[R_1[x]] \wedge \cdots \wedge F'[R_k[x]],$$

which only depends on Q, R_1, $\ldots$, R_k. It is obtained by replacing the function S by $\mathbb{T}$, and the function C by the logical *and* for combining the recursive calls. Namely, the termination condition is

$$(\forall x\ : I_F[x])\ (F'[x] = \mathbb{T}), \tag{3.166}$$

which must be proven, based on the logical formulae corresponding to the definition of F' and the theory of the domain of $Q, R_1, \ldots, R_k$.

Moreover, proving that

$$(\forall x\ : I_F[x])\ (F'[x] = \mathbb{T}) \tag{3.167}$$

is equivalent to proving termination of $F'[x]$, for all x satisfying $I_F[x]$ (it is so, because if $F'[x]$ terminates it returns $\mathbb{T}$, and vice versa), which may be used alternatively.

Note, that different recursive programs may have the same *simplified version*. Let us now consider another very simple recursive program for computing the sum function:

$$Sum[n] = \textbf{ If } n = 0 \textbf{ then } 0 \textbf{ else } n + Sum[n-1], \tag{3.168}$$

with the specification of Sum, Input:

$$\forall n\ (I_{Sum}[n] \Longleftrightarrow n \in \mathbb{N}) \tag{3.169}$$

and Output:

$$\forall n, m\ (O_{Sum}[n,m] \Longleftrightarrow \frac{n * (n+1)}{2} = m). \tag{3.170}$$

The verification condition for the termination of Sum is expressed using a *simplified version* of the initial function:

$$Sum'[n] = \textbf{ If } n = 0 \textbf{ then } \mathbb{T} \textbf{ else } Sum'[n-1], \tag{3.171}$$

namely, the verification condition is

$$(\forall n : n \in \mathrm{N})\ (Sum'[n] = \mathbb{T}). \tag{3.172}$$

Notably, the termination verification conditions (3.164) and (3.172) of the programs (3.163) and (3.171) are the same.

Primitive recursive functions (3.173) are very broadly used in practice. It is well known that they always terminate [9]. However, proving that a function is primitive recursive has to be carried over.

$$Prim[n] = \textbf{ If } n = 0 \textbf{ then } S[n] \textbf{ else } C[n, Prim[n-1]], \tag{3.173}$$

Now, the simplified version of (3.173) is:

$$Prim'[n] = \textbf{ If } n = 0 \textbf{ then } \mathbb{T} \textbf{ else } Prim'[n-1], \tag{3.174}$$

namely, the verification condition is

$$(\forall n : n \in \mathrm{N})\ (Prim'[n] = \mathbb{T}). \tag{3.175}$$

which is the same as (3.164).

For serving the termination proofs, we are now creating libraries containing *simplified versions* together with their input conditions, whose termination is proven. The proof of the termination may now be skipped if the *simplified version* is already in the library and this membership check is much easier than an induction proof—it only involves matching against simplified versions.

Starting from a small library—actually it is not only one, but more, because each recursive schema has several domain based libraries—we intend to enlarge it. One way of doing so is by carrying over the whole proof of any new candidate, appearing during a verification process.

3.9.2 Enlargement within libraries

Enlargement within a library is also possible by applying special knowledge retrieval. As we have seen, termination depends on the *simplified version* F' and on the input condition I_F. Considering again the factorial example (3.160), in order to prove its termination we need to prove (3.164). Assume, now the pair (3.163),(3.164) is in our library. We may now strengthen the input condition I_{Fact} and actually produce a new one:

$$I_{F-new}[n] \iff (n \in \mathrm{N} \wedge n \geq 100).$$

The *simplified version* $Fact'$ remains the same (3.163) – we did not change the initial program (3.160), however, the termination condition becomes:

$$(\forall n : n \in \mathrm{N} \wedge n \geq 100)\ (Fact'[n] = \mathbb{T}), \tag{3.176}$$

and (after proving them) we add it to the library. It is easy to see that any new version of a simplified program which is obtained by strengthening the input condition can also be included in the library without further proof. Assume

$$(\forall x\ : I_F[x])\ (F'[x] = \mathbb{T})$$

is a member of a library. Then for any "stronger" input condition $I_{F-strng}$, we have:

$$I_{F-strng}[x] \Longrightarrow I_F[x],$$

and thus

$$(\forall x \ : I_{F-strng}[x]) \ (F'[x] = \mathbb{T}).$$

This is of course not the case for weakening the input condition. Consider the following weakening of I_{Fact}:

$$I_{F-real}[n] \Longleftrightarrow (n \in \mathrm{R}),$$

which leads to nontermination of our $Fact'$ as defined in (3.164), that is:

$$(\forall n \ : n \in \mathrm{R}) \ (Fact'[n] = \mathbb{T}),$$

which does not hold.

Strengthening of input conditions leads to preserving the termination properties and thus enlarging a library without additional proof is possible. However, for a fixed *simplified version*, keeping (and collecting in some cases) the weakest input condition is the most efficient strategy, because then proving the implication from stronger to weaker condition is relatively easier.

3.9.3 Conclusions

Termination proofs are reduced to proving properties of simplified versions, which themselves are reusable. They may be done by using a *Theorema* prover (see, e.g., [14],[17]). However, delivering the proof problem itself to another specialized tool (e.g., [30],[18]) is also possible. Enlarging the libraries by taking (and adopting) *simplified versions* directly from other libraries (e.g., the *Coq Library* [5]) can be considered as well.

Alternatively, one may add to the libraries simplified versions of conjectures whose proofs are not provided, but well believed or considered as obvious. For instance, the well studied Collatz conjecture [41], [42] whose termination is an open problem since 1937 may be included into a library, since it has been explored and checked mechanically for extremely big numbers. However, in order to keep at the safe side, one may add as a precondition that the input will not exceed the number-of-today's achievement.

3.10 Examples and Discussion

After developing a theory, it is always recommendable to give some relevant examples. Even though, in this thesis, most of the examples appear together with the theoretical observations, or immediately after them, in this section we collected a couple of examples which did not find their place at the other sections. However, they may still be of some interest to the reader.

Although the examples presented here appear to be relatively simple, they already demonstrate the usefulness of our approach in the general case. We aim at extending these experiments to industrial-scale examples, which are in fact not more complex from the mathematical point of view.

The last two examples (3.10.7) and (3.10.8) are dedicated to the discussion about possible applications of our research in domains which seem to be a bit far from program verification. However, we believe that the methods developed within the frame of this thesis may well be accepted and specifically adapted by researchers from other fields of Mathematics, Computer Science, Physics, etc. More specifically, we address two issues, namely

- Teaching Formal Methods to graduate students
- Algorithm investigation for scientific computing.

3.10.1 Factorial

The first example books on programming start with is that of factorial. Besides the fact that it is very elementary and easy to explain and understand, it may be used as a standard example for comparing tools and strategies for program verification.

The emphasis we want to put here is on the similarities of the program definition, the logical formulae expressed as the verification conditions and the properties of the mathematical function factorial.

Consider the program $Fact$, for computing the factorial function:

$$Fact[x] = \textbf{ If } x = 0 \textbf{ then } 1 \textbf{ else } x * Fact[x-1], \tag{3.177}$$

with the specification:

$$(\forall x)\ (I_{Fact}[x] \iff x \in \mathbb{N}),$$

$$(\forall x, y)\ (O_{Fact}[x, y] \iff y = x!).$$

Before starting with the essential part of the verification, we first check if $Fact$ is coherent with respect to its specification, the auxiliary programs and their specifications. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies \mathbb{T}) \tag{3.178}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies x - 1 \in \mathbb{N}) \tag{3.179}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies \mathbb{T}) \tag{3.180}$$

$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge [x-1]! = y) \implies \mathbb{T}). \tag{3.181}$$

As we can see, the conditions (3.178), (3.180) and (3.181) are trivial to prove, because we have $\mathbb{T}$ at the right hand side of an implication. The origin of these $\mathbb{T}$ are the preconditions of the auxiliary functions $\lambda x.1$, the minus one function $\lambda x.x - 1$, and multiplication $\lambda x, y.x * y$.

In fact, only (3.179) requires a proofs, however, it is easily tractable in the theory of natural numbers.

After we are convinced that $Fact$ is coherent, we instantiate the relevant verification conditions for proving correctness:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies 1 = x!) \tag{3.182}$$

$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge y = [x-1]! \implies x * y = x!) \tag{3.183}$$

We see, that (3.182), (3.183) and are tractable in the theory of natural numbers.

Now we need to prove the termination of $Fact$, that is:

$$(\forall x : x \in \mathbb{N})\ (Fact'[x] = \mathbb{T}) \tag{3.184}$$

where:

$$Fact'[x] = \textbf{ If } x = 0 \textbf{ then } \mathbb{T} \textbf{ else } Fact'[x-1]. \tag{3.185}$$

We have arrived to the most popular simplified version, namely the primitive recursive one and thus we are done.

3.10.2 Summation of a number

The next example we consider is the summation of a number:

$$\sum_{i=1}^{x} i.$$

The emphasis we want to put here is on the similarities of the verification conditions we obtain here and the verification conditions from the previous example (3.10.1).

In our opinion, many practical programs, when verifying, will have similar verification conditions. This fact could be considered as a research problem by the Mathematical Knowledge Management community.

Consider the program Sum, for computing the summation of a number function:

$$Sum[x] = \textbf{ If } x = 0 \textbf{ then } 0 \textbf{ else } x + Sum[x-1], \tag{3.186}$$

with the specification:

$$(\forall x)\ (I_{Sum}[x] \iff x \in \mathbb{N}),$$

$$(\forall x, y)\ (O_{Sum}[x, y] \iff y = \frac{x * (x+1)}{2}).$$

Even though, this program is suppose to compute the summation of a number, we provide here an alternative output specification.

Before starting with the essential part of the verification, we first check if Sum is coherent with respect to its specification, the auxiliary programs and their specifications. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies \mathbb{T}) \tag{3.187}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies x - 1 \in \mathbb{N}) \tag{3.188}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies \mathbb{T}) \tag{3.189}$$

$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge \frac{(x-1) * ((x-1)+1)}{2} = y) \implies \mathbb{T}). \tag{3.190}$$

As we can see, the conditions (3.187), (3.189) and (3.190) are trivial to prove, because we have $\mathbb{T}$ at the right hand side of an implication. The origin of these $\mathbb{T}$ are the preconditions of the auxiliary functions $\lambda x.0$, the minus one function $\lambda x.x - 1$, and addition $\lambda x, y.x + y$.

In fact, only (3.188) requires a proofs, however, it is easily tractable in the theory of natural numbers.

Note that in the previous example (3.10.1), we have the same condition (3.179).

After we are convinced that Sum is coherent, we instantiate the relevant verification conditions for proving correctness:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies 0 = \frac{x * (x+1)}{2}) \tag{3.191}$$

$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \land y = \frac{(x-1) * ((x-1)+1)}{2} \implies x + y = \frac{x * (x+1)}{2}) \tag{3.192}$$

We see, that (3.191), (3.192) and are tractable in the theory of natural numbers.

Note that these are the specific verification conditions. In fact, the proofs of all the other verification conditions, that is, coherence and termination, may be reusable, because they are the same for different programs.

Now we need to prove the termination of Sum, that is:

$$(\forall x : x \in \mathbb{N})\ (Sum'[x] = \mathbb{T}) \tag{3.193}$$

where:

$$Sum'[x] = \textbf{ If } x = 0 \textbf{ then } \mathbb{T} \textbf{ else } Sum'[x-1]. \tag{3.194}$$

We again arrived at the most popular simplified version, namely the primitive recursive one and thus we are done.

3.10.3 Floor of a real number

The next example we consider is the floor of a real number.

The purpose of showing this example is to demonstrate the ability of our method in domains different from $\mathbb{N}$. In fact, we do not give any restrictions on the possible domains on which the programs are executed and verified. For this particular example we take $\mathbb{R}$ as our domain.

Consider the program $Floor$, for computing the floor of a real nonnegative number:

$$Floor[x] = \textbf{If } 0 \leq x < 1 \textbf{ then } 0 \textbf{ else } 1 + Floor[x-1], \tag{3.195}$$

with the specification:

$$(\forall x)\ (I_{Floor}[x] \iff x \in \mathbb{R} \ \wedge \ x \geq 0),$$

$$(\forall x, y)\ (O_{Floor}[x, y] \iff y \in \mathbb{N} \ \wedge \ x - y < 1 \ \wedge \ y \leq x).$$

Before starting with the essential part of the verification, we first check if $Floor$ is coherent with respect to its specification, the auxiliary programs and their specifications. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x : x \in \mathbb{R} \ \wedge \ x \geq 0)\ (0 \leq x < 1 \implies \mathbb{T}) \tag{3.196}$$

$$(\forall x : x \in \mathbb{R} \ \wedge \ x \geq 0)\ (\neg(0 \leq x < 1) \implies x - 1 \in \mathbb{R} \ \wedge \ x - 1 \geq 0) \tag{3.197}$$

$$(\forall x : x \in \mathbb{R} \ \wedge \ x \geq 0)\ (\neg(0 \leq x < 1) \implies \mathbb{T}) \tag{3.198}$$

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\ (\neg(0 \leq x < 1) \wedge y \in \mathbb{N} \wedge (x-1) - y < 1 \wedge y \leq (x-1)) \implies \mathbb{T}). \tag{3.199}$$

As we can see, the conditions (3.196), (3.198) and (3.199) are trivial to prove, because we have $\mathbb{T}$ at the right hand side of an implication. The origin of these $\mathbb{T}$ are the preconditions of the auxiliary functions $\lambda x.0$, the minus one function $\lambda x.x - 1$, and the plus one function $\lambda x.1 + x$.

In fact, only (3.197) requires a proofs, however, it is easily tractable in the theory of real numbers.

After we are convinced that $Floor$ is coherent, we instantiate the relevant verification conditions for proving correctness:

$$(\forall x : x \in \mathbb{R} \ \wedge \ x \geq 0)\ (0 \leq x < 1 \implies 0 \in \mathbb{N} \ \wedge \ x - 0 < 1 \ \wedge \ 0 \leq x) \tag{3.200}$$

$$\begin{gathered}(\forall x : x \in \mathbb{R} \ \wedge \ x \geq 0)\ (\neg(0 \leq x < 1) \wedge y \in \mathbb{N} \ \wedge \ (x-1) - y < 1 \ \wedge \ y \leq (x-1) \\ \implies \\ 1 + y \in \mathbb{N} \ \wedge \ x - (1 + y) < 1 \ \wedge \ (1 + y) \leq x).\end{gathered} \tag{3.201}$$

We see, that (3.200) and (3.201) are tractable in the theory of real numbers.

Now we need to prove the termination of $Floor$, that is:

$$(\forall x : x \in \mathbb{R} \ \wedge \ x \geq 0)\ (Floor'[x] = \mathbb{T}) \tag{3.202}$$

where:

$$Floor'[x] = \ \textbf{If}\ 0 \leq x < 1\ \textbf{then}\ \mathbb{T}\ \textbf{else}\ Floor'[x-1]. \tag{3.203}$$

This is a new simplified version, and one has to prove its termination, which easily possible by induction. We split the $\mathbb{R}^+$ into intervals with length one and then perform normal induction over the naturals.

3.10.4 A wrong version of Floor

The next example we consider is a wrong version of the floor function. We introduce here a bug in order to explore the verification conditions in such a situation. Moreover, a distinctive feature of our approach is the hint on "what is wrong" in case of a verification failure.

Consider the program $WrFloor$, for computing the floor of a real nonnegative number:

$$WrFloor[x] = \textbf{ If } 0 \leq x < 1 \textbf{ then } 5 \textbf{ else } 1 + WrFloor[x-1], \tag{3.204}$$

with the specification:

$$(\forall x)\,(I_{WrFloor}[x] \iff x \in \mathbb{R} \wedge x \geq 0),$$

$$(\forall x, y)\,(O_{WrFloor}[x, y] \iff y \in \mathbb{N} \wedge x - y < 1 \wedge y \leq x).$$

The the specification of $WrFloor$ is same as the $Floor$, presented in (3.10.3), however, in the definition of $WrFloor$ we introduced a bug, namely when $0 \leq x < 1$, $WrFloor[x] = 5$, in contrast to $Floor[x] = 0$.

Before starting with the essential part of the verification, we first check if $WrFloor$ is coherent with respect to its specification, the auxiliary programs and their specifications. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\,(0 \leq x < 1 \implies \mathbb{T}) \tag{3.205}$$

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\,(\neg(0 \leq x < 1) \implies x - 1 \in \mathbb{R} \wedge x - 1 \geq 0) \tag{3.206}$$

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\,(\neg(0 \leq x < 1) \implies \mathbb{T}) \tag{3.207}$$

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\,(\neg(0 \leq x < 1) \wedge y \in \mathbb{N} \wedge (x-1) - y < 1 \wedge y \leq (x-1)) \implies \mathbb{T}). \tag{3.208}$$

As we can see, the conditions (3.205), (3.207) and (3.208) are trivial to prove, because we have $\mathbb{T}$ at the right hand side of an implication. The origin of these $\mathbb{T}$ are the preconditions of the auxiliary functions $\lambda x.5$, the minus one function $\lambda x.x - 1$, and the plus one function $\lambda x.1 + x$.

Only (3.206) requires a proofs, however, it is easily tractable in the theory of real numbers.

In fact, all the conditions here are the same as in the correct version of $Floor$ (3.10.3).

After we are convinced that $WrFloor$ is coherent, we instantiate the relevant verification conditions for proving correctness:

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\,(0 \leq x < 1 \implies 5 \in \mathbb{N} \wedge x - 5 < 1 \wedge 5 \leq x) \tag{3.209}$$

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\,(\neg(0 \leq x < 1) \wedge y \in \mathbb{N} \wedge (x-1) - y < 1 \wedge y \leq (x-1) \tag{3.210}$$

$$\Longrightarrow$$

$$1 + y \in \mathbb{N} \wedge x - (1 + y) < 1 \wedge (1 + y) \leq x).$$

$$(\forall x : x \in \mathbb{R} \wedge x \geq 0)\ (WrFloor'[x] = \mathbb{T}) \tag{3.211}$$

where:

$$WrFloor'[x] = \textbf{ If } 0 \leq x < 1 \textbf{ then } \mathbb{T} \textbf{ else } WrFloor'[x - 1]. \tag{3.212}$$

We see, that (3.210), (3.211) are tractable in the theory of real numbers and they are the same as in the correct version of $Floor$.

Now, for this buggy version of $WrFloor$ we see that all the verification conditions remain the same, except one, namely, (3.209). Therefore, according to the *completeness* of the method, we conclude that the program $WrFloor$ does not satisfy its specification.

Furthermore, in order to demonstrate how a bug might be located in an automatic manner, we have a broad discussion on that topic in (3.10.7).

3.10.5 Remainder Rem in division of integers

In arithmetic, when the result of the division of two integers cannot be expressed with an integer quotient, the remainder is the amount "left over." The next example we consider is the remainder function Rem.

The purpose of showing this example is to demonstrate the ability of our method in vector domains, that is, the arguments x, y, z could be not only single variables but vectors (tuples) as well.

Consider the program Rem, for computing the remainder of the division of two natrurals:

$$Rem[x,y] = \textbf{ If } x < y \textbf{ then } x \textbf{ else } Rem[x-y,y], \tag{3.213}$$

with the specification:

$$(\forall x, y)\ (I_{Rem}[x,y] \iff x \in \mathbb{N} \wedge y \in \mathbb{N}^{+}),$$

$$(\forall x, y, z)\ (O_{Rem}[x,y,z] \iff (\exists q : q \in \mathbb{N})\ (x = z + y * q \ \wedge\ z < y)).$$

Before starting with the essential part of the verification, we first check if Rem is coherent with respect to its specification, the auxiliary programs and their specifications. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x, y :\ x \in \mathbb{N} \ \wedge\ y \in \mathbb{N}^{+})\ (x < y \implies \mathbb{T}) \tag{3.214}$$

$$(\forall x, y :\ x \in \mathbb{N} \ \wedge\ y \in \mathbb{N}^{+})\ (\neg(x < y) \implies x - y \in \mathbb{N} \ \wedge\ y \in \mathbb{N}^{+}) \tag{3.215}$$

$$(\forall x, y :\ x \in \mathbb{N} \ \wedge\ y \in \mathbb{N}^{+})\ (\neg(x < y) \implies \mathbb{T}) \tag{3.216}$$

$$(\forall x, y, z :\ x \in \mathbb{N} \wedge y \in \mathbb{N}^{+})\ (\neg(x < y) \wedge (\exists q : q \in \mathbb{N})\ (x - y = z + y * q \wedge z < y)) \implies \mathbb{T}). \tag{3.217}$$

As we can see, the conditions (3.214), (3.216) and (3.217) are trivial to prove, because we have $\mathbb{T}$ at the right hand side of an implication. The origin of these $\mathbb{T}$ are the preconditions of the auxiliary functions: the identity $\lambda x.x$, the minus function $\lambda x, y.x - y$, and the projection $\lambda x, y.y$.

In fact, only (3.215) requires a proofs, however, it is easily tractable in the theory of natural numbers.

After we are convinced that Rem is coherent, we instantiate the relevant verification conditions for proving correctness:

$$(\forall x, y :\ x \in \mathbb{N} \ \wedge\ y \in \mathbb{N}^{+})\ (x < y \implies (\exists q : q \in \mathbb{N})\ (x = x + y * q \ \wedge\ x < y)) \tag{3.218}$$

$$(\forall x, y, z :\ x \in \mathbb{N} \ \wedge\ y \in \mathbb{N}^{+})\ (\neg(x < y) \wedge (\exists q : q \in \mathbb{N})\ (x - y = z + y * q \ \wedge\ z < y) \tag{3.219}$$

$$\Longrightarrow$$
$$(\exists q : q \in \mathbb{N})\ (x = z + y * q\ \wedge\ z < y)).$$

We see, that (3.218) and (3.219) are tractable in the theory of natural numbers.

Now we need to prove the termination of Rem, that is:

$$(\forall x, y :\ x \in \mathbb{N}\ \wedge y \in \mathbb{N}^{+})\ (Rem'[x, y] = \mathbb{T}) \tag{3.220}$$

where:

$$Rem'[x, y] = \textbf{ If } x < y \textbf{ then } \mathbb{T} \textbf{ else } Rem'[x - y, y]. \tag{3.221}$$

This is a new simplified version, and one has to prove its termination. For that proof, we would suggest to use induction on q, where $x = Rem[x, y] + y * q$ for any x and y: $x \in \mathbb{N}\ \wedge\ y \in \mathbb{N}^{+}$.

3.10.6 Even and Odd

This example is dedicated to demonstrating mutual recursive definitions—actually what are the necessary and sufficient conditions for the program to be correct.

Consider the system E, defined with the help of the functions EV and OD for checking whether given natural number is even or not:

$$F[x] = EV[x], \tag{3.222}$$

where:

$$EV[x] = \textbf{ If } x = 0 \textbf{ then } \mathbb{T} \textbf{ else } OD[x-1], \tag{3.223}$$

$$OD[x] = \textbf{ If } x = 0 \textbf{ then } \mathbb{F} \textbf{ else } EV[x-1], \tag{3.224}$$

with the specification:

$$(\forall x)\ (I_{EV}[x] \iff x \in \mathbb{N}),$$

$$(\forall x, y)\ (O_{EV}[x, y] \iff (Even[x] \ \wedge \ y = \mathbb{T}) \ \vee \ (Odd[x] \ \wedge \ y = \mathbb{F})),$$

$$(\forall x)\ (I_{OD}[x] \iff x \in \mathbb{N}),$$

$$(\forall x, y)\ (O_{OD}[x, y] \iff (Even[x] \ \wedge \ y = \mathbb{F}) \ \vee \ (Odd[x] \ \wedge \ y = \mathbb{T})),$$

The program E is suppose to check whether a given natural number is even or not, that is, for any natural number x, if x is even number, it should return $\mathbb{T}$, and if x is odd, $\mathbb{F}$.

Before starting with the essential part of the verification, we first check if the program E is coherent. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies \mathbb{T}) \tag{3.225}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies x - 1 \in \mathbb{N}) \tag{3.226}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies \mathbb{T}) \tag{3.227}$$

$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (Even[x-1] \ \wedge \ y = \mathbb{F}) \ \vee \ (Odd[x-1] \ \wedge \ y = \mathbb{T}) \implies \mathbb{T}) \tag{3.228}$$

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies \mathbb{T}) \tag{3.229}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies x - 1 \in \mathbb{N}) \tag{3.230}$$

$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies \mathbb{T}) \tag{3.231}$$

$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (\mathit{Even}[x-1]\ \wedge\ y = \mathbb{T})\ \vee\ (\mathit{Odd}[x-1]\ \wedge\ y = \mathbb{F}) \implies \mathbb{T}). \tag{3.232}$$

As we can see, most of the conditions are trivial to prove, because we have $\mathbb{T}$ at the right hand side of an implication. The origin of these $\mathbb{T}$ are the preconditions of some of the auxiliary functions, e.g., the constant function $\lambda x.\mathbb{T}$, the minus one function $\lambda x.x-1$, etc.

In fact, only (3.226) and (3.230) require proofs, however, they are easily tractable in the theory of natural numbers.

After we are convinced that E is coherent, we instantiate the relevant verification conditions for proving correctness:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies (\mathit{Even}[x]\ \wedge\ \mathbb{T} = \mathbb{T})\ \vee\ (\mathit{Odd}[x]\ \wedge\ \mathbb{T} = \mathbb{F})) \tag{3.233}$$

$$\begin{gathered}(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (\mathit{Even}[x-1]\ \wedge\ y = \mathbb{F})\ \vee\ (\mathit{Odd}[x-1]\ \wedge\ y = \mathbb{T})\\ \implies \\ (\mathit{Even}[x]\ \wedge\ y = \mathbb{T})\ \vee\ (\mathit{Odd}[x]\ \wedge\ y = \mathbb{F}))\end{gathered} \tag{3.234}$$

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies (\mathit{Even}[x]\ \wedge\ \mathbb{T} = \mathbb{T})\ \vee\ (\mathit{Odd}[x]\ \wedge\ \mathbb{T} = \mathbb{F})) \tag{3.235}$$

$$\begin{gathered}(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (\mathit{Even}[x-1]\ \wedge\ y = \mathbb{T})\ \vee\ (\mathit{Odd}[x-1]\ \wedge\ y = \mathbb{F})\\ \implies \\ (\mathit{Even}[x]\ \wedge\ y = \mathbb{F})\ \vee\ (\mathit{Odd}[x]\ \wedge\ y = \mathbb{T}))\end{gathered} \tag{3.236}$$

We see, that all of verification conditions are tractable in the theory of natural numbers. Essentially, one has to prove that: if $x \neq 0$ and $\mathit{Even}[x-1]$ then $\mathit{Odd}[x]$, and, complementary: if $x \neq 0$ and $\mathit{Odd}[x-1]$ then $\mathit{Even}[x]$.

Now we need to prove the termination of E, that is:

$$(\forall x : x \in \mathbb{N})\ (EV'[x] = \mathbb{T}) \tag{3.237}$$

where:

$$EV'[x] = \textbf{ If } x = 0 \textbf{ then } \mathbb{T} \textbf{ else } OD'[x-1] \tag{3.238}$$

$$OD'[x] = \textbf{ If } x = 1 \textbf{ then } \mathbb{T} \textbf{ else } EV'[x-1]. \tag{3.239}$$

It is now interesting to see, that EV' and OD' have the same definitions (up to renaming), and we can merge into one, namely:

$$F'[x] = \textbf{ If } x = 1 \textbf{ then } \mathbb{T} \textbf{ else } F'[x-1]. \tag{3.240}$$

Now, we see that we have arrived to the most popular simplified version, namely the primitive recursive one and thus we are done.

3.10.7 Binary Powering

We consider again a powering function P, however we provide this time a different implementation, namely *binary powering*:

$$P[x, n] = \textbf{If } n = 0 \textbf{ then } 1 \tag{3.241}$$
$$\textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \tag{3.242}$$
$$\textbf{else } x * P[x * x, (n-1)/2]. \tag{3.243}$$

This program in the context of the theory of real numbers, and in the following formulae, all variables are implicitly assumed to be real. Additional type information (e. g. $n \in \mathbb{N}$) may be explicitly included in some formulae.

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x, n] = x^n. \tag{3.244}$$

The (automatically generated) conditions for **coherence** are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow \mathbb{T}) \tag{3.245}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \land \mathrm{Even}[n]\ \Rightarrow \mathrm{Even}[n]) \tag{3.246}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \land \neg\mathrm{Even}[n]\ \Rightarrow \mathrm{Odd}[n]) \tag{3.247}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \land \mathrm{Even}[n] \land m = (x * x)^{n/2} \Rightarrow \mathbb{T}) \tag{3.248}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \land \neg\mathrm{Even}[n] \land m = (x * x)^{(n-1)/2}\ \Rightarrow \mathbb{T}) \tag{3.249}$$

One sees that the formulae (3.245), (3.248) and (3.249) are trivial, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the 1 *constant-function-one* and the $*$ *multiplication*.

The formulae (3.246) and (3.247) are easy consequences of the elementary theory of reals and naturals. For the further check of **correctness** the generated conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 1 = x^n) \tag{3.250}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \land \mathrm{Even}[n]\ \Rightarrow n/2 \in \mathbb{N}) \tag{3.251}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \land \mathrm{Even}[n] \land m = (x * x)^{n/2}\ \Rightarrow m = x^n) \tag{3.252}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \land \neg\mathrm{Even}[n]\ \Rightarrow (n-1)/2 \in \mathbb{N}) \tag{3.253}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \land \neg\mathrm{Even}[n] \land m = (x * x)^{(n-1)/2}\ \Rightarrow x * m = x^n) \tag{3.254}$$

$$(\forall x, n : n \in \mathbb{N})\ P'[x, n] = 0, \tag{3.255}$$

where

$$\begin{aligned} P'[x, n] = \quad & \textbf{If } n = 0 \textbf{ then } 0 & (3.256)\\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P'[x * x, n/2] & (3.257)\\ & \textbf{else } P'[x * x, (n-1)/2]. & (3.258) \end{aligned}$$

The proofs of these verification conditions are straightforward.

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program P is now almost the same as the previous one, but in the base case (when $n = 0$) the return value is 0.

$$\begin{aligned} P[x, n] = \quad & \textbf{If } n = 0 \textbf{ then } 0 & (3.259)\\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] & (3.260)\\ & \textbf{else } x * P[x * x, (n-1)/2]. & (3.261) \end{aligned}$$

Now, for this buggy version of P we may see that all the respective verification conditions remain the same—and thus the program is correct—except one, namely, (3.250) is now:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 0 = x^n) \tag{3.262}$$

which itself reduces to:

$$0 = 1$$

(because we consider a theory where $0^0 = 1$).

Therefore, according to the *completeness* of the method, we conclude that the program P does not satisfy its specification. Moreover, the failed proof gives a hint for "debuging": we need to change the return value in the case $n = 0$ to 1.

Furthermore, in order to demonstrate how a bug might be located, we construct one more "buggy" example where in the "Even" branch of the program we have $P[x, n/2]$ instead of $P[x * x, n/2]$:

$$\begin{aligned} P[x, n] = \quad & \textbf{If } n = 0 \textbf{ then } 1\\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x, n/2]\\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, we may see again that all the respective verification conditions remain the same as in the original one, except one, namely, (3.252) is now:

$$(\forall x, n : n \in \mathbb{N})\ (\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x)^{n/2}\ \Rightarrow m = x^n) \tag{3.263}$$

which itself reduces to:

$$m = x^{n/2}\ \Rightarrow m = x^n$$

From here, we see that the “Even” branch of the program is problematic and one should satisfy the implication. The most natural candidate would be:

$$m = (x^2)^{n/2} \Rightarrow m = x^n$$

which finally leads to the correct version of P.

The question whether (actually how), this correction of the program could be done automatically or semi-automatically is a matter of investigation at the border between program verification and program synthesis. In our opinion, research results in that direction would have a big practical impact, and thus would be welcome.

3.10.8 Neville's Algorithm

We demonstrate our method on Neville's algorithm for polynomial interpolation [29], [56] and show how it may be validated fully automatically. We choose this example because, first, we want to demonstrate the usefulness of our framework on examples which go beyond triviality, e.g., factorial and summa, and second, we want to attract some attention from researchers working in the field of numerical analysis.

Neville's algorithm is an algorithm used for polynomial interpolation. Given n points, there is a unique polynomial of degree $n-1$ which goes through the given points. Neville's algorithm constructs this polynomial.

Neville's algorithm is based on the Newton form of the interpolating polynomial and the recursion relation for the divided differences. Algorithmically, it fits to the Fibonacci-like recursive schema.

The original problem is as follows: Given a field K, two non-empty tuples $\overline{x}$ and $\overline{a}$ over K of same length n, such that

$$(\forall i, j \ : \ i, j = 1, \ldots, n)\ (i \neq j \Rightarrow x_i \neq x_j),$$

that is, no two x_i from $\overline{x}$ are the same.

Find a polynomial p over the field K, such that

- $deg[p] \leq n-1$ and
- $(\forall i \ : \ i = 1, \ldots, n)\ (Eval[p, x_i] = a_i)$,

where the $Eval$ function evaluates a polynomial p at value x_i.

This original problem, as stated here, was solved by E. H. Neville [43] by inventing an algorithm for the construction of such a polynomial [52]. The algorithm itself may be formulated as follows:

$$p[\overline{x}, \overline{a}] = \ \textbf{If}\ \|\overline{a}\| \leq 1 \qquad (3.264)$$

$$\textbf{then}\ First[\overline{a}]$$

$$\textbf{else}\ \frac{(\mathcal{X} - First[\overline{x}])(p[Tail[\overline{x}], Tail[\overline{a}]]) - (\mathcal{X} - Last[\overline{x}])(p[Bgn[\overline{x}], Bgn[\overline{a}]])}{Last[\overline{x}] - First[\overline{x}]},$$

where we use the following notation:

- $\|\overline{a}\|$ gives the number n of elements of $\overline{a}$,
- $First[\overline{a}]$ gives the first element a_1 of $\overline{a}$,
- $Last[\overline{a}]$ gives the last element a_n of $\overline{a}$, provided $\|\overline{a}\| = n$,
- $Tail[\overline{a}]$ gives the tail of $\overline{a}$, that is, $\overline{a}$ without its first element,
- $Bgn[\overline{a}]$ gives the beginning of $\overline{a}$, that is, $\overline{a}$ without its last element, and

- $\mathcal{X}$ is a constant expressing the single polynomial of degree 1, leading coefficient 1 and free coefficient 0.

In fact, in abstract algebra $\mathcal{X}$ may also be interpreted as an indeterminate of the polynomials, that is, the variable in polynomial functions. This is a discussion which is not relevant for this presentation, however, it is very important when constructing the theory of polynomials, on which the verification conditions would have to be proven.

In order to illustrate how Neville's algorithm works, we consider the following example: $\overline{x} = \langle -1, 0, 1\rangle$ and $\overline{a} = \langle 3, 4, 7\rangle$. After executing (3.264), we obtain:

$$p[\langle -1,0,1\rangle,\ \langle 3,4,7\rangle] = \cdots = \mathcal{X}^2 + 2\mathcal{X} + 4.$$

This polynomial has a degree 2, as expected, and if we now evaluate it at the values -1, 0, and 1, we obtain:

$$Eval[\mathcal{X}^2 + 2\mathcal{X} + 4, -1] = 3,$$

$$Eval[\mathcal{X}^2 + 2\mathcal{X} + 4, 0] = 4,$$

and

$$Eval[\mathcal{X}^2 + 2\mathcal{X} + 4, 1] = 7,$$

which corresponds to the initial $\overline{a}$.

However, in order to be sure that this algorithm would always return the correct polynomial, one has to prove its correctness, and this was done by Neville himself [52].

Our contribution consists in automating the process of the correctness proof. Moreover, we will see that even if a small part of the specification is missing, which sometimes happens, the algorithm would not be correct anymore.

Verification of Neville's Algorithm

In order to verify (3.264), we first formalize the specification, and then produce the respective verification conditions. Finally, we discuss how each of these conditions may be proven in the theory of lists and tuples.

We give here some notations which we use for the formalization of the specification:

- $[\![\overline{a}]\!]_i$ gives the i^{th} element a_i of a tuple $\overline{a}$. Sometimes, a_i is used as an abbreviation for $[\![\overline{a}]\!]_i$. In addition to it, we have the restriction $1 \leq i \leq \|\overline{a}\|$.
- $IsPoly[poly]$ is a predicate standing that the expression $poly$ is a polynomial. For example $IsPoly[\mathcal{X}^2 + 2\mathcal{X} + 4]$.
- $IsTuple[\overline{a}]$ is a predicate standing that the expression $\overline{a}$ is a tuple. For example $IsTuple[\langle 3,4,7\rangle]$.
- $deg[poly]$ gives the degree of the polynomial $poly$. For example $deg[\mathcal{X}^2 + 2\mathcal{X} + 4] = 2$.
- $Eval[poly, x]$ evaluates a polynomial $poly$ at value x.

The preconditions of the functions used for the definition of (3.264) are as follows:

- $First$: $I_{First}[\overline{a}] \iff IsTuple[\overline{a}] \wedge \|\overline{a}\| \geq 1$
- $Last$: $I_{Last}[\overline{a}] \iff IsTuple[\overline{a}] \wedge \|\overline{a}\| \geq 1$
- $Tail$: $I_{Tail}[\overline{a}] \iff IsTuple[\overline{a}] \wedge \|\overline{a}\| \geq 1$
- Bgn: $I_{Bgn}[\overline{a}] \iff IsTuple[\overline{a}] \wedge \|\overline{a}\| \geq 1$
- $[\![\overline{a}]\!]_i$: $I_{Projection}[\overline{a}, i] \iff IsTuple[\overline{a}] \wedge 1 \leq i \leq \|\overline{a}\|$
- $\frac{u}{v}$: $I_{Div}[u, v] \iff v \neq 0$
- $u\, v$: $I_{Mult}[u, v] \iff \mathbb{T}$
- $u + v$: $I_{Add}[u, v] \iff \mathbb{T}$
- $u - v$: $I_{Sub}[u, v] \iff \mathbb{T}$

We are now ready to give the formal specification of (3.264). The precondition is:

$$(\forall \overline{x}, \overline{a})\ (I_p[\overline{x}, \overline{a}] \iff \tag{3.265}$$

$$\iff IsTuple[\overline{x}] \ \wedge\ IsTuple[\overline{a}] \ \wedge\ \|\overline{x}\| = \|\overline{a}\| \ \wedge\ \|\overline{a}\| \geq 1 \wedge$$

$$\wedge\ ((\forall i, j : i, j \in \mathrm{N})\ (1 \leq i, j \leq \|\overline{a}\| \ \wedge i \neq j \implies [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j))),$$

and the postcondition is:

$$(\forall \overline{x}, \overline{a})\ (O_p[\overline{x}, \overline{a}, p] \iff \tag{3.266}$$
$$\iff IsPoly[p] \ \wedge\ deg[p] \leq \|\overline{a}\| - 1 \wedge$$
$$\wedge\ ((\forall i : i \in \mathrm{N})(1 \leq i \leq \|\overline{a}\| \ \wedge i \neq j \implies Eval[p, x_i] = a_i))).$$

As we can see, the algorithm (3.264), fits to the Fibonacci-like recursive schema, and thus, we know how to generate its verification conditions. However, before going to the real verification, we first check if (3.264) with its specification (3.265), (3.266) is coherent with respect to its auxiliary functions and their specifications.

The (automatically generated) conditions for coherence are:

$$(\forall \overline{x}, \overline{a})\ (IsTuple[\overline{x}] \ \wedge\ IsTuple[\overline{a}] \ \wedge\ \|\overline{x}\| = \|\overline{a}\| \ \wedge \|\overline{a}\| \geq 1 \wedge \tag{3.267}$$
$$\wedge\ ((\forall i, j : i, j \in \mathrm{N})\ (1 \leq i, j \leq \|\overline{a}\| \ \wedge i \neq j \implies [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)) \ \wedge\ \|\overline{a}\| \leq 1$$
$$\implies$$
$$IsTuple[\overline{a}] \ \wedge \|\overline{a}\| \geq 1)$$

$$(\forall \overline{x}, \overline{a})\ (IsTuple[\overline{x}] \ \wedge\ IsTuple[\overline{a}] \ \wedge\ \|\overline{x}\| = \|\overline{a}\| \ \wedge \|\overline{a}\| \geq 1 \wedge \tag{3.268}$$
$$\wedge\ ((\forall i, j : i, j \in \mathrm{N})\ (1 \leq i, j \leq \|\overline{a}\| \ \wedge i \neq j \implies [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)) \ \wedge\ \neg(\|\overline{a}\| \leq 1)$$

$$\Longrightarrow$$
$$(IsTuple[Tail[\overline{x}]] \ \wedge \ IsTuple[Tail[\overline{a}]] \ \wedge \ \|Tail[\overline{x}]\| = \|Tail[\overline{a}]\| \ \wedge \|Tail[\overline{a}]\| \geq 1 \ \wedge$$
$$\wedge \ ((\forall i, j : i, j \in \mathrm{N}) \ (1 \leq i, j \leq \|Tail[\overline{a}]\| \ \wedge i \neq j \Longrightarrow \ [\![Tail[\overline{x}]]\!]_i \neq [\![Tail[\overline{x}]]\!]_j))$$

$$(\forall \overline{x}, \overline{a}) \ (IsTuple[\overline{x}] \ \wedge \ IsTuple[\overline{a}] \ \wedge \ \|\overline{x}\| = \|\overline{a}\| \ \wedge \|\overline{a}\| \geq 1 \ \wedge \tag{3.269}$$
$$\wedge \ ((\forall i, j : i, j \in \mathrm{N}) \ (1 \leq i, j \leq \|\overline{a}\| \ \wedge i \neq j \Longrightarrow \ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)) \ \wedge \ \neg(\|\overline{a}\| \leq 1)$$
$$\Longrightarrow$$
$$(IsTuple[Bgn[\overline{x}]] \ \wedge \ IsTuple[Bgn[\overline{a}]] \ \wedge \ \|Bgn[\overline{x}]\| = \|Bgn[\overline{a}]\| \ \wedge \|Bgn[\overline{a}]\| \geq 1 \ \wedge$$
$$\wedge \ ((\forall i, j : i, j \in \mathrm{N}) \ (1 \leq i, j \leq \|Bgn[\overline{a}]\| \ \wedge i \neq j \Longrightarrow \ [\![Bgn[\overline{x}]]\!]_i \neq [\![Bgn[\overline{x}]]\!]_j))))$$

$$(\forall \overline{x}, \overline{a}) \ (IsTuple[\overline{x}] \ \wedge \ IsTuple[\overline{a}] \ \wedge \ \|\overline{x}\| = \|\overline{a}\| \ \wedge \|\overline{a}\| \geq 1 \ \wedge \tag{3.270}$$
$$\wedge \ ((\forall i, j : i, j \in \mathrm{N}) \ (1 \leq i, j \leq \|\overline{a}\| \ \wedge i \neq j \Longrightarrow \ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)) \ \wedge \ \neg(\|\overline{a}\| \leq 1)$$
$$\Longrightarrow$$
$$(IsTuple[\overline{x}] \ \wedge \ \|\overline{x}\| \geq 1 \ \wedge \ IsTuple[\overline{a}] \ \wedge \ \|\overline{a}\| \geq 1))$$

$$(\forall \overline{x}, \overline{a}) \ (IsTuple[\overline{x}] \ \wedge \ IsTuple[\overline{a}] \ \wedge \ \|\overline{x}\| = \|\overline{a}\| \ \wedge \|\overline{a}\| \geq 1 \ \wedge \tag{3.271}$$
$$\wedge \ ((\forall i, j : i, j \in \mathrm{N}) \ (1 \leq i, j \leq \|\overline{a}\| \ \wedge i \neq j \Longrightarrow \ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)) \ \wedge \ \neg(\|\overline{a}\| \leq 1)$$
$$\Longrightarrow$$
$$(IsTuple[\overline{x}] \ \wedge \ \|\overline{x}\| \geq 1 \ \wedge \ IsTuple[\overline{a}] \ \wedge \ \|\overline{a}\| \geq 1))$$

$$(\forall \overline{x}, \overline{a}, p_1, p_2) \ (IsTuple[\overline{x}] \ \wedge \ IsTuple[\overline{a}] \ \wedge \ \|\overline{x}\| = \|\overline{a}\| \ \wedge \|\overline{a}\| \geq 1 \ \wedge \tag{3.272}$$
$$\wedge \ ((\forall i, j : i, j \in \mathrm{N}) \ (1 \leq i, j \leq \|\overline{a}\| \ \wedge i \neq j \Longrightarrow \ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)) \ \wedge \ \neg(\|\overline{a}\| \leq 1) \ \wedge$$
$$\wedge \ IsPoly[p_1] \ \wedge \ ((\forall i : i \in \mathrm{N}) \ (1 \leq i \leq \| \ Tail[\overline{x}]\| \ \Longrightarrow \ Eval[p_1, [\![Tail[\overline{x}]]\!]_i] = [\![Tail[\overline{a}]]\!]_i] \ \wedge$$
$$\wedge \ deg[p_1] \leq \| \ Tail[\overline{a}]\| - 1 \ \wedge$$
$$\wedge \ IsPoly[p_2] \ \wedge \ ((\forall i : i \in \mathrm{N}) \ (1 \leq i \leq \| \ Bgn[\overline{x}]\| \ \Longrightarrow \ Eval[p_2, [\![Bgn[\overline{x}]]\!]_i] = [\![Bgn[\overline{a}]]\!]_i] \ \wedge$$
$$\wedge \ deg[p_2] \leq \| \ Bgn[\overline{a}]\| - 1$$
$$\Longrightarrow$$
$$(Last[\overline{x}] - First[\overline{x}] \neq 0) \ \wedge \ IsTuple[\overline{x}] \ \wedge \ \|\overline{x}\| \geq 1))).$$

At the first side, the formulae look very complicated, however, they are almost trivial to prove. Let us have a closer look at them—one-by-one.

In (3.267) the outermost symbol is "$\Longrightarrow$" and, at the right-hand-side, we have to prove:

$$IsTuple[\overline{a}] \ \wedge \|\overline{a}\| \geq 1,$$

which is assumed at the left-hand-side. Thus, the formula holds.

In (3.268) the outermost symbol is "$\Longrightarrow$" and, at the right-hand-side, we have to prove:

$$(IsTuple[Tail[\overline{x}]] \ \wedge \ IsTuple[Tail[\overline{a}]] \ \wedge \ \|Tail[\overline{x}]\| = \|Tail[\overline{a}]\| \ \wedge \|Tail[\overline{a}]\| \geq 1 \ \wedge$$

$$\wedge \ (\forall i,j : i,j \in \mathtt{N}) \ (1 \leq i,j \leq \|Tail[\overline{a}]\| \ \wedge i \neq j \Longrightarrow \ [\![Tail[\overline{x}]]\!]_i \neq [\![Tail[\overline{x}]]\!]_j).$$

Under the assumption that $IsTuple[\overline{x}]$ and $IsTuple[\overline{a}]$ and $\|\overline{x}\| \geq 1$ and $\neg(\|\overline{x}\| \leq 1)$, it follows $IsTuple[Tail[\overline{x}]]$ and $IsTuple[Tail[\overline{a}]]$.

Additionally, from $\|\overline{x}\| = \|\overline{a}\|$ and $\|\overline{x}\| \geq 1$ and $\neg(\|\overline{x}\| \leq 1)$, it follows $\|Tail[\overline{x}]\| = \|Tail[\overline{a}]\|$ and $\|Tail[\overline{a}]\| \geq 1$.

Now, it remains to prove that:

$$(\forall i,j : i,j \in \mathtt{N}) \ (1 \leq i,j \leq \|Tail[\overline{a}]\| \ \wedge i \neq j \Longrightarrow \ [\![Tail[\overline{x}]]\!]_i \neq [\![Tail[\overline{x}]]\!]_j),$$

which follows from the assumption:

$$(\forall i,j : i,j \in \mathtt{N}) \ (1 \leq i,j \leq \|\overline{a}\| \ \wedge i \neq j \Longrightarrow \ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j).$$

Thus, the formula (3.268) holds.

In (3.269) the outermost symbol is "$\Longrightarrow$" and, at the right-hand-side, we have to prove:

$$(IsTuple[Bgn[\overline{x}]] \ \wedge \ IsTuple[Bgn[\overline{a}]] \ \wedge \ \|Bgn[\overline{x}]\| = \|Bgn[\overline{a}]\| \ \wedge \|Bgn[\overline{a}]\| \geq 1 \ \wedge$$

$$\wedge \ ((\forall i,j : i,j \in \mathtt{N}) \ (1 \leq i,j \leq \|Bgn[\overline{a}]\| \ \wedge i \neq j \Longrightarrow \ [\![Bgn[\overline{x}]]\!]_i \neq [\![Bgn[\overline{x}]]\!]_j))).$$

Under the assumption that $IsTuple[\overline{x}]$ and $IsTuple[\overline{a}]$ and $\|\overline{x}\| \geq 1$ and $\neg(\|\overline{x}\| \leq 1)$, it follows $IsTuple[Bgn[\overline{x}]]$ and $IsTuple[Bgn[\overline{a}]]$.

Additionally, from $\|\overline{x}\| = \|\overline{a}\|$ and $\|\overline{x}\| \geq 1$ and $\neg(\|\overline{x}\| \leq 1)$, it follows $\|Bgn[\overline{x}]\| = \|Bgn[\overline{a}]\|$ and $\|Bgn[\overline{a}]\| \geq 1$.

It remains to prove that:

$$(\forall i,j : i,j \in \mathtt{N}) \ (1 \leq i,j \leq \|Bgn[\overline{a}]\| \ \wedge i \neq j \Longrightarrow \ [\![Bgn[\overline{x}]]\!]_i \neq [\![Bgn[\overline{x}]]\!]_j),$$

which follows from the assumption:

$$(\forall i,j : i,j \in \mathtt{N}) \ (1 \leq i,j \leq \|\overline{a}\| \ \wedge i \neq j \Longrightarrow \ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j).$$

Thus, the formula (3.269) holds.

In (3.270), we have to prove:

$$(IsTuple[\overline{x}] \ \wedge \ \|\overline{x}\| \geq 1 \ \wedge \ IsTuple[\overline{a}] \ \wedge \ \|\overline{a}\| \geq 1),$$

where we have assumed:

$$(IsTuple[\overline{x}]\ \wedge\ IsTuple[\overline{a}]\ \wedge\ \|\overline{x}\| = \|\overline{a}\|\ \wedge\ \|\overline{a}\| \geq 1).$$

Thus, the formula (3.270) holds.

Condition (3.271) is the same as (3.270) and thus it holds.

In (3.272) we have to prove that:

$$(Last[\overline{x}] - First[\overline{x}] \neq 0)\ \wedge\ IsTuple[\overline{x}]\ \wedge\ \|\overline{x}\| \geq 1.$$

In the assumption list we have that:

$$IsTuple[\overline{x}]\ \wedge\ \|\overline{x}\| = \|\overline{a}\|\ \wedge\ \|\overline{a}\| \geq 1,$$

and thus the second half of the proof of (3.272) is done.

Now, from the assumptions:

$$\|\overline{x}\| = \|\overline{a}\|\ \wedge\ \|\overline{a}\| \geq 1\ \wedge\ \neg(\|\overline{a}\| \leq 1),$$

we derive that $\|\overline{x}\| > 1$. From here and from the assumption:

$$(\forall i, j : i, j \in \mathtt{N})\ (1 \leq i, j \leq \|\overline{a}\|\ \wedge i \neq j \Longrightarrow\ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)$$

we derive that $Last[\overline{x}] \neq First[\overline{x}]$ and thus:

$$Last[\overline{x}] - First[\overline{x}] \neq 0,$$

which concludes the proof of (3.272).

After proving that the algorithm is coherent, we now generate the verification conditions which would ensure the total correctness of the algorithm. First we list them, as they are automatically generated, and then we analyze all of them in more details.

The condition treating the special case, that is, the bottom of the recursion is:

$$\begin{gathered}(\forall \overline{x}, \overline{a})\ (IsTuple[\overline{x}]\ \wedge\ IsTuple[\overline{a}]\ \wedge\ \|\overline{x}\| = \|\overline{a}\|\ \wedge \|\overline{a}\| \geq 1\ \wedge \\ \wedge\ ((\forall i, j : i, j \in \mathtt{N})\ (1 \leq i, j \leq \|\overline{a}\|\ \wedge i \neq j \Longrightarrow\ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j))\ \wedge\ \|\overline{a}\| \leq 1 \\ \Longrightarrow \\ IsPoly[First[\overline{a}]]\ \wedge \\ \wedge\ ((\forall i : i \in \mathtt{N})\ (1 \leq i \leq \|\overline{a}\| \Longrightarrow Eval[First[\overline{a}], [\![\overline{x}]\!]_i] = [\![\overline{a}]\!]_i)\ \wedge \\ \wedge\ deg[First[\overline{a}]] \leq \|\overline{a}\| - 1)).\end{gathered} \tag{3.273}$$

The condition treating the general case, that is, the recursive calls is:

$$\begin{gathered}(\forall \overline{x}, \overline{a}, p_1, p_2)\ (IsTuple[\overline{x}]\ \wedge\ IsTuple[\overline{a}]\ \wedge\ \|\overline{x}\| = \|\overline{a}\|\ \wedge \|\overline{a}\| \geq 1\ \wedge \\ \wedge\ ((\forall i, j : i, j \in \mathtt{N})\ (1 \leq i, j \leq \|\overline{a}\|\ \wedge i \neq j \Longrightarrow\ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j))\ \wedge\ \neg(\|\overline{a}\| \leq 1)\ \wedge\end{gathered} \tag{3.274}$$

$$\wedge\, IsPoly[p_1] \;\wedge\; ((\forall i : i \in \mathtt{N})\; (1 \leq i \leq \|\, Tail[\overline{x}]\| \implies Eval[p_1, [\![Tail[\overline{x}]]\!]_i] = [\![Tail[\overline{a}]]\!]_i) \wedge$$
$$\wedge\, deg[p_1] \leq \|\, Tail[\overline{a}]\| - 1 \wedge$$
$$\wedge\, IsPoly[p_2] \;\wedge\; ((\forall i : i \in \mathtt{N})\; (1 \leq i \leq \|\, Bgn[\overline{x}]\| \implies Eval[p_2, [\![Bgn[\overline{x}]]\!]_i] = [\![Bgn[\overline{a}]]\!]_i] \wedge$$
$$\wedge\, deg[p_2] \leq \|\, Bgn[\overline{a}]\| - 1$$
$$\implies$$
$$IsPoly[\frac{(\mathcal{X} - First[\overline{x}])p_1 - (\mathcal{X} - Last[\overline{x}])p_2}{Last[\overline{x}] - First[\overline{x}]}] \wedge$$

$$\wedge\, (\forall i : i \in \mathtt{N})(1 \leq i \leq \|\, \overline{x}\| \implies Eval[\frac{(\mathcal{X} - First[\overline{x}])p_1 - (\mathcal{X} - Last[\overline{x}])p_2}{Last[\overline{x}] - First[\overline{x}]}, [\![x]\!]_i] = [\![a]\!]_i) \wedge$$

$$\wedge\, deg[\frac{(\mathcal{X} - First[\overline{x}])p_1 - (\mathcal{X} - Last[\overline{x}])p_2}{Last[\overline{x}] - First[\overline{x}]}] \leq \|\overline{a}\| - 1)).$$

The condition treating termination is:

$$(\forall \overline{x}, \overline{a})\; (p'[\overline{x}, \overline{a}] = \mathbb{T}), \tag{3.275}$$

where:

$$p'[\overline{x}, \overline{a}] = \quad \textbf{If } \|\overline{a}\| \leq 1 \quad \textbf{then } \mathbb{T} \quad \textbf{else } p'[Tail[\overline{x}], Tail[\overline{a}]] \;\wedge p'[Bgn[\overline{x}], Bgn[\overline{a}]]. \tag{3.276}$$

Now, let us see how these formulae may be proven.

The first verification condition (3.273) is about the special case of the algorithm, that is, the bottom of the recursion. There, the outermost symbol is "$\Longrightarrow$" and, at the right-hand-side, we have to prove:

$$IsPoly[First[\overline{a}]] \wedge$$
$$\wedge\, (\forall i : i \in \mathtt{N})\; (1 \leq i \leq \|\overline{a}\| \Longrightarrow Eval[First[\overline{a}], [\![\overline{x}]\!]_i] = [\![\overline{a}]\!]_i) \wedge$$
$$\wedge\, deg[First[\overline{a}]] \leq \|\overline{a}\| - 1.$$

From the assumptions $IsTuple[\overline{a}] \;\wedge\; \|\overline{a}\| \;\geq\; 1 \;\wedge\; \|\overline{a}\| \;\leq\; 1$ we derive that $\|\overline{a}\| = 1$ and $IsPoly[First[\overline{a}]]$.

On the other hand, since $First[\overline{a}]$ is a constant, we have that $deg[First[\overline{a}]] = 0$ and thus $deg[First[\overline{a}]] \leq \|\overline{a}\| - 1$.

Now we need to prove:

$$(\forall i : i \in \mathtt{N})\; (1 \leq i \leq \|\overline{a}\| \Longrightarrow Eval[First[\overline{a}], [\![\overline{x}]\!]_i] = [\![\overline{a}]\!]_i),$$

which reduces to

$$Eval[First[\overline{a}], [\![\overline{x}]\!]_1] = [\![\overline{a}]\!]_1,$$

because $\|\overline{a}\| = 1$. Since $First[\overline{a}]$ is a polynomial of degree 0, the evaluation of it at any point would be $First[\overline{a}]$ itself, thus

$$Eval[First[\overline{a}], \llbracket\overline{x}\rrbracket_1] = First[\overline{a}] = \llbracket\overline{a}\rrbracket_1,$$

which completes the proof of (3.273).

The second verification condition (3.274) is the most complicated one and it actually corresponds to the essence of Neville's algorithm. There, at the right-hand-side of the implication we have three different conjuncts, corresponding to the three requirements to the result, namely: p is a polynomial, the degree of p is not more than $n-1$ (where n is the length of the tuples $\overline{x}$ and $\overline{a}$), and the evaluation of the polynomial p at each point x_i form $\overline{x}$ is equal to a_i form $\overline{a}$. In order to prove (3.274) we split it into three formulae, corresponding to the three conjuncts at right-hand-side.

The first part of (3.274) is:

$$\begin{gathered}
(\forall\overline{x},\overline{a},p_1,p_2)\ (IsTuple[\overline{x}]\ \wedge\ IsTuple[\overline{a}]\ \wedge\ \|\overline{x}\| = \|\overline{a}\|\ \wedge \|\overline{a}\| \geq 1\ \wedge \\
\wedge\ ((\forall i,j : i,j \in \mathrm{N})\ (1 \leq i,j \leq \|\overline{a}\|\ \wedge i \neq j \Longrightarrow\ \llbracket\overline{x}\rrbracket_i \neq \llbracket\overline{x}\rrbracket_j))\ \wedge\ \neg(\|\overline{a}\| \leq 1)\ \wedge \\
\wedge\ IsPoly[p_1]\ \wedge\ ((\forall i : i \in \mathrm{N})\ (1 \leq i \leq \|\,Tail[\overline{x}]\|\ \Longrightarrow\ Eval[p_1, \llbracket Tail[\overline{x}]\rrbracket_i] = \llbracket Tail[\overline{a}]\rrbracket_i])\ \wedge \\
\wedge\ deg[p_1] \leq \|\,Tail[\overline{a}]\| - 1\ \wedge \\
\wedge\ IsPoly[p_2]\ \wedge\ ((\forall i : i \in \mathrm{N})\ (1 \leq i \leq \|\,Bgn[\overline{x}]\|\ \Longrightarrow\ Eval[p_2, \llbracket Bgn[\overline{x}]\rrbracket_i] = \llbracket Bgn[\overline{a}]\rrbracket_i]\ \wedge \\
\wedge\ deg[p_2] \leq \|\,Bgn[\overline{a}]\| - 1 \\
\Longrightarrow \\
IsPoly[\frac{(\mathcal{X} - First[\overline{x}])p_1 - (\mathcal{X} - Last[\overline{x}])p_2}{Last[\overline{x}] - First[\overline{x}]}].
\end{gathered} \tag{3.277}$$

Under the assumptions that p_1 and p_2 are polynomials and $First[\overline{x}]$ and $Last[\overline{x}]$ are constants, it follows that:

$$IsPoly[\frac{(\mathcal{X} - First[\overline{x}])p_1 - (\mathcal{X} - Last[\overline{x}])p_2}{Last[\overline{x}] - First[\overline{x}]}].$$

The second part of (3.274) is:

$$\begin{gathered}
(\forall\overline{x},\overline{a},p_1,p_2)\ (IsTuple[\overline{x}]\ \wedge\ IsTuple[\overline{a}]\ \wedge\ \|\overline{x}\| = \|\overline{a}\|\ \wedge \|\overline{a}\| \geq 1\ \wedge \\
\wedge\ ((\forall i,j : i,j \in \mathrm{N})\ (1 \leq i,j \leq \|\overline{a}\|\ \wedge i \neq j \Longrightarrow\ \llbracket\overline{x}\rrbracket_i \neq \llbracket\overline{x}\rrbracket_j))\ \wedge\ \neg(\|\overline{a}\| \leq 1)\ \wedge \\
\wedge\ IsPoly[p_1]\ \wedge\ ((\forall i : i \in \mathrm{N})\ (1 \leq i \leq \|\,Tail[\overline{x}]\|\ \Longrightarrow\ Eval[p_1, \llbracket Tail[\overline{x}]\rrbracket_i] = \llbracket Tail[\overline{a}]\rrbracket_i])\ \wedge \\
\wedge\ deg[p_1] \leq \|\,Tail[\overline{a}]\| - 1\ \wedge \\
\wedge\ IsPoly[p_2]\ \wedge\ ((\forall i : i \in \mathrm{N})\ (1 \leq i \leq \|\,Bgn[\overline{x}]\|\ \Longrightarrow\ Eval[p_2, \llbracket Bgn[\overline{x}]\rrbracket_i] = \llbracket Bgn[\overline{a}]\rrbracket_i))\ \wedge \\
\wedge\ deg[p_2] \leq \|\,Bgn[\overline{a}]\| - 1 \\
\Longrightarrow \\
(\forall i : i \in \mathrm{N})(1 \leq i \leq \|\,\overline{x}\|\ \Longrightarrow\ Eval[\frac{(\mathcal{X} - First[\overline{x}])p_1 - (\mathcal{X} - Last[\overline{x}])p_2}{Last[\overline{x}] - First[\overline{x}]}, \llbracket x\rrbracket_i] = \llbracket a\rrbracket_i).
\end{gathered} \tag{3.278}$$

First, let us observe that $\|\overline{x}\| = \|\overline{a}\| > 1$ because among the assumptions we have $\|\overline{x}\| = \|\overline{a}\|$ and $\|\overline{a}\| \geq 1$ and $\neg(\|\overline{a}\| \leq 1)$. In order to simplify the notation, we make some conventions, namely:

$$\|\overline{x}\| = n,$$
$$First[\overline{x}] = x_1,$$
$$Last[\overline{x}] = x_n.$$

From $n > 1$ and from

$$(\forall i, j : i, j \in \mathrm{N})\ (1 \leq i, j \leq \|\overline{a}\|\ \wedge i \neq j \Longrightarrow\ [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)$$

follows $x_1 \neq x_n$, that is, $First[\overline{x}] \neq Last[\overline{x}]$.

Now, we take arbitrary but fixed i such that $1 \leq i \leq n$ and we prove:

$$Eval[\frac{(\mathcal{X} - x_1)p_1 - (\mathcal{X} - x_n)p_2}{x_n - x_1}, x_i] = a_i.$$

We consider the following three cases:

- $i = 1$

  Now we have to show that:

  $$Eval[\frac{(\mathcal{X} - x_1)p_1 - (\mathcal{X} - x_n)p_2}{x_n - x_1}, x_1] = a_1,$$

  which by having the properties of $Eval$:

  $$Eval[\mathcal{X} - x_1, x_1] = x_1 - x_1,$$
  $$Eval[\mathcal{X} - x_n, x_1] = x_1 - x_n,$$

  we transform into:

  $$Eval[\frac{(\mathcal{X} - x_1)p_1 - (\mathcal{X} - x_n)p_2}{x_n - x_1}, x_1] =$$

  $$= \frac{Eval[(\mathcal{X} - x_1), x_1]\ Eval[p_1, x_1] - Eval[(X - x_n), x_1]\ Eval[p_2, x_1]}{Eval[(x_n - x_1), x_1]} =$$

  $$= \frac{-(x_1 - x_n)\ Eval[p_2, x_1]}{x_n - x_1} =$$

  and by $Eval[p_2, x_1] = a_1$, which follows from the assumption:

  $$((\forall i : i \in \mathrm{N})\ (1 \leq i \leq \|\ Bgn[\overline{x}]\|\ \Longrightarrow\ Eval[p_2, [\![Bgn[\overline{x}]]\!]_i] = [\![Bgn[\overline{a}]]\!]_i)),$$

  we finally obtain:

  $$= \frac{(x_1 - x_n)\ a_1}{x_n - x_1} = a_1,$$

  because $x_n \neq x_1$.

- $i = n$

  Now we have to show that:

$$Eval[\frac{(\mathcal{X}-x_1)p_1-(\mathcal{X}-x_n)p_2}{x_n-x_1}, x_n] = a_n,$$

  which we transform into:

$$Eval[\frac{(\mathcal{X}-x_1)p_1-(\mathcal{X}-x_n)p_2}{x_n-x_1}, x_n] =$$

$$= \frac{Eval[(\mathcal{X}-x_1), x_n]\ Eval[p_1, x_n] - Eval[(X-x_n), x_n]\ Eval[p_2, x_n]}{Eval[(x_n-x_1), x_n]} =$$

$$= \frac{(x_n-x_1)\ Eval[p_1, x_n] - (x_n-x_n)\ Eval[p_2, x_n]}{x_n-x_1} =$$

$$= \frac{(x_n-x_1)\ Eval[p_1, x_n]}{x_n-x_1} =$$

  and by $Eval[p_1, x_n] = a_n$, which follows from the assumption:

$$((\forall i : i \in \mathrm{N})\ (1 \le i \le \|\ Tail[\overline{x}]\| \implies Eval[p_1, [\![Tail[\overline{x}]]\!]_i] = [\![Tail[\overline{a}]]\!]_i)),$$

  we finally obtain:

$$= \frac{(x_n-x_1)\ a_n}{x_n-x_1} = a_n$$

  because $x_n \neq x_1$.

- $1 < i < n$ (this case exists only if $n \geq 3$)

  We have to show that:

$$Eval[\frac{(\mathcal{X}-x_1)p_1-(\mathcal{X}-x_n)p_2}{x_n-x_1}, x_i] = a_i,$$

  which we transform into:

$$Eval[\frac{(\mathcal{X}-x_1)p_1-(\mathcal{X}-x_n)p_2}{x_n-x_1}, x_i] =$$

$$= \frac{Eval[(\mathcal{X}-x_1), x_i]\ Eval[p_1, x_i] - Eval[(\mathcal{X}-x_n), x_i]\ Eval[p_2, x_i]}{Eval[(x_n-x_1), x_i]} =$$

$$= \frac{(x_i-x_1)\ Eval[p_1, x_i] - (x_i-x_n)\ Eval[p_2, x_i]}{x_n-x_1} =$$

  and by $Eval[p_1, x_i] = a_i$ and $Eval[p_2, x_i] = a_i$, which follows from the assumptions:

$$((\forall i : i \in \mathtt{N})\ (1 \le i \le \|\ Tail[\overline{x}]\| \implies Eval[p_1, [\![Tail[\overline{x}]]\!]_i] = [\![Tail[\overline{a}]]\!]_i))$$

and

$$((\forall i : i \in \mathtt{N})\ (1 \le i \le \|\ Bgn[\overline{x}]\| \implies Eval[p_2, [\![Bgn[\overline{x}]]\!]_i] = [\![Bgn[\overline{a}]]\!]_i)),$$

we finally obtain:

$$= \frac{(x_i - x_1)\, a_i - (x_i - x_n)\, a_i}{x_n - x_1} = \frac{(x_n - x_1)\, a_i}{x_n - x_1} = a_i,$$

because $x_n \neq x_1$.

With this we conclude the proof of (3.278).

The third part of (3.274) is:

$$\begin{gathered}(\forall \overline{x}, \overline{a}, p_1, p_2)\ (IsTuple[\overline{x}] \ \wedge\ IsTuple[\overline{a}] \ \wedge\ \|\overline{x}\| = \|\overline{a}\| \ \wedge \|\overline{a}\| \ge 1 \wedge \\ \wedge\ ((\forall i, j : i, j \in \mathtt{N})\ (1 \le i, j \le \|\overline{a}\| \ \wedge i \neq j \implies [\![\overline{x}]\!]_i \neq [\![\overline{x}]\!]_j)) \ \wedge\ \neg(\|\overline{a}\| \le 1) \wedge \\ \wedge\ IsPoly[p_1] \ \wedge\ ((\forall i : i \in \mathtt{N})\ (1 \le i \le \|\ Tail[\overline{x}]\| \implies Eval[p_1, [\![Tail[\overline{x}]]\!]_i] = [\![Tail[\overline{a}]]\!]_i) \wedge \\ \wedge\ deg[p_1] \le \|\ Tail[\overline{a}]\| - 1 \wedge \\ \wedge\ IsPoly[p_2] \ \wedge\ ((\forall i : i \in \mathtt{N})\ (1 \le i \le \|\ Bgn[\overline{x}]\| \implies Eval[p_2, [\![Bgn[\overline{x}]]\!]_i] = [\![Bgn[\overline{a}]]\!]_i)) \wedge \\ \wedge\ deg[p_2] \le \|\ Bgn[\overline{a}]\| - 1 \\ \implies \\ deg[\frac{(\mathcal{X} - First[\overline{x}])p_1 - (\mathcal{X} - Last[\overline{x}])p_2}{Last[\overline{x}] - First[\overline{x}]}] \le \|\overline{a}\| - 1)).\end{gathered} \tag{3.279}$$

In fact, we have to prove that:

$$deg[\frac{(\mathcal{X} - x_1)\, p_1 - (\mathcal{X} - x_n)\, p_2}{x_n - x_1}] \le n - 1.$$

First, we see that $deg[x_n - x_1] = 0$ because x_n and x_1 are constants.

From the assumptions we know that:

$$deg[p_1] \le \|Tail[\overline{a}]\| - 1 = n - 2$$

$$deg[p_2] \le \|Bgn[\overline{a}]\| - 1 = n - 2.$$

This implies that

$$deg[(\mathcal{X} - x_1)\, p_1] \le n - 1,$$

$$deg[(\mathcal{X} - x_2)\, p_2] \le n - 1,$$

and thus

$$deg[\frac{(\mathcal{X} - x_1)\, p_1 - (\mathcal{X} - x_n)\, p_2}{x_n - x_1}] \leq n - 1.$$

With this, we conclude the proof of (3.279) and (3.274).

The third verification condition (3.275) is the one which ensures termination of Neville's algorithm. We now prove that the simplified version p' (3.276) with the precondition (3.265) always terminates.

As we have already discussed, there is no general recipe for proving termination. For this particular case, we will use induction on the length of the initial tuples $\overline{a}$ and $\overline{x}$.

For optical reasons, we give here once more the definition of p':

$$p'[\overline{x}, \overline{a}] = \quad \textbf{If } \|\overline{a}\| \leq 1 \quad \textbf{then } \mathbb{T} \quad \textbf{else } p'[Tail[\overline{x}], Tail[\overline{a}]] \ \wedge \ p'[Bgn[\overline{x}], Bgn[\overline{a}]].$$

Let the length of the tuples $\overline{a}$ and $\overline{x}$ be n, that is, $\|\overline{a}\| = \|\overline{x}\| = n$. Using induction on n we will prove p' always terminates. Since $\overline{a}$ and $\overline{x}$ must be non-empty, the smallest length n can be 1 and thus we start with it.

- $n = 1$

  Then obviously, $\|\overline{a}\| \leq 1$ and thus $p'[\overline{a}, \overline{x}] = \mathbb{T}$.

- $n > 1$

  Assume that for all $\|\overline{a_0}\|$ and $\|\overline{x_0}\|$ with length smaller than n, that is, $\|\overline{a_0}\| = \|\overline{x_0}\| < n$, we have $p'[\overline{a_0}, \overline{x_0}] = \mathbb{T}$.

  Let the length of the tuples $\overline{a}$ and $\overline{x}$ be n, that is, $\|\overline{a}\| = \|\overline{x}\| = n$. Now we have:

  $$p'[\overline{a}, \overline{x}] = p'[Tail[\overline{x}], Tail[\overline{a}]] \ \wedge \ p'[Bgn[\overline{x}], Bgn[\overline{a}]].$$

  Since we have:

  $$\|Tail[\overline{x}]\| = \|Tail[\overline{a}]\| = n - 1$$

  and

  $$\|Bgn[\overline{x}]\| = \|Bgn[\overline{a}]\| = n - 1,$$

  we apply the induction hypothesis and obtain that

  $$p'[Tail[\overline{x}], Tail[\overline{a}]] = \mathbb{T}$$

  and

  $$p'[Bgn[\overline{x}], Bgn[\overline{a}]] = \mathbb{T}.$$

  From here, we obtain that $p'[\overline{a}, \overline{x}] = \mathbb{T}$, which completes the proof of (3.275).

As we described in the chapter about proving termination, before proving (3.275), one first checks if the simplified version p' is in a library. However, we have also the possibility to take it as an unproven conjecture, if it is believed to be correct.

Chapter 4

Conclusions and Further Work

In this thesis we present an experimental prototype environment for defining and verifying recursive functional programs.

In contrast to classical books on program verification [31], [16], [44] which expose methods for verifying correct programs, we put special emphasize on verifying incorrect programs. The user may easily interact with the system in order to correct the program definition or the specification.

We first perform a check whether the program under consideration is *coherent* with respect to its specification, that is, each function call is applied to arguments obeying the respective input specification.

The program correctness is then transformed into a set of first-order predicate logic formulae by a Verification Condition Generator (VCG)—a device, which takes the program (its source code) and the specification (precondition and postcondition) and produces several verification conditions, which themselves, do not refer to any theoretical model for program semantics or program execution, but only to the theory of the domain used in the program.

However, there is no "universal" VCG, due to the fact that proving program correctness is undecidable in general. On the other hand, in practice, proving program's correctness is possible in many particular cases and, therefore, many VCG-s have been developed for serving a big variety of situations. Our research is contributing exactly in this direction.

The kinds of programs we are dealing may be split into the following general classes:

- Recursive programs which may have multiple choice *if-then-else* with zero, one or more recursive calls on each branch – these are the most used in practice;
- Programs which are defined by mutual recursion;
- Programs with nested recursive definitions.

For coherent programs we are able to define a necessary and sufficient set of verification conditions, thus our condition generator is not only *sound*, but also *complete*. This distinctive feature of our method is very useful in practice for program debugging – as we also demonstrate by several examples.

The applicability of the research presented in this thesis goes beyond the area of Program Verification and Formal Methods. We discuss also the possibility of applying our methods in domains which seem to be a bit far from program verification.

For the purpose of serving education, in [55] we presented our experimental prototype environment. The discussion there is about improving the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

The use of formal methods for software design is motivated by the expectation that, performing appropriate mathematical reasoning can contribute to the reliability of a design. In our opinion, these methods should be used in practice, however, their acceptance by industry is not yet very broad. The author is convinced that in order to increase the practical impact of formal methods, the education of future software engineers should be improved, and parts of this thesis may serve as basis for such an improvement.

In section (3.10), in particular (3.10.8), we describe the possibility of supporting research in Numerical Analysis by tools for program verification. We give some ideas on how our framework may increase the efficiency of algorithm creation in Scientific Computing.

We address not only logicians (interested on program verification and automatic theorem proving), but also mathematicians, physicists and engineers who are inventing algorithms for solving concrete problems. On one hand, the help comes with the automatically obtained correctness proof. On the other hand, the inventor may try to prove the correctness of any conjecture, and in case of a failure obtain a counterexample, which may eventually help making a new conjecture.

The approach to program verification presented here is a result of an experimental work with the aim of practical verification of recursive programs. Although the examples presented here appear to be relatively simple, they already demonstrate the usefulness of our approach in the general case. We aim at extending these experiments to industrial-scale examples, which are in fact not more complex from the mathematical point of view. Furthermore we aim at improving the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

Furthermore, we want to approach the problem of program synthesis which may replace the nowadays standard way of programming.

One possible direction of our further work is the development of methods for proving total correctness of tail recursive programs. More precisely, methods for programs having a specific structure in which an auxiliary tail recursive function is driven by a main nonrecursive function, and only the specification of the main function is provided.

The difficulty there is that it is impossible to find automatically, in general, verification conditions for an arbitrary tail recursive function without knowing its specification. However, in many particular cases this is, nevertheless, possible.

The specification of the auxiliary function could be obtained automatically, for example by solving coupled linear recursive sequences with constant coefficients.

Furthermore, we aim at developing methods for synthesis of recursive programs for computing concrete problems by means of “cheap” operations e.g., additions, subtractions and multiplications.

The correctness of the synthesized programs should follow from the general correctness of the synthesis method. The synthesis method itself should be proven once for all, using, for example, the verification methods presented in this thesis.

We already made the first steps in that direction [39], [54], however, there is a lot to be investigated.

Throughout this thesis we declared many times our willingness to work only with coherent programs. However, in order to serve the specific research problems, we make some effort dealing with non-coherent programs. This is so, because we want to explore some properties of tail-recursive programs which would otherwise go beyond the capability of our already established techniques.

Bibliography

[1] ACL2. http://www.cs.utexas.edu/users/moore/acl2/.

[2] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1rd edition, 1974.

[3] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[4] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[5] F. Blanqui, S. Hinderer, S. Coupet-Grimal, W Delobel, and A. Kroprowski. CoLoR, a Coq Library on Rewriting and Termination. In A. Geser and H. Søndergaard, editors, *Proceedings of 8th International Workshop on Termination*, Seattle, WA, USA, August 2006.

[6] L. Blum. Computing over the reals: where Turing meets Newton. *Notices of the AMS*, 51:1024–1034, 2004.

[7] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, 1998.

[8] R.S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[9] Walter S. Brainerd and Lawrence H. Landweber. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, USA, 1974.

[10] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. (English translation Journal of Symbolic Computation 41 (2006) 475–511).

[11] B. Buchberger. Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In N.K. Bose, editor, *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232. Reidel Publishing Company, Dodrecht - Boston - Lancaster, 1985.

[12] B. Buchberger. Algorithm Supported Mathematical Theory Exploration: A Personal View and Stragegy. In B. Buchberger and John Campbell, editors, *Proceedings of AISC 2004 (7 th International Conference on Artificial Intelligence and Symbolic Computation)*, volume 3249 of

Springer Lecture Notes in Artificial Intelligence, pages 236–250. Copyright: Springer, Berlin-Heidelberg, 22-24 September 2004.

[13] B. Buchberger. Towards the Automated Synthesis of a Groebner Bases Algorithm. *RACSAM - Revista de la Real Academia de Ciencias (Review of the Spanish Royal Academy of Science), Serie A: Mathematicas*, 98(1):65–75, 2004.

[14] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, pages 470–504, 2006.

[15] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema Project: A Progress Report. In *Calculemus 2000: Integration of Symbolic Computation and Mechanized Reasoning*, 2000.

[16] B. Buchberger and F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics (in German).* Springer, 2nd edition, 1981.

[17] B. Buchberger and D. Vasaru. Theorema: The Induction Prover over Lists. Technical Report 97-20, RISC Report Series, University of Linz, Austria, June 9-10 1997.

[18] C. Byron, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In T. Ball and R.B. Jones, editors, *Computer aided verification : 18th International Conference, CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418, Seattle, WA, USA, 2006. Springer.

[19] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.

[20] A. Craciun. *Lazy Thinking Algorithm Synthesis in Gröbner Bases Theory.* PhD thesis, RISC, Johannes Kepler University Linz, Austria, April 2008.

[21] S. Danicic, C. Fox, M. Harman, R. Hierons, J. Howroyd, and M. R. Laurence. Static Program Slicing Algorithms are Minimal for Free Liberal Program Schemas. *The Computer Journal*, 48(6):737–748, 2005.

[22] S. Danicic, M. Harman, R. Hierons, J. Howroyd, and M. R. Laurence. Equivalence of Linear, Free, Liberal, Structured Program Schemas is Decidable in Polynomial Time. *Theoretical Computer Science*, 373(1-2):1–18, 2007.

[23] J. W. de Bakker and D. Scott. A Theory of Programs. In *IBM Seminar*, Vienna, Austria, 1969.

[24] R. W. Floyd. Assigning Meanings to Programs. In *Proc. Symphosia in Applied Mathematics 19*, pages 19–37, 1967.

[25] Thomas Forster. *Logic, Induction and Sets.* Cambridge University Press, 2003.

[26] Mike Gordon. From LCF to HOL: A Short History. pages 169–185, 2000.

[27] Sheila A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[28] PVS group. PVS Specification and Verification System. http://pvs.csl.sri.com, 2004.

[29] Begnaud Francis Hildebrand. *Introduction to Numerical Analysis: 2nd Edition*. Dover Publications, Inc., New York, NY, USA, 1987.

[30] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool. In J. Giesl, editor, *Rewriting Techniques and Applications, RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 175–184, Nara, Japan, 2005. Springer-Verlag, Berlin.

[31] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.

[32] HOL. http://hol.sourceforge.net/.

[33] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989.

[34] Jacek Jachymski. Fixed Point Theorems in Metric and Uniform Spaces via the Knaster-Tarski Principle. *Nonlinear Anal.*, 32(2):225–233, 1998.

[35] M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.

[36] A. W. Kirk and B. Sims. *Handbook of Metric Fixed Point Theory*. Springer-Verlag, 2001.

[37] B. Knaster. Un theoreme sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.

[38] L. Kovacs. *Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema*. PhD thesis, RISC, Johannes Kepler University Linz, Austria, October 2007. RISC Technical Report No. 07-16.

[39] L. Kovacs, N. Popov, and T. Jebelean. Verification Environment in Theorema. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(2):27–34, 2005.

[40] L. Kovacs, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In T. Margaria and B. Steffen, editors, *Proceedings ISOLA 2006*, Paphos, Cyprus, November 2006. To appear.

[41] Jeffrey C. Lagarias. The 3x+1 Problem: An Annotated Bibliography (1963-2000). 2006. Available at: http://arxiv.org/PS-cache/math/pdf/0608/0608208.pdf.

[42] Jeffrey C. Lagarias. The 3x+1 Problem: An Annotated Bibliography, II (2001-). 2006. Available at: http://eprintweb.org/S/authors/math/la/Lagarias/4.

[43] W. J. Langford, T. A. A. Broadbent, and R. L. Goodstein. Obituary: Professor Eric Harold Neville. *The Mathematical Gazette*, 48(364):131–145, 1964.

[44] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Teubner, second edition, 1987.

[45] D. C. Luckham, D. M. R. Park, and M. Paterson. On Formalised Computer Programs. *Journal of Computer and Systems Sciences*, 4(3):220–249, 1970.

[46] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Inc., 1974.

[47] Z. Manna and J. McCarthy. Properties of Programs and Partial Function Logic. *Machine Intelligence*, 5:27–37.

[48] Z. Manna and A. Pnueli. Formalization of Properties of Functional Programs. *J. ACM*, 17(3):555–569, 1970.

[49] J. Matthews, J S. Moore, S. Ray, and D. Vroon. Verification Condition Generation Via Theorem Proving. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, volume 4246 of *LNCS*, pages 362–376, Phnom Penh, Cambodia, November 2006.

[50] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[51] J. Myhill and J. Shepherdson. Effective Operations on Partial Recursive Functions. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 1:310–317, 1955.

[52] E. H. Neville. Iterative Interpolation. *Journal of the Indian Mathematical Society*, 20:87–120, 1934.

[53] Rozsa Peter. *Rekursive Funktionen in der Komputer-Theorie*. Verlag d. ungarisch. Akademie d. Wiss., Budapest, 1976.

[54] N. Popov and T. Jebelean. Using Computer Algebra Techniques for the Specification, Verification and Synthesis of Recursive Programs. *Mathematics and Computers in Simulation*, pages 1–13, 2007. to appear.

[55] N. Popov and T. Jebelean. A Prototype Environment for Verification of Recursive Programs. In Z. Istenes, editor, *Proceedings of FORMED'08*, pages 121–130, March 2008. to appear as ENTCS volume, Elsevier.

[56] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2 edition, 1992.

[57] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA, 1987.

[58] Heinrich Rolletschek. Computability Theory (Lecture Notes). Technical Report 01-27, RISC Report Series, University of Linz, Austria, December 2001.

[59] Konrad Slind. Another Look at Nested Recursion. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 498–518, London, UK, 2000. Springer-Verlag.

[60] D. R. Smith. Top-Down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, (27):43–96, 1985.

[61] I. Sosskow and A. Ditschew. *Theory of Programs*. Sofia University, 1996.

[62] Sunrise. http://www.cis.upenn.edu/˜hol/sunrise/.

[63] M. van der Voort. Introducing Well-founded Function Definitions in HOL. In *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, pages 117–132.

[64] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[65] Wikipedia. http://en.wikipedia.org/wiki/.

Index

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Juni, 2008 ....

Nikolaj Popov

Curriculum Vitae

Personal data

Name	Nikolaj Popov
Date and place of birth	July 18, 1970, Sofia, Bulgaria
Nationality	Austria
Personal Status	Married, two daughters

Affiliation

Research Institute for Symbolic Computation
Johannes Kepler Universität Linz
Altenberger Straße 69
4040 Linz
AUSTRIA

www.risc.uni-linz.ac.at
popov@risc.uni-linz.ac.at

Education

1988	Matura (high school graduation) at National High School of Mathematics and Science "Acad. L. Tchakalov", Sofia, Bulgaria
1988–1993	Studies in Mathematics and Computer Science at Faculty of Mathematics and Informatics, Sofia University, Bulgaria
1996–1998	Master Studies at Department of Mathematical Logic and Its Applications, Faculty of Mathematics and Informatics, Sofia University, Bulgaria *Diploma thesis:* "Periodic Interactions" *Thesis advisor:* Prof. Anatoly Buda
2000–2008	Doctorate studies at the Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria

Languages

Bulgarian	native
English	fluent
German	very good
Russian	very good

Career History

1999–2000	Assistant Professor at Department of Pure Mathematics, Technical University, Sofia, Bulgaria
2000:	Research Assistant at the Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria

Selected Publications

Articles in Journals

1. N. Popov and T. Jebelean. Using Computer Algebra Techniques for the Specification, Verification and Synthesis of Recursive Programs. *Mathematics and Computers in Simulation*, 2008. To appear.

2. T. Jebelean, L. Kovacs, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2008. To appear.

3. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

4. L. Kovacs, N. Popov, and T. Jebelean. Verification Environment in Theorema. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(2):27–34, 2005.

Articles in Refereed Conference Proceedings

1. N. Popov and T. Jebelean. A Prototype Environment for Verification of Recursive Programs. In Z. Istenes, editor, *Proceedings of FORMED'08*, pages 121–130, Budapest, Hungary, March 2008. to appear as ENTCS volume, Elsevier.

2. N. Popov and T. Jebelean. Verification of Functional Programs Containing Nested Recursion. In B. Buchberger, T. Ida and T. Kutsia, editors, *Proceedings of SCSS'08*, pages 163–175, Hagenberg, Austria, July 2008.

3. N. Popov and T. Jebelean. Proving Termination of Recursive Programs by Matching Against Simplified Program Versions and Construction of Specialized Libraries in Theorema. In D. Hofbauer and A. Serebrenik, editors, *Proceedings of 9-th International Workshop on Termination (WST'07)*, pages 48–52, Paris, France, June 2007.

4. L. Kovacs, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In T. Margaria and B. Steffen, editors, *Proceedings ISOLA 2006*, Paphos, Cyprus, November 2006. To appear.

5. T. Jebelean, L. Kovacs, and N. Popov. Experimental Program Verification in the Theorema System. In T. Margaria and B. Steffen, editors, *Proceedings ISOLA 2004*, pages 92–99, Paphos, Cyprus, November 2004.

6. L. Kovacs, T. Jebelean, and N. Popov. Verification of Imperative Programs in Theorema. In D. Dranidis and K. Tigka, editors, *Proceedings of the 1st South-East European Workshop on Formal Methods (SEEFM'03)*, pages 140–147, Thessaloniki, Greece, November 2003.

7. N. Popov and T. Jebelean. A Practical Approach to Proving Termination of Recursive Programs in Theorema. In M. Codish and A. Middeldorp, editors, *Proceedings of 7th International Workshop on Termination*, pages 43–46, Aachen, Germany, June 2004.

8. N. Popov and T. Jebelean. A Practical Approach to Verification of Recursive Programs in Theorema. In V. Negru and A. Popovici, editors, *Proceedings of International Workshop on Symbolic and Numeric Algorithms in Scientific Computing (SYNASC'03)*, pages 329–332, Timisoara, Romania, October 2003.

Contributed Talks at Conferences without Formal Proceedings

1. N. Popov and T. Jebelean. Functional Program Verification in Theorema. Recent Achievements and Perspectives, May 28 2008. Contributed talk at INTAS (Third Meeting of the INTAS Project), Kiev, Ukraine.

2. Nikolaj Popov. Verification of Recursive Functional Programs, December, 12 2007. Invited colloquium talk at Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary.

3. N. Popov and T. Jebelean. Proving Termination of Recursive Programs or How to Avoid Proving Termination, August 27 2007. Contributed talk at INTAS (Second Meeting of the INTAS Project), Moscow, Russia.

4. N. Popov and T. Jebelean. Automated Support for the Algorithm Validation, July 6 2007. Contributed talk at New Trends in Mathematics and Informatics, 60 years Institute of Mathematics, Bulgarian Academy of Sciences - A Keynote Talk, Sofia, Bulgaria.

5. N. Popov and T. Jebelean. Logical Aspects of Algorithm Verification, April, 14 2007. Contributed talk at SFB Statusseminar, Strobl, Austria.

6. N. Popov and T. Jebelean. Functional Program Verification in Theorema – Using Completeness for Debugging, December 09 2006. Contributed talk at INTAS (First Meeting of the INTAS Project), Timisoara, Romania.

7. N. Popov and T. Jebelean. Algebraic Methods in the Verification of Recursive Programs, April, 20 2006. Contributed talk at SFB Statusseminar, Strobl, Austria.

8. N. Popov and T. Jebelean. Using Computer Algebra Techniques for the Specification and Verification of Recursive Programs, June 28 2006. Contributed talk at ACA (Applications of Computer Algebra), Varna, Bulgaria.

9. N. Popov and T. Jebelean. Supporting Functional Program Verification in Theorema, July 7 2006. Contributed talk at Calculemus, Genova, Italy.

10. N. Popov and T. Jebelean. Verification and Synthesis of Tail Recursive Programs in Theorema, October 20 2006. Contributed talk at NWPT (Nordic Workshop on Programming Theory), Reykjavik, Iceland.

11. N. Popov and T. Jebelean. The Role of Algebraic Simplification in the Verification of Functional Programs, April 01 2005. Contributed talk at SFB Statusseminar, Strobl, Austria.

12. Nikolaj Popov. Functional Program Verification in Theorema, November 14 2005. Contributed talk at Theorema-Ultra-Omega'05 Workshop, Saarbruecken, Germany.

13. L. Kovacs, N. Popov, and T. Jebelean. A Verification Environment for Imperative and Functional Programs in the Theorema System, 18-19 November 2005. Contributed talk at 2nd South-East European Workshop on Formal Methods (SEEFM05), Ohrid, Macedonia.

14. N. Popov. Verification of Recursive Programs in Theorema, July 2004. Invited colloquium talk at Department of Mathematical Logic, Faculty of Mathematics and Computer Science, Sofia University, Bulgaria.

15. Nikolaj Popov. A Practical Approach to Verification of Recursive Programs in Theorema, February 2004. Technical report 04-06, Institute e-Austria Timisoara (www.ieat.ro). Contributed talk at Computer Aided Verification of Information Systems (CAVIS-04), Timisoara, Romania.

16. Nikolaj Popov. Verification of Simple Recursive Programs in Theorema: Completeness of the Method, September 2004. Contributed talk at 6th International Workshop on Symbolic and Numeric Algorithms in Scientific Computing (SYNASC04), Timisoara, Romania.

17. Nikolaj Popov. Verification of Functional Programs in Theorema, November 2004. Contributed talk at Fifth Symposium on Trends in Functional Programming (TFP 04), Munich, Germany.

18. T. Jebelean, L. Kovacs, and N. Popov. Verification of Imperative Programs in Theorema, 24-26 April 2003. Contributed talk at SFB Statusseminar, Strobl, Austria.

19. L. Kovacs and N. Popov. Procedural Program Verification in Theorema, 24-27 May 2003. Contributed talk at Omega-Theorema Workshop, RISC-Linz, Austria.