TÉCNICO LISBOA

# Formal verification of Ethereum smart contracts using Isabelle/HOL

**Maria Saraiva de Campos Mendes Ribeiro**

Thesis to obtain the Master of Science Degree in

**Mathematics and Applications**

Supervisors: Prof. Paulo Alexandre Carreira Mateus
Prof. Pedro Miguel dos Santos Alves Madeira Adão

**Examination Committee**

Chairperson: Prof. Maria Cristina de Sales Viana Serôdio Sernadas
Supervisor: Prof. Paulo Alexandre Carreira Mateus
Member of the Committee: Dr. Ricardo Simões do Canto de Loura

**December 2019**

To my parents, brother and ”little sister”
To my godmother, Lila
To my friends
To Inês

Thank you.

This work was partially supported by
Programa Operacional Competitividade e Internacionalização,
Programa Operacional Regional de Lisboa,
Project Bloch, POCI-01-0247-FEDER-033823.

I'm also grateful for the opportunity of being part of project Bloch and
would like to thank my supervisors and colleagues.

# Abstract

The concept of blockchain was developed with the purpose of decentralizing the trade of assets, suppressing the need for intermediaries during this process, as well as achieving a digital trust between parties. A blockchain consists in a public immutable ledger, constituted by chronologically ordered blocks such that each contains records of a finite number of transactions.

The Ethereum platform, that this thesis particularly approaches, is implemented using a blockchain architecture and introduces the possibility of storing Turing complete programs. These programs, known as smart contracts, can then be executed using the Ethereum Virtual Machine. Despite its core language being EVM bytecode, they can also be implemented using a high-level language, being Solidity the most used. Among its applications stand out decentralized information storage, tokenization of assets and digital identity verification, which can lead to revolutions in several systems, such as supply chain management.

In this thesis, a method for the formal verification of Solidity smart contracts is proposed. An imperative language, which describes a great subset of Solidity, is implemented using the Isabelle/HOL proof assistant, and associated with big-step execution semantics. Properties about programs are described using Hoare logic, a proof system is defined for the language, for which results on soundness and completeness are obtained.

The verification of an electronic voting smart contract is described, which shows the degree of proof complexity that can be achieved using this method. Examples of vulnerable smart contracts are also presented, containing overflow and reentrancy bugs.

# Keywords

formal verification, Isabelle/HOL, Hoare logic, Ethereum, blockchain, smart contracts

# Resumo

O conceito de blockchain foi desenvolvido com o objectivo de descentralizar a troca de bens, suprimindo a necessidade de intermediários durante o processo. Propõe-se a atingir uma confiança digital entre as partes envolvidas. A blockchain consiste num arquivo público e imutável, constituído por blocos ordenados cronologicamente tais que cada contém o registo de um conjunto finito de transacções.

A plataforma Ethereum, que esta tese aborda particularmente, é implementada numa arquitectura blockchain e introduz, ainda, a possibilidade de guardar programas Turing completos. Estes programas, denominados por smart contracts, podem então ser executados através da Ethereum Virtual Machine. Apesar de a sua linguagem núcleo ser EVM bytecode, também podem ser implementados usando linguagens de alto nível, sendo Solidity a mais usada. Entre as suas aplicações destacam-se o armazenamento de informação descentralizado, a tokenização de bens e a gestão de sistemas de verificação de identidades digitais. Estes podem levar a revoluções em vários sistemas, tais como a gestão de cadeias de abastecimento.

Nesta tese é proposto um método para a verificação formal de smart contracts escritos em Solidity. Uma linguagem imperativa que descreve grande parte da linguagem Solidity, é implementada no assistente de prova Isabelle/HOL, ao qual é associada a uma semântica de execução big-step. Propriedades acerca dos programas são descritas usando cálculo de Hoare, um sistema formal é definido para a linguagem, para o qual são apresentados resultados de correcção e completude.

É descrita a verificação de um smart contract de voto electrónico que demonstra a complexidade de prova que consegue ser alcançada com este método. São também apresentados exemplos de vulnerabilidade existentes como bugs de overflow ou re-entrância.

# Palavras Chave

verificação formal, Isabelle/HOL, cálculo de Hoare, Ethereum, blockchain, contractos inteligentes

# Contents

**1 Introduction** **1**

1.1 Motivation . . . 1

1.2 Literature review . . . 2

1.3 Outline . . . 3

**2 The Ethereum blockchain** **5**

2.1 Formalization of concepts . . . 5

2.2 Solidity, a high-level language for smart contracts . . . 7

**3 The SOLI language** **9**

3.1 Syntax . . . 9

3.2 Semantics . . . 10

3.3 The state space . . . 12

3.4 Environment variables . . . 13

3.5 Concrete syntax . . . 14

3.6 Additional language features . . . 15

3.6.1 Exceptions . . . 15

3.6.2 Calling a function . . . 16

3.6.2.A Internal call . . . 17

3.6.2.B External call . . . 18

3.6.3 Reverting state changes . . . 19

**4 Hoare Logic** **21**

4.1 The proof system . . . 22

4.1.1 The Consequence rule . . . 23

4.1.2 Recursive procedures . . . 25

4.2 Weakest precondition calculus . . . 26

4.3 Soundness and completeness . . . 27

4.3.1 Soundness . . . 27

4.3.2 Completeness . . . 32

4.4 Computation of verification conditions . . . 35

4.4.1 An alternative formulation of rules . . . 37

4.4.2 Factorial as an example . . . 39

4.5 Rules for derived statements . . . 40

4.5.1 Calling a function . . . 41

**5 Application to real-world smart contracts** **45**

5.1 Electronic voting . . . 45

5.2 Ethereum tokens . . . 49

5.3 Reentrancy . . . 51

**6 Conclusions** **55**

6.1 Contributions . . . 55

6.2 Future Work . . . 55

**Bibliography** **56**

**A Isabelle theories** **59**

A.1 Command syntax . . . 59

A.2 Big-step semantics . . . 60

A.3 Hoare logic . . . 61

A.4 Ballot . . . 65

# List of Figures

3.1 Big-step execution rules for SOLI . . . . . 11

3.2 Syntax translations . . . . . 14

3.3 Factorial program in SOLI - imperative version . . . . . 15

3.4 Execution rules for Require . . . . . 16

3.5 Control flow for *call* . . . . . 17

3.6 Execution rule for *call* . . . . . 17

3.7 Control flow for *init* . . . . . 19

3.8 Execution rules for *init* . . . . . 20

4.1 Hoare logic for SOLI . . . . . 23

4.2 Weakest precondition style rules for Skip, Revert and Upd . . . . . 38

4.3 Weakest precondition style rule for If . . . . . 38

4.4 Weakest precondition style rule for While . . . . . 39

4.5 Control flow for *call* with assertions . . . . . 41

4.6 Factorial program in SOLI - recursive version . . . . . 42

5.1 $winningProposal$ and $winnerName$ functions . . . . . 46

5.2 $winningProposal$ updated function . . . . . 47

5.3 $transfer$ function from the $Hexagon$ contract . . . . . 49

5.4 $withdraw$ function . . . . . 51

5.5 *call_value* definition . . . . . 52

5.6 Malicious $fallback$ function . . . . . 52

*This is Disneyland for hackers.*

— Martin Swende (Ethereum Foundation) on Ethereum security

# 1

# Introduction

## 1.1 Motivation

The concept of blockchain was developed with the purpose of decentralizing the trade of assets, suppressing the need for intermediaries during the process, as well as achieving digital trust between parties, with the use of cryptography. The emergence of this concept was associated with the appearance of Bitcoin, one of the first decentralized cryptocurrencies, introduced in 2008 by Satoshi Nakamoto [1]. A cryptocurrency is independent of any central administrative entities but instead, uses a *peer-to-peer* digital system, managed by a network of nodes. Bitcoin transactions are stored in a blockchain, an append-only public ledger, through the process of *mining*. Nodes in the network, the *miners*, try to solve a difficult - costly and time consuming - computational problem which consists in finding an adequate value for which the transaction is valid. These computations are called *proof of work* and involve hash functions so, once the value is found, it can be easily verified and the *miner* gets rewarded. When a transaction is verified by the network it is timestamped and incorporated to the blockchain using a signature obtained by a cryptographic hash function. This hash includes data from the previous block's hash which makes the whole chain cryptografically secure and, therefore, immutable. If one tries to delete or change data after it has been validated, the subsequent blocks would reject that modification as their hashes would no longer be valid.

This work is focused on the Ethereum platform, proposed by Vitalik Buterin [2] in 2013, which similarly uses a blockchain architecture but also introduces the feature of storing Turing complete programs, known as smart contracts. These programs can then be executed by the stack-based Ethereum Virtual Machine (EVM) using EVM bytecode. The formalization of EVM was first approached by Gavin Wood [3]. Developers can write smart contracts in higher level languages, being Solidity the most popular. It follows an object oriented structure and is highly influenced by C++, Python and JavaScript.

Ethereum also introduces the concept of *gas*, each operation in the virtual machine has an associated cost in *ether*, the Ethereum currency, and when a contract is executed, either by being called by a transaction or code in another contract, the original transaction initiator needs to pay for the total cost of operations.

Ethereum enables the creation of decentralized applications. Its applications, mainly based on the trade of digital assets, can go from substituting middlemen in areas such as banking and finances, the tokenization of real-world assets, facilitating their transference or transaction, management and verification of digital identities, which associated with record storing can be applied to medical records, to a revolution of the traditional supply chain.

In a potential future where decentralization becomes a costume, there's still a long path to follow regarding Ethereum security. Specially since it deals with valuable assets and because of its immutability. For instance in 2016, the DAO contract, with the goal of implementing a Decentralized Autonomous Organization, was exploited due to a bug in its code. The attack was reentrancy based and resulted in the loss of approximately $50 million in *ether*. Another common Solidity bugs are overflow and underflow, related to the fact that EVM works with 256-bit unsigned integers.

That being said, the main goal of this work is introduced: the formal verification of Ethereum smart contracts using Isabelle, a higher order logic (HOL) theorem prover. The primary idea was to verify contracts written in EVM bytecode by defining the EVM using this prover. However, it was realized that there's was already some work being developed in the subject [4]. Nevertheless, the option of working with smart contracts written in Solidity, a language prone to some vulnerability, was followed. The construction of a simple imperative language was developed in Isabelle until a relevant subset of Solidity was reached. To formalize the meaning of the language, in terms of execution, operational big-step semantics are defined. The verification of programs defined in this language is approached using Hoare logic, a proof system is defined with that intent. A program's specification is stated through a set of preconditions and postconditions, that should be verified before and after execution and correct according to rules defined for each command. The notion of a valid Hoare formula is formally defined and having so, soundness and (relative) completeness results are obtained for the proof system. The concept of weakest precondition [5] is used in the completeness result and in the description of an optimized method for applying the Hoare rules. Finally, relevant examples of application, which describe the expressiveness of the language, are described and analyzed, namely regarding electronic voting, tokens and reentrancy.

## 1.2 Literature review

The introductory article [2] from Ethereum's founder, Vitalik Buterin, gives an initial intuition about Ethereum and its possible applications. The formalization of these ideas are described in Ethereum's yellow paper, by Gavin Wood [3]. The work by our colleague Beatriz [6] was also important to understand Ethereum's main concepts.

Having in mind the formal verification of Ethereum in Isabelle/HOL some introductory works [7] were followed as tutorials. There are already some implementations of the EVM, namely one in Lem by Yochi Hirai [4], which can be translated to several theorem provers such as Coq and Isabelle/HOL. This work was analyzed and contributed to the change of subject from EVM bytecode to a higher level language, Solidity. The book concrete semantics [8] includes an introduction to Isabelle/HOL, but also an introduction to semantics and its applications, based on a simple imperative language, `IMP`. It covers topics such as operational semantics, compiler correctness, and Hoare logic. This initial language was developed into `SOLI`. The polymorphic way `SOLI` statements were defined was based on a language described in a compilation of Isabelle/HOL examples [9] and the language for concurrent programs by Nipkow and Nieto [10]. The work by Schirmer [11] for sequential programs was a big influence in this work, either for the formalization of concepts such as big-step and axiomatic semantics, as well as modeling some features like calling functions and handling exceptions.

Regarding Hoare logic, the original papers by Hoare [12] and [13] were followed to develop the proof system in Isabelle/HOL. The question regarding the use auxiliary variables in specifications was analyzed, Kleymann [14] introduces a Consequence rule for that matter. Weakest precondition calculus was a concept introduced by Dijkstra [5] and the computation of weakest preconditions and verification conditions are based, again, on concrete semantics [8] and the work by Frade and Pinto [15].

Regarding the soundness result, the proof was based on the proof by Schirmer [11], including the concepts of execution with limit and validity with limit and context. The completeness result, and in this case the correct term is relative completeness, a concept introduced by Cook [16], is based on the proof by Winskel [17], which uses weakest preconditions, with the addition of new cases.

The main literature for all the aspects of Solidity were the language's documentation available online [18]. The realization of the immense possibilities for Solidity hacking was due to the Ethernaut game [19].

## 1.3 Outline

The present chapter, introduces a background to this thesis, motivations, as well as a literature review.

Chapter 2 introduces the formalization of concepts behind the Ethereum blockchain: the world state, accounts, transactions and the blockchain itself. Also a proper introduction to Solidity is approached.

Chapter 3 introduces the `SOLI` language. It covers its syntax, big-step semantics and some implementation details regarding the state space - storage vs. local variables - and the execution environment. Additional language features, created using the initial set of commands, define how to throw and handle exceptions - in Solidity all state changes are reverted - and how to call a procedure with argument passing, returning from the call and returning results. A syntax is defined, as well as execution rules.

Chapter 4 introduces Hoare logic and its formalization for `SOLI` regarding partial correctness. There's a detailed explanation for the Consequence rule and how to deal with recursive procedures. Results for soundness and (relative) completeness are obtained. The definition and computation of weakest precondition for `SOLI` is presented, as well as the computation of verification conditions based on this concept. An alternative formulation of rules following a weakest precondition style and a method for their application is formulated. Finally, Hoare rules for additional language features defined in the previous chapter are introduced. Some examples follow along the way such as the verification of the factorial function for both the imperative and recursive cases.

In Chapter 5 some examples of program verification are analyzed: an electronic voting contract and two contracts known to be vulnerable, a Token with an overflow bug and a simplified version of DAO.

Chapter 6 wraps this thesis with an overview of the achieved results and some considerations about future work.

Appendix A presents the main Isabelle/HOL theories developed during this thesis.

# 2

# The Ethereum blockchain

As described in the introduction of this thesis, Ethereum can be seen as a decentralized computing platform, it uses a blockchain architecture and introduces the feature of storing Turing complete programs, known as *smart contracts*. This chapter introduces the formal definition of some Ethereum concepts and presents the structure of a Solidity contract.

## 2.1 Formalization of concepts

In the following definitions let $\mathbb{N}_x$ the set of non-negative integers with size up to $x$ bits. An account is an object part of the Ethereum environment that is identified by a 160-bit string known as the account's address.

**Definition 1** (World state)**.** *The world state is a mapping $\sigma$ between addresses (160 bit strings) and account states.*

$$\sigma : \{0,1\}^{160} \rightarrow \mathbb{N}_{256} \times \mathbb{N}_{256} \times \{0,1\}^{256} \times \{0,1\}^{256}$$

There are two types of accounts: externally owned accounts (EOA) and accounts associated with code (contract accounts). The first are owned by an external user and have no associated code, while the latter contain code, and its respective storage, associated. Blockchain uses asymmetric cryptography to manage accounts. Each EOA has its own public and a private keys. The account's address is derived from its public key, the owner has the private key which allows to control the account. The currency used in Ethereum is called ether.

**Definition 2** (Account state)**.** *Given an address a, the account state $\sigma(a)$ is a tuple $\mathcal{A} = \langle$nonce, balance, storagehash, codehash$\rangle$, where*

- ***nonce*** $\in \mathbb{N}_{256}$ *is equal to, if a is the address of an EOA, the number of transactions sent from this address or, if a is the address of a contract account, the number of contract-creations made by this account;*
- ***balance*** $\in \mathbb{N}_{256}$ *is the value of ether owned by a;*
- ***storagehash*** *is the 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account;*

- ***codehash*** *is the Keccak-256 hash of the EVM code of this account, in case of an EOA corresponds to the hash of the empty string.*

There are two types of transactions, contract creation transactions and transactions which result in message calls. A transaction is triggered by an external actor.

**Definition 3** (Transaction)**.** *A transaction is a tuple $\mathcal{T}$= ⟨nonce, gasPrice, gasLimit, to, value, v, r, s, init/data⟩, where*

- ***nonce*** $\in \mathbb{N}_{256}$ *is the number of transactions sent by the sender of the transaction;*
- ***gasprice*** $\in \mathbb{N}_{256}$ *is equal to the cost per unit of gas, in ether, for all computation costs of this transaction;*
- ***gaslimit*** $\in \mathbb{N}_{256}$ *is equal to the maximum amount of gas that should be used in the execution of this transaction;*
- ***to*** $\in \{0,1\}^{256}$ *is the address of the message call's recipient or, for a contract creation transaction,* $\emptyset$*;*
- ***value*** $\in \mathbb{N}_{256}$ *is the value of ether to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account.*
- ***v, r, s*** *are values corresponding to the signature of the transaction, which are used to determine the sender of the transaction;*

*Additionally, in the case of a contract creation transaction*

- ***init*** *is the EVM code for the account initialisation procedure;*

*Or, in the case of a message call*

- ***data*** *is the input data of the message call.*

A message call is an internal concept which consists in data (a set of bytes) and value (specified as ether) passed from an account to another. It may be triggered by a transaction, where the sender is an EOA, or by EVM code, where the sender is a contract account.

Transactions are grouped and stored in finite blocks.

**Definition 4** (Block)**.** *A block B is a package of data constituted by*

- *the* ***header*** *which is a collection of relevant pieces of information such as its number, difficulty level, gas limit and usage, timestamp, logs bloom filter and nonce. It also contains hashes of the parent block's header, the ommers list and the roots of the following structures: state, transaction list and transaction receipts;*

- *the **transaction list**;*
- *a set of block headers which have the same parent as the current block's parent's parent. Such blocks are called ommers.*

Ethereum can be seen as a transaction-based state machine. In such a representation a transaction represents a valid transition between two states $\sigma_t$ and $\sigma_{t+1}$. Since transactions are grouped in finite blocks, a block may also represent a state transition $\sigma'_t$ and $\sigma'_{t+1}$. These transitions between blocks introduce the concept of a chain of blocks, a blockchain.

**Definition 5** (Blockchain)**.** *A blockchain is defined as an ordered sequence of blocks*

$$\mathcal{B} = \{B_0, B_1, ...\}.$$

## 2.2 Solidity, a high-level language for smart contracts

In this thesis, an approach to the formal verification of Solidity smart contracts is presented. Solidity is an object-oriented language highly influenced by C++, Python and JavaScript. It is the most famous high level language for smart contract implementation, that is, which compiles to EVM bytecode. The EVM consists in a stack machine which operates with 256-bit elements. The operations are associated with a volatile memory and can access the contract account's storage. Each operation has a cost associated in ether, known as *gas*.

As formalized in the previous section, a contract account is associated with storage and code and resides at a specific address on the Ethereum blockchain.

Regarding code structure, a Solidity contract declaration consists, as it follows a object-oriented structure, in a set of state variables, which are part of the account's storage, and a set of functions declarations. Functions in a contract can introduce local variables, which are stored in the EVM memory. Solidity also contains a set of globally available variables which can be accessed regarding the current block, transaction, message call and address. Also some mathematical and cryptographic functions are defined such as the Keccak-256 and SHA-256 hash functions.

A function can be called by an external user, an EOA, which initiates a transaction, or by another contract. This happens when a called contract contains code which call another contract, generating a new message call. Note that a message call is an internal concept, which differs from a transaction but can be triggered by it.

A function call can be internal, when the called function belongs to the same contract, or external, when the function is part of another contract. An external call can be a regular call, or a delegate call, in which code is executed in the context of the calling contract. Details about these methods and respective implementation are presented in 3.6.2.

Every contract has a fallback function, a function with no arguments or return values, which is automatically executed whenever a call is made to the contract and none of its other functions match the given function identifier or no data is supplied. This is the case when the contract receives ether, with no data specified. The fallback function is distinguished by the fact of having no name associated.

In Solidity there are three methods to send ether: `send()`, `transfer()` and `call.value(amount)()`. The first two have a 2300 *gas* limit for code execution, due to the triggering of the receiver contract's fallback function. `call.value` allows to set a *gas* limit, but if no value is defined there is no limit for the execution. `send()` returns false on failure, while `transfer()` and `call.value()` return an exception.

When an exception is thrown, all states changes are reverted. The approach for modelling exceptions and state reversion is described in sections 3.6.1 and 3.6.3.

# 3

# The SOLI language

This chapter is focused on the construction of an imperative language on Isabelle/HOL, which aims to be a subset of Solidity, capturing its more important concepts in order to succeed on proving properties about smart contracts.

The core elements of the language are defined and a set of big-step executions rules is introduced to describe their semantics. Some more complex language structures, such as calling function and throwing/handling exceptions, are constructed using the core commands.

## 3.1 Syntax

In order to formalize a subset of Solidity, basic imperative constructs are defined, such as assignments, sequential composition, conditional statements and while loops, along with auxiliary commands to call procedures and to throw and handle exceptions. This set captures the expressiveness of the most relevant Solidity control structures.

The syntax of the language is a combination of deep and shallow embeddings. Commands are represented by an inductive datatype whereas some other syntactic elements are represented as abbreviations of its semantics, such as assertions, expressions and assignments. Boolean expressions, *bexp*, and assertions, *assn*, are defined as state sets. For instance, let $s$ be a state and $b$ a boolean expression. If $s$ belongs do $b$, it means the expression evaluates as true in that state, otherwise, if $s$ doesn't belong to $b$, it means the expression evaluates as false in that state.

The state space representation, which models the values of the program variables, is independent of this definition and approached in 3.3. `SOLI` commands are built upon a dependence on the state space.

**Definition 6** (Syntax)**.** *Let ´s be the state space type. The syntax for boolean expressions and assertions is defined by the types* `'s bexp` *and* `'s assn`*, respectively. The syntax for commands is defined by the polymorphic datatype* `´s com`*, where ´s ⇒ ´s is a state-update function and* $fname$ *the type of function names.*

$$\begin{array}{lcl}
'\textit{s bexp} & := & '\textit{s set} \\
'\textit{s assn} & := & '\textit{s set} \\
'\textit{s com} & := & \textbf{Skip} \mid \textbf{Upd}\ '\textit{s} \Rightarrow '\textit{s} \mid \textbf{Seq}\ '\textit{s com}\ '\textit{s com} \mid \textbf{If}\ '\textit{s bexp}\ '\textit{s com}\ '\textit{s com} \\
& & \mid \textbf{While}\ '\textit{s bexp}\ '\textit{s com} \mid \textbf{Dyncom}\ '\textit{s} \Rightarrow '\textit{s com} \mid \textbf{Call}\ \textit{fname} \\
& & \mid \textbf{Revert} \mid \textbf{Handle}\ '\textit{s com}\ '\textit{s com}
\end{array}$$

**Upd** is used to model assignments by executing a state-update function $'s \Rightarrow' s$. Conditional statements and while loops are defined with the usual syntax. The **Skip** command, which does nothing, is also defined and can be used, for instance, to express a conditional with no else branch.

In order to add some complexity to the language, such as calling other functions and reverting all state changes, the following commands are introduced in `SOLI`. **Dyncom** allows to write statements which are state dependent, this is useful when referring to states in different steps of execution. A general **Call** is introduced here, which receives a function name. It corresponds to the simplest form of calling a procedure, where there are no parameter passing, returning results or variables to reset. The different types of procedure calls and respective execution details are described in 3.6.2. **Revert** throws a revert type exception which signals that the state must be reverted. **Handle** is an auxiliary command used to handle state reversion if signaled.

## 3.2 Semantics

The most common approaches to formalize the meaning of a program are denotational semantics, operational semantics and axiomatic semantics. Denotational semantics describe the meaning of a program by attaching mathematical meaning to the expressions of the language. The axiomatic approach consists on a set of logical rules that describe the effect of a program on assertions about its state. The most common example is Hoare Logic which will be formalized for `SOLI` in chapter 4.

The aim of operational semantics is to to capture the meaning of a program as a relation that describes how the program executes directly, that is, by describing how a sequence of computational steps should be executed from state to state. The execution can be described individually - small-step semantics - or as a whole result - big-step semantics. In this section, the big-step semantics are defined.

To model the operational semantics, the state space $'s$ is augmented, as described by the datatype $'s\ state$, with information about whether or not any exceptions where thrown.

$$'s\ state \quad := \quad \text{Normal}\ \ 's \mid \text{Rev}\ \ 's$$

The execution starts in a normal state and changes to a revert state if a **Revert** command occurs. This flags that all state changes should be or were reverted.

To formalize the execution relation, the function body environment $\Gamma$ is introduced, which maps function names to the corresponding bodies.

**Definition 7** (Big-step semantics)**.** *The big-step semantics for* `SOLI` *are based on a deterministic evaluation relation formalized by predicate*

$$\Gamma \vdash \langle c,\ s\rangle \Rightarrow t$$

*where*

$$\begin{aligned} \Gamma &:: \mathit{fname} \rightharpoonup {}'s\ \mathit{com} \\ c &:: {}'s\ \mathit{com} \\ s,\ t &:: {}'s\ \mathit{state} \end{aligned}$$

*and evaluated accordingly to the set of rules represented in Figure 3.1.*

The meaning for this relation is that if command $c$ is executed in state $s$ then the execution terminates in state $t$.

$$\frac{}{\Gamma \vdash \langle \mathbf{Skip},\ \text{Normal } s\rangle \Rightarrow \text{Normal } s}\ (\mathit{Skip}) \qquad \frac{}{\Gamma \vdash \langle \mathbf{Upd}\ f,\ \text{Normal } s\rangle \Rightarrow \text{Normal } (f\ s)}\ (\mathit{Upd})$$

$$\frac{\Gamma \vdash \langle c_1,\ \text{Normal } s_1\rangle \Rightarrow s_2 \qquad \Gamma \vdash \langle c_2,\ s_2\rangle \Rightarrow s_3}{\Gamma \vdash \langle \mathbf{Seq}\ c_1\ c_2,\ \text{Normal } s_1\rangle \Rightarrow s_3}\ (\mathit{Seq})$$

$$\frac{s \in b \qquad \Gamma \vdash \langle c1, \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \mathbf{If}\ b\ c1\ c2,\ \text{Normal } s\rangle \Rightarrow t}\ (\mathit{IfTrue}) \qquad \frac{s \notin b \qquad \Gamma \vdash \langle c_2,\ \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \mathbf{If}\ b\ c_1\ c_2,\ \text{Normal } s\rangle \Rightarrow t}\ (\mathit{IfFalse})$$

$$\frac{s_1 \in b \qquad \Gamma \vdash \langle c,\ s_1\rangle \Rightarrow s_2 \qquad \Gamma \vdash \langle \mathbf{While}\ b\ c,\ s_2\rangle \Rightarrow s_3}{\Gamma \vdash \langle \mathbf{While}\ b\ c,\ \text{Normal } s_1\rangle \Rightarrow s_3}\ (\mathit{WhileTrue})$$

$$\frac{s \notin b}{\Gamma \vdash \langle \mathbf{While}\ b\ c,\ \text{Normal } s\rangle \Rightarrow \text{Normal } s}\ (\mathit{WhileFalse})$$

$$\frac{\Gamma \vdash \langle c\ s,\ \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \mathbf{DynCom}\ c,\ \text{Normal } s\rangle \Rightarrow t}\ (\mathit{DynCom}) \qquad \frac{\Gamma \vdash \langle \mathit{the}(\Gamma\ f),\ \text{Normal } s\rangle \Rightarrow t}{\Gamma \vdash \langle \mathbf{Call}\ f,\ \text{Normal } s\rangle \Rightarrow t}\ (\mathit{Call})$$

$$\frac{\Gamma \vdash \langle c_1, \text{Normal } s\rangle \Rightarrow \text{Normal } t}{\Gamma \vdash \langle \mathbf{Handle}\ c_1\ c_2, \text{Normal } s\rangle \Rightarrow \text{Normal } t}\ (\mathit{HandleNormal})$$

$$\frac{\Gamma \vdash \langle c_1, \text{Normal } s\rangle \Rightarrow \text{Rev } r \qquad \Gamma \vdash \langle c_2,\ \text{Normal } r\rangle \Rightarrow t}{\Gamma \vdash \langle \mathbf{Handle}\ c_1\ c_2,\ \text{Normal } s\rangle \Rightarrow t}\ (\mathit{HandleRevert})$$

$$\frac{}{\Gamma \vdash \langle \mathbf{Revert},\ \text{Normal } s\rangle \Rightarrow \text{Rev } s}\ (\mathit{Revert}) \qquad \frac{}{\Gamma \vdash \langle c,\ \text{Rev } s\rangle \Rightarrow \text{Rev } s}\ (\mathit{RevState})$$

**Figure 3.1:** Big-step execution rules for `SOLI`

Most of these rules start in a normal state and either stay in the same type of state or end in an unknown state type due to the inductive nature of this definition. The only rules that explicitly model states of the revert type are *Revert*, *HandleRevert* and *RevState*. The detailed description of the rules follows.

*Skip* is an axiom which states that the command **Skip** makes no alterations to the state. The *Upd* axiom states that if the command is an assignment, **Upd** $f$, the final state is the application of the state update function $f$ to the initial state. If the command is **Seq** $c_1$ $c_2$, the final state is the result of executing $c_2$ from the resulting state of executing $c_1$ from the initial state.

The guards for conditional statements and loops are evaluated semantically as state sets to which the current state belongs (or not). If the command is a conditional statement, **If** $b$ $c_1$ $c_2$, and $b$ is true, the final state is the result of executing $c_1$ from the initial state. If $b$ is false, the final state is the result of executing $c_2$ from the initial state. If the command is **While** $b$ $c$ and $b$ is false, the state remains unaltered. If $b$ is true, the body of the loop $c$ is executed from the initial state, resulting in a intermediate state. The final state appears by recursively applying the *WhileTrue* or *WhileFalse* rule, depending on the case. If the command is **DynCom** $c$, where $c$ is a command which receives a state, the final state is the execution of $cs$ from $s$, where $s$ is the initial state. If the command is **Call** $f$, the final state is the same as executing $\Gamma f$ from the initial state.

If the command is **Handle** $c_1$ $c_2$, the result depends on the execution of $c_1$. If executing $c_1$ terminates in a normal state, the final state is that state, there is no need to handle any exceptions thrown. If the execution terminates in a revert state, the final state is the result of executing $c_2$, the command which handles the exception, from that same state but as normal, otherwise some rules such as the assignment would not have the expected effect. If the command is **Revert**, the final state corresponds to signaling the initial state, that is, the modification from normal state to revert type state. *RevState* is an axiom which allows the propagation of the revert state.

## 3.3 The state space

The state space is usually represented by a mapping from variable names to their values. However this approach induces some typing issues in HOL, a datatype would have to be created to represent the different types of values. To simplify this problem and since the goal of this thesis is to verify properties about specific programs, it was chosen to explicitly state the HOL type for each variable by working with records.

Some of the used typed are constructed from the HOL types. For instance, Solidity works with 256-bit unsigned integers and an account address is represented by a 160-bit value. Here follow the construction of some types from the type word, which represents a bit.

$$\begin{aligned} byte &:= 8\ word \\ address &:= 160\ word \\ uint &:= 256\ word \end{aligned} \tag{3.1}$$

A record of Isabelle/HOL is a collection of fields where each has a specified name and type. A record comes with select and update operations for each field. Record types can also be defined by extending other record types.

Consider the record *st* which represents the storage of a Solidity contract and *loc* which is defined by extending *st* and represents the local variables of the functions in that contract.

$$\begin{aligned} record\ st &:= \\ owner &:: address \\ \\ record\ loc &:= st + \\ a &:: uint \\ b &:: uint \\ res &:: uint \end{aligned} \tag{3.2}$$

Consider the field $a$ from the *loc* record, its selector and update functions are $a\ :: \ loc\ \Rightarrow\ uint$ and $a - update\ ::\ uint\ \Rightarrow\ loc\ \Rightarrow\ uint$. $a - update\ n\ loc$ is expressed with the syntax $loc(\!|a\ :=\ n|\!)$.

The operations on records only modify its relative field, this is useful since there is no need to define frame conditions.

## 3.4 Environment variables

Solidity defines a set of global variables regarding the execution environment, mainly to provide information about the blockchain. With respect to the block, variables regarding the current block's hash, miner's address, difficulty, gaslimit, number and timestamp, are defined. For the transaction, there are variables regarding the current message call: data, gas, sender and signature, which corresponds to the first 4 bytes of the call data, and for the whole transaction: gasprice and origin. In `SOLI` these variables are part of the environment record *env*.

Also, an account state is defined as record *Account* with four fields corresponding to its nonce, balance, storage and code. The world state is defined as a field of the *env* record *gs* which maps addresses to their account states. For instance, to access the balance of the account with address $a$ one has to state *balance gs a*.

Finally, variables regarding the current address and the current timestamp are added to the *env* record.

$$
\begin{aligned}
\textit{record env} &:= \\
\textit{block\_coinbase} &:: \textit{uint} \\
\textit{block\_difficulty} &:: \textit{address} \\
\textit{block\_gaslimit} &:: \textit{uint} \\
\textit{block\_number} &:: \textit{uint} \\
\textit{block\_timestamp} &:: \textit{uint} \\
\textit{msg\_data} &:: \textit{byte list} \\
\textit{msg\_sender} &:: \textit{address} \\
\textit{msg\_value} &:: \textit{uint} \\
\textit{tx\_origin} &:: \textit{address} \\
\textit{tx\_gasPrice} &:: \textit{uint} \\
\textit{address\_this} &:: \textit{address} \\
\textit{gs} &:: \textit{address} \Rightarrow \textit{Account}
\end{aligned}
\tag{3.3}
$$

Upon the creation of a contract, the storage record $st$ is defined as an extension of this record.

$$
\begin{aligned}
\textit{record st} &:= \textit{env} + \\
\textit{owner} &:: \textit{address} \\
\\
\textit{record loc} &:= \textit{st} + \\
a &:: \textit{uint} \\
b &:: \textit{uint} \\
\textit{res} &:: \textit{uint}
\end{aligned}
\tag{3.4}
$$

## 3.5 Concrete syntax

To improve readability of `SOLI` programs, some syntax translations are introduced. To simplify stating boolean expressions and assertions, the translation $\{\!|b|\!\}$ is used. $\{\!|b|\!\}$ is defined as the set of states for which the predicate $b$ holds.

To refer to a variable $x$ without being explicit about the record where it belongs, $\acute{x}$ is used. Let $c_1$ and $c_2$ be commands, $b$ a boolean and $s$ a state.

$$
\begin{aligned}
\acute{x} \ ::= \ a &\rightharpoonup \textbf{Upd } (\lambda s.\ s(\!|x := a|\!)) \\
c_1;;c_2 &\rightharpoonup \textbf{Seq } c_1\ c_2 \\
\textbf{IF } b \textbf{ THEN } c_1 \textbf{ ELSE } c_2 &\rightharpoonup \textbf{If } \{\!|b|\!\}\ c_1\ c_2 \\
\textbf{IF } b \textbf{ THEN } c_1 &\rightharpoonup \textbf{IF } b \textbf{ THEN } c_1 \textbf{ ELSE Skip} \\
\textbf{WHILE } b \textbf{ DO } c &\rightharpoonup \textbf{While } \{\!|b|\!\}\ c \\
\textbf{REQUIRE } b &\rightharpoonup \textbf{Require } \{\!|b|\!\}
\end{aligned}
$$

**Figure 3.2:** Syntax translations

The **Require** command is introduced in section 3.6.1.

As an example, consider the imperative definition of factorial written in `SOLI`. Suppose this function is part of a contract for which the state space is the one described in 3.2. The local variables for this specific function are represented in record *loc*. Function parameters are modelled according to the state space definition.

$$\begin{aligned}
fac\_imp &:: \ st\ com\ where \\
fac\_imp &\equiv \acute{b} \ ::= \ 1;; \\
&\quad (\textbf{WHILE}\ (\acute{a} > 0)\ \textbf{DO} \\
&\qquad \acute{b} \ ::= \ \acute{b} \ \times \ \acute{a};; \\
&\qquad \acute{a} \ ::= \ \acute{a} \ - \ 1);; \\
&\quad \acute{res} \ ::= \ \acute{b}
\end{aligned}$$

**Figure 3.3:** Factorial program in `SOLI` - imperative version

## 3.6 Additional language features

### 3.6.1 Exceptions

To deal with exceptions, the EVM has two available opcodes: `REVERT` and `INVALID`. Both undo all state changes, but `REVERT` will also allow to return a value and refund all remaining gas to the caller, whereas `INVALID` will simply consume all remaining gas. Solidity uses these opcodes to handle exceptions using the `revert()`, `require()` and `assert()` functions. The `require()` and `assert()` functions receive a bool and throw the respective exception if the condition is not met. `revert()` simply throws the exception. Both `revert()` and `require()` use the `REVERT` opcode and can also receive an error message to display to the user; `assert()` uses the `INVALID` opcode.

Gas isn't modeled in `SOLI`, the main reason is because it expresses a high level language and would not be accurate to measure gas consumption since is defined for each opcode. Also the goal is to verify properties which are expressed in a symbolic way, in most of the cases this measure is not very relevant. Of course one could estimate bounds for `SOLI` commands, maybe with the help of some side tools such as the Remix compiler, and define the consumption inductively.

Require functions are statements that usually remain in the final Solidity program since they are useful to ensure valid inputs or state conditions, while asserts should only be used to verify errors in the early stages. Also the conditions that would be checked in an assert can be verified along with the desired properties in the verification.

In `SOLI` the `revert()` and `require()` functions are modeled. `revert()` corresponds to the **Revert** command, which modifies the current state type from *Normal* to *Rev*. The `require()` function is defined by the **Require** command, as a conditional statement.

$$\begin{aligned} \mathit{Require} &:: \ 's\ \mathit{bexp} \Rightarrow \ 's\ \mathit{com} \\ \mathit{Require}\ b &:= \ \textbf{If}\ b\ \textbf{Skip}\ \textbf{Revert} \end{aligned} \tag{3.5}$$

The big-step execution rules for **Require** are represented in Figure 3.4. The derivation of this and remaining execution rules described in this chapter are proven in Isabelle/HOL. They follow by unfolding definitions and using the introduction rules from Figure 3.1.

$$\frac{s \in b \qquad \Gamma \vdash \langle \textbf{Skip},\ \mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ s}{\Gamma \vdash \langle \textbf{Require}\ b,\ \mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ s}\ (\mathit{RequireTrue})$$

$$\frac{s \notin b \qquad \Gamma \vdash \langle \textbf{Revert},\ \mathit{Normal}\ s\rangle \Rightarrow \mathit{Rev}\ s}{\Gamma \vdash \langle \textbf{Require}\ b,\ \mathit{Normal}\ s\rangle \Rightarrow \mathit{Rev}\ s}\ (\mathit{RequireFalse})$$

**Figure 3.4:** Execution rules for Require

### 3.6.2 Calling a function

In Solidity, a function call can be internal or external. Internal calls are direct calls, intended for functions of the same contract, which are translated into simple jumps in EVM, therefore the current memory is not cleared. External calls, which call functions from other contracts, are made via message call, so all function arguments have to be copied to memory. The amount of ether or gas sent with the call may be specified.

External calls can be regular calls or delegate calls. A regular call is a regular message call. A delegate call is a message call where the code is executed in the context of the calling contract, this includes the message call parameters, such as sender and value, and the storage of the contract. Due to this, delegate call should only be used if one trusts the called contract to modify the storage and if the storage layout is compatible.

In order to model these types of functions, calling must be extended with the following definition, which introduces the modelling of passing arguments, resetting local variables and returning results.

$$\begin{aligned} &\mathit{call} :: [\![\ 's \Rightarrow \ 's,\ \mathit{fname},\ 's \Rightarrow \ 's \Rightarrow \ 's,\ 's \Rightarrow \ 's \Rightarrow \ 's\ \mathit{com}\ ]\!] \Rightarrow \ 's\ \mathit{com} \\ &\mathit{call}\ \mathit{pass}\ f\ \mathit{return}\ \mathit{result} := \\ &\qquad \textbf{DynCom}\ (\lambda s.\ (\textbf{Upd}\ \mathit{pass};; \\ &\qquad\qquad \textbf{Call}\ f;; \\ &\qquad\qquad (\textbf{DynCom}\ (\lambda t.\ \textbf{Upd}\ (\mathit{return}\ s);;\ \mathit{result}\ s\ t)))) \end{aligned} \tag{3.6}$$

Here **DynCom** is used to abstract over the state space and refer to certain program states, in this case the initial state $s$ is captured by the first **DynCom** and the state after executing the body of the called procedure, $t$, is captured by the second **DynCom**. The function *pass*, receives the initial state $s$ and is used to pass the arguments of the function to the intended variables in the memory before the body of the function is executed. The *return* function is used to return from the procedure by cleaning the state, that is by restoring the local variables. In the case of a function call with a return value, the *result* function is used to communicate the results to the caller environment by updating the result variable. Both *return* and *result* functions receive the states $s$ and $t$. The control flow of *call* is described is in Figure 3.5.

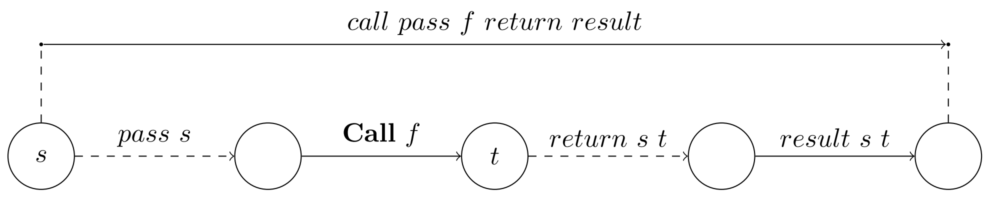

**Figure 3.5:** Control flow for *call*

The big-step execution rule for *call*, which is described in Figure 3.6, follows intuitively from the above description, first the body of the called function is executed after passing the arguments, that is, by starting in state *pass s*. Then the result command is executed after returning from the call, that is, starting in state *return s t*.

$$\frac{(the\ (\Gamma f),\ Normal\ (pass\ s))\ \Rightarrow\ Normal\ t \qquad (result\ s\ t,\ Normal\ (return\ s\ t))\ \Rightarrow\ u}{(call\ pass\ f\ return\ result,\ Normal\ s)\ \Rightarrow\ u}\ (ExecCall)$$

**Figure 3.6:** Execution rule for *call*

#### 3.6.2.A Internal call

The dynamics of this definition are described considering the simplest case of internal calling, in which the functions are in the same contract and so, the state variables and the memory are the same.

Consider the $fac$ function defined in 3.3 and the corresponding state record. In this contetx, the argument of the function is stored in the variable $a$ and the result value in the variable $res$. Consider another function in the same contract which contains a statement that internally calls $fac$ giving as argument a value stored in variable $i$ and states the result should be passed to the variable $m$. This statement is described in `SOLI` using the *call* function.

To initialize the call and pass the parameters, the value of $i$ should be copied to $a$.

$$pass\ =\ \lambda s.\ s(|a := i\ s|)$$

Since there's no actual message call, there's no need to clear the memory so the return function becomes

$$return \ = \ \lambda st.\ t$$

Finally, the execution result is communicated to the caller. Since the result is stored in $res$, the variable $m$ is updated with this value. Here, $u$ is the state after returning from the call.

$$result \ = \ \lambda st.\ \textbf{Upd}\ (\lambda u.\ u(|m := res\ t|))$$

In conclusion, to perform the described internal call in `SOLI` the following auxiliar procedure has to be defined.

$$call\ \ (\lambda s.\ s(|a := i\ s|))\ \ fac\ \ (\lambda st.\ t)\ \ (\lambda st.\ \textbf{Upd}\ (\lambda u.\ u(|m := res\ t|)))$$

### 3.6.2.B External call

For a external call both the caller and the callee contract have to be considered, so the state record of the callee is extended with the state record of the caller.

When the called function is being executed the context of execution changes, some environment variables regarding the new message call have to be updated such as *msg_sender*, *msg_value*, *msg_data* and *address_this*. The remainder, which refer to the transaction itself, are not altered.

The call is initialized by passing the parameters as previously explained and updating the sender of the message to the current address and the current address to the address of the called contract.

$$pass \ = \ \lambda s.\ s(|a := i\ s,\ msg\_sender \ := \ address\_this\ s,\ address\_this \ := \ fac\_adr\ s|)$$

To return from the procedure, only the state variables resulting from the execution of the function body are propagated to the caller. The local and environment variables (*msg_sender* and *address_this*) which were used to execute the function body are reset. After this update the state becomes equal to the initial state $s$ except for the state variables which are propagated from $t$. This is expressed as

$$return \ = \ \lambda st.\ s(|owner := owner\ t|)$$

The result of the execution is, again, communicated to the caller.

$$result \ = \ \lambda st.\ \textbf{Upd}\ (\lambda u.\ u(|m := res\ t|))$$

The auxiliary procedure to perform an external call follows.

$$\begin{aligned} call\ \ & (\lambda s.\ s(|a := i\ s,\ msg\_sender \ := \ address\_this\ s,\ address\_this \ := \ fac\_adr\ s|)) \\ & fac \\ & (\lambda st.\ s(|owner := owner\ t|) \\ & (\lambda st.\ \textbf{Upd}\ (\lambda u.\ u(|m := res\ t|))) \end{aligned}$$

### 3.6.3 Reverting state changes

In Solidity, whenever an error occurs, for instance when some condition is not satisfied, a `REVERT` exception is thrown and all state changes made in the current call must be reverted. In section 3.2 it was introduced that whenever a **Revert** occurs, the state type changes from *Normal* to *Rev*, which works as a flag for state reversion.

Suppose one wants to execute the `SOLI` statement *bdy* starting in a normal type state $s$. The execution can run without any errors and terminate in a normal state $t$. But, if an exception is thrown, the execution must be stopped with the current *Rev* state $t'$ in order to proceed to the state reversion. This is modelled with **Handle** *bdy* $c$, where $c$ is the statement which handles the reversion.

Inside $c$, first, the state is passed to a normal state in order to allow the regular `SOLI` statements, for instance **Upd** to be executed. The update of the state variables to their original value is made with the *rvrt* function, which receives the initial state $s$ and the current state $t$. Finally the error is propagated by re-throwing **Revert**. The control flow of a statement execution is described in 3.7.

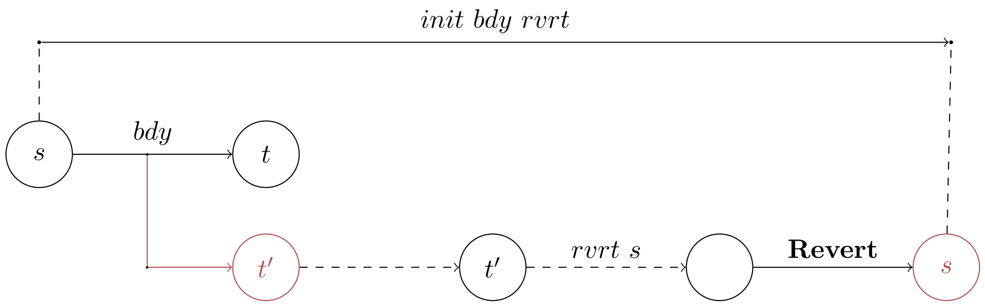

**Figure 3.7:** Control flow for *init*

In order to actually revert the state, first one needs to get hold of the initial state which can be captured using **DynCom**. Also, while taking care of the state reversion, another **DynCom** is used to refer to the current state when updating the variables.

Therefore whenever a block statement, such as a function, is written in `SOLI` it is encapsulated in an *init* function which receives the function body and the *rvrt* function which models the reset of all state variables in case of error.

$$\mathit{init} :: [\![\, 's\ \mathit{com}, 's \Rightarrow 's \Rightarrow 's ]\!] \Rightarrow 's\ \mathit{com}$$

$$\mathit{init}\ \mathit{bdy}\ \mathit{rvrt}\ :=$$

$$\mathbf{DynCom}\ (\lambda s.\ (\mathbf{Handle}\ \mathit{bdy};;$$
$$(\mathbf{DynCom}\ (\lambda t.\ \mathbf{Upd}\ (\mathit{rvrt}\ s);;\ \mathbf{Revert}))))$$

The big-step execution rules for *init* are defined in Figure 3.8. For normal execution it is immediate, is just the regular execution of *bdy*. If an exception is thrown when executing *bdy*, its execution stops in a revert type state $t$ and the full execution will terminate in state $rvrt\ s\ t$.

$$\frac{(bdy,\ Normal\ s)\ \Rightarrow\ Normal\ t}{(init\ bdy\ rvrt,\ Normal\ s)\ \Rightarrow\ Normal\ t}\ (ExecNormal)$$

$$\frac{(bdy,\ Normal\ s)\ \Rightarrow\ Rev\ t}{(init\ bdy\ rvrt,\ Normal\ s)\ \Rightarrow\ Rev\ (rvrt\ s\ t)}\ (ExecRev)$$

**Figure 3.8:** Execution rules for *init*

# 4

# Hoare Logic

In order to check the validity of a program specification, a proof system for axiomatic semantics is introduced: Hoare logic. This system was proposed by Tony Hoare in 1969 [12], [13]. After the formalization of Hoare rules for `SOLI`, a detailed explanation is introduced for the *Consequence* rule, where the propagation of auxiliary variables is described, and both *CallRec* and *Asm* rules, which are used to deal with recursive procedures. Results on soundness and (relative) completeness of the system are presented. The concept of weakest precondition is introduced, and is used in the completeness proof and the computation of verification conditions. With the goal of achieving automation of program verification, a strategy for rule application is described where a main verification condition is propagated from the end of the program to the beginning using verification condition computations, this results in a final condition to be verified: the inclusion of the precondition in this condition. Along this application some auxiliary verification conditions may be generated. An alternative formulation of rules, known as weakest precondition style and which follows this strategy, is introduced. The relation between the verification conditions and these rules is established. Finally, rules for the additional language features introduced in the previous chapter are presented.

A formula for Hoare logic, which is called a Hoare triple, is of the form

$$P\ c\ Q,$$

where $c$ is a command and $P$ and $Q$ are assertions. $P$ is said to be the precondition and $Q$ the postcondition. Semantically, an assertion is a predicate on the state. The specification of a program can be intuitively defined by such a triple. In `SOLI` assertions are defined as sets of states. For a given state $s$, an assertion $P$ is said to be true, that is, $s$ satisfies $P$, if $s \in P$.

A Hoare triple is said to be semantically valid if, the execution of $c$ starting in a state that satisfies $P$, ends in a state that satisfies $Q$. This definition corresponds to validity in terms of partial correction and is formalized in section 4.3. There is also the approach of total correctness in which termination of the program is also guaranteed. In this work we will consider partial correctness even though total correction may also be achieved by adding the notion of a well-founded relation of states to the rules which deal with cycles and recursive calls.

## 4.1 The proof system

A proof system for partial correctness of Hoare logic is introduced. Command execution can result in a *Normal* state or in a *Rev* state. To model this feature in this proof system, the postcondition is split in two, $Q$ and $A$, one for regular and other for exceptional termination.

To reason about recursive procedures, a set of assumptions $\Theta$ is introduced. This set contains function specifications, which will be used as hypothesis when proving the body of a recursive procedure. An assumption for a function is a tuple which contains its precondition, name and both postconditions.

$$'s\ assmpt\ :=\ \langle 's\ assn,\ fname,\ 's\ assn,\ 's\ assn \rangle$$

The notation used for a derivable Hoare formula, which becomes a quadruple, is associated with the procedure body environment $\Gamma$ and with the set of assumptions $\Theta$.

**Definition 8** (Hoare logic)**.** *A Hoare logic is defined for* `SOLI` *such that a derivable formula is represented by where and the proof system is constituted by the rules in figure 4.1.*

$$\Gamma, \Theta \vdash P\ c\ Q, A$$

$$\begin{aligned} \Gamma &::\ fname\ \Rightarrow\ 's\ com \\ \Theta &::\ 's\ assmpt\ set \\ P, Q, A &::\ 's\ assn \\ c &::\ 's\ com, \end{aligned}$$

There is a rule for each `SOLI` command and, additionally, the *Asm* and *Consequence* rules. To have an intuitive meaning for the rules of this system, one should read it backwards. For instance, for the *Upd* rule, if $Q$ holds (for regular execution) after the update then $P$ is the set of states such that the application of $f$ to them belongs to $Q$.

The *Skip* and *Revert* rules are dual. In the first, the precondition is the same as the postcondition for normal termination and in the second as the exceptional postcondition. The *Seq* and *Handle* rules are also dual. In the *Seq* rule the intermediary assertion $R$ connects the precondition of the second statement with the postcondition for normal termination of the first statement. For the *Handle* rule the connection is made between the precondition of the second statement and the exceptional precondition of the first.

The *If* rule also generates a proof for each of the two branches which leads to the postcondition $Q$, and for each the precondition is the set of spaces which satisfy $P$ intercepted with the sets of states which satisfy or not $b$. In the *While* rule, $P$ has the role of loop invariant. $P$ is preserved by the execution of the loop body, that is, $P$ is always satisfied after the execution of $c$ provided that is also satisfied before execution. The boolean condition $b$ is true before every loop execution and false upon loop termination.

$$\frac{}{\Gamma, \Theta \vdash Q\ \textbf{Skip}\ Q, A}\ (\mathit{Skip}) \qquad \frac{}{\Gamma, \Theta \vdash \{s.\ f\ s \in Q\}\ \ (\textbf{Upd}\ f)\ \ Q, A}\ (\mathit{Upd})$$

$$\frac{\Gamma, \Theta \vdash P\ c_1\ R, A \qquad \Gamma, \Theta \vdash R\ c_2\ Q, A}{\Gamma, \Theta \vdash P\ (\textbf{Seq}\ c_1\ c_2)\ Q, A}\ (\mathit{Seq})$$

$$\frac{\Gamma, \Theta \vdash (P \cap\ b)\ c_1\ Q, A \qquad \Gamma, \Theta \vdash (P \cap -b)\ c_2\ Q, A}{\Gamma, \Theta \vdash P\ (\textbf{If}\ b\ c_1\ c_2)\ Q, A}\ (\mathit{If}) \qquad \frac{\Gamma, \Theta \vdash (P \cap\ b)\ c\ P, A}{\Gamma, \Theta \vdash P\ (\textbf{While}\ b\ c)\ (P \cap -b), A}\ (\mathit{While})$$

$$\frac{\Gamma, \Theta \vdash P\ c_1\ Q, R \qquad \Gamma, \Theta \vdash R\ c_2\ Q, A}{\Gamma, \Theta \vdash P\ (\textbf{Handle}\ c_1\ c_2)\ Q, A}\ (\mathit{Handle})$$

$$\frac{}{\Gamma, \Theta \vdash A\ \textbf{Revert}\ Q, A}\ (\mathit{Revert}) \qquad \frac{\forall s \in P.\ \ \Gamma, \Theta \vdash P\ (c\ s)\ Q, A}{\Gamma, \Theta \vdash P\ (\textbf{DynCom}\ c)\ Q, A}\ (\mathit{DynCom})$$

$$\frac{(P, f, Q, A)\ \in\ S \qquad \forall \langle P, f, Q, A\rangle \in S.\ \ f \in \mathit{dom}\ \Gamma\ \wedge\ \Gamma, (\Theta \cup S) \vdash P\ (\mathit{the}\ (\Gamma\ f))\ Q, A}{\Gamma, \Theta \vdash P\ (\textbf{Call}\ f)\ Q, A}\ (\mathit{CallRec})$$

$$\frac{(P, f, Q, A)\ \in\ \Theta}{\Gamma, \Theta \vdash P\ (\textbf{Call}\ f)\ Q, A}\ (\mathit{Asm})$$

$$\frac{\forall s \in P'.\ \exists PQA.\ \Gamma, \Theta \vdash P\ c\ Q, A\ \wedge\ Q \subseteq Q'\ \wedge\ A \subseteq A'\ \wedge\ s \in P}{\Gamma, \Theta \vdash P'\ c\ Q', A'}\ (\mathit{Conseq})$$

**Figure 4.1:** Hoare logic for `SOLI`

In the *DynCom* rule, the triple has to hold for every state that satisfies the precondition because the dynamic command depends on the state. Detailed descriptions for the *CallRec*, *Consequence* and *Asm* rules are approached in the following sections.

### 4.1.1 The Consequence rule

In order to explain the additional structural rule of *Consequence*, used to adjust the precondition and postconditions, the concepts of stronger and weaker assertions in Hoare logic are introduced. If the truth of $P'$ can be inferred from the truth of $P$, stated as $P \subseteq P'$, then $P$ is said to be stronger than $P'$ and, reversely, $P'$ is said to be weaker than $P$.

A consequence rule allows to infer $\Gamma, \Theta \vdash P'\ c\ Q', A'$ given that $\Gamma, \Theta \vdash P\ c\ Q, A$ and the following side conditions hold: $P'$ is stronger than $P$ and both $Q'$ and $A'$ are weaker than $Q$ and $A$, respectively.

A simplified version of this rule for each case, strengthening the precondition or weakening the postconditions, is derived.

Kleymann [14] argues about the importance of auxiliary variables, also known as logical variables, in Hoare logic. These are variables that appear in the assertions but do not appear in the program itself.

$$\frac{\Gamma, \Theta \vdash P'\ c\ Q, A \qquad P \subseteq P'}{\Gamma, \Theta \vdash P\ c\ Q, A}\ (strengthen\_pre)$$

$$\frac{\Gamma, \Theta \vdash P\ c\ Q', A' \qquad Q' \subseteq Q \qquad A' \subseteq A}{\Gamma, \Theta \vdash P\ c\ Q, A}\ (weaken\_post)$$

Consider the factorial specification 4.1.

$$\langle \{|\ \acute{}a\ =\ n\ |\},\ fac,\ \{|\ \acute{}res\ =\ fac\ n\ |\},\ \{\} \rangle \tag{4.1}$$

In the precondition, the variable $n$ freezes the program variable $a$ so that one can refer to its initial value in the postcondition. Auxiliar variables allow the connection between pre and postconditions.

In the verification of recursive procedures, the specification of the procedure is assumed while verifying its body. In the factorial example an issue comes up when the specification for $n-1$ is required. It cannot be derived from the specification for $n$ using the traditional consequence rule.

Kleymann proposes a consequence rule that introduces a dependence of assertions on auxiliary variables beyond the dependence on program variables (the state space). This rule allows to modify auxiliary variables when strengthening preconditions and weakening postconditions. Nonetheless, to reason with such a rule in HOL has the condition of having a fixed type to the auxiliar variable.

The *Consequence* rule follows another path. In `SOLI` auxiliary variables are not introduced and can be stated as free variables. Therefore, to solve the issue with the factorial specification 4.1 one can simply universally quantify the auxiliary varible $n$.

However, suppose one wants to prove a formula 4.2 in which an auxiliary variable $n$ stores information about a variable $i$, in the initial state. And also, to reason about the postcondition $Q$, which depends on $i$, one needs information about the precondition on the initial state of $i$, which is saved in $n$.

$$\Gamma, \Theta \vdash \{|P\ \acute{}i|\}\ c\ \{|Q\ \acute{}i|\} \tag{4.2}$$

$$\Gamma, \Theta \vdash \{|\acute{}i = n \wedge P\ \acute{}i|\}\ c\ \{|P\ n \longrightarrow Q\ \acute{}i|\} \tag{4.3}$$

The solution is to reformulate 4.2 to 4.3 so that information about the precondition is transported to the postcondition. Since $n$ is an auxiliary variable and therefore independent from any state modification, we can infer that $P\ n$ holds in the postcondition and state it as and hypothesis for $Q\ \acute{}i$.

Generally speaking, to prove a formula, one can fix information about all program variables, which corresponds to the initial state, to the auxiliary variable Z and include this information in the postcondition.

$$\Gamma, \Theta \vdash P\ c\ Q \tag{4.4}$$

$$\forall Z.\ \Gamma, \Theta \vdash \{s.\ s = Z\ \wedge s \in P\}\ c\ \{t.\ Z \in P \longrightarrow t \in Q\} \tag{4.5}$$

The formula 4.5 is derivable from 4.4 using the simple consequence rule. It suffices to show the strengthening of the precondition 4.6 and the weakening of the postcondition 4.7, which hold trivially.

$$\{s.\ s = Z\ \wedge s \in P\}\ \subseteq P \tag{4.6}$$

$$Q \subseteq \{t.\ Z \in P \longrightarrow t \in Q\} \tag{4.7}$$

However, to prove that 4.4 is derived from 4.5, one has to prove 4.8 and 4.9. First there are no means to instantiate $Z$ as a $s$, neither to establish the fact that $Z \in P$. As seen previously the purpose of this statement is to connect the precondition with the postcondition.

$$P \subseteq \{s.\ s = Z\ \wedge s \in P\} \tag{4.8}$$

$$\{t.\ Z \in P \longrightarrow t \in Q\} \subseteq Q \tag{4.9}$$

That being said, the simple consequence rule does not suffice to deal with auxiliary variables and prove the equivalence between 4.4 and 4.5.

To instantiate $Z$ as $s$ and infer $Z \in P$, the side condition $P \subseteq P'$ must be weakened to $\forall s \in P.\ s \in P'$. The quantification of the initial state is also extended to the remaining side conditions. The Hoare formula is also incorporated into the side conditions premise. This leads to the *Consequence* rule in 4.1. Using this rule, the condition to prove becomes 4.10, which holds trivially.

$$\forall s \in P.\ s \in \{s = s\ \wedge\ s \in P\} \wedge \{t.\ s \in P \longrightarrow t \in Q\} \subseteq Q \tag{4.10}$$

It is possible to derive other consequence rules, for instance a Kleymann conequence rule. As opposed to the *Consequence* rule, the premises are split but remain connected through the auxiliary variable $Z$. Being a derived rule makes it possible that $Z$ only appears in the premises without having a fixed type. Being explicit about $Z$ may facilitate some proofs.

$$\frac{\forall Z.\ \Gamma, \Theta \vdash (P'\ Z)\ c\ (Q'\ Z), (A'\ Z) \qquad \forall s \in P.\ \exists Z.\ s \in (P'\ Z)\ \wedge\ (Q'\ Z) \subseteq Q\ \wedge\ (A'\ Z) \subseteq A}{\Gamma, \Theta \vdash P\ c\ Q, A}\ (ConseqK)$$

### 4.1.2 Recursive procedures

For the sake of simplicity, consider the parameterless call. Let $c$ be the body of the procedure $p$, executing **Call** $p$ should correspond to executing $c$. However, a issue arises when $c$ contains a call to $p$, it leads to infinite derivations.

Hoare proposes the introduction of hypothetical derivation to prevent this issue. If one reasons under the assumption that the call specification holds, one is able to prove the body $c$.

In order to do so, Hoare formulas are annotated with the set of assumptions $\Theta$. The verification of a call statement will result in the addition of its specification to $\Theta$ and verification of the procedure body. If a recursive call to that procedure appears, the *Asm* rule is used.

The general *CallRec* rule (4.1) introduces this idea for dealing with any number of procedure specifications. For instance to add all instances of a specification with auxiliary variables (see 4.1) or mutually recursive procedures. A set of specifications $S$ is introduced and every procedure body in $S$ is verified under the assumptions that the specifications hold. A side condition arises which states that all procedures must belong to the execution environment $\Gamma$.

A variant of this rule, which is explicit about procedures and auxiliary variables, can be derived. Here the assertions depend on the procedure and on the set of auxiliary variables Z. *Pr* is the set of mutually recursive procedures.

$$\frac{Pr \subseteq dom\ \Gamma \quad \forall Z.\ \Gamma, (\Theta \cup (\cup p \in Pr.\ \cup Z.\ \{\langle P\ p\ Z,\ f,\ Q\ p\ Z,\ A\ p\ Z\rangle\})) \vdash (P\ p\ Z)\ (the\ (\Gamma\ f))\ (Q\ p\ Z), (A\ p\ Z)}{\forall p \in Pr.\ \forall Z.\ \Gamma, \Theta \vdash (P\ p\ Z)\ (\mathbf{Call}\ f)\ (Q\ p\ Z), (A\ p\ Z)}\ (CallProcRec)$$

This rule can also be modified to be used directly for one single recursive procedure.

$$\frac{p \in dom\ \Gamma \qquad \forall Z.\ \Gamma, (\Theta \cup (\cup Z.\ \{\langle P\ Z,\ f,\ Q\ Z,\ A\ Z\rangle\})) \vdash (P\ Z)\ (the\ (\Gamma\ f))\ (Q\ Z), (A\ Z)}{\forall Z.\ \Gamma, \Theta \vdash (P\ Z)\ (\mathbf{Call}\ f)\ (Q\ Z), (A\ Z)}\ (CallProcRec1)$$

A simple rule that directly unfolds the body of a procedure from a Call statement may also be derived from the *CallRec* rule.

$$\frac{P \subseteq P' \qquad \Gamma, \Theta \vdash P'\ (the\ (\Gamma\ f))\ Q, A}{\Gamma, \Theta \vdash P\ (\mathbf{Call}\ f)\ Q, A}\ (CallBody)$$

## 4.2 Weakest precondition calculus

In order to verify properties about programs using Hoare logic, a backward propagation method is followed. In this method, sufficient conditions for a certain result, the postcondition, are determined. The rules are successively applied backwards, starting in the postcondition until the beginning of the program. Some side conditions may be generated.

In backward reasoning, to prove $\Gamma, \Theta \vdash P\ c\ Q, A$, one could prove that $\Gamma, \Theta \vdash P'\ c\ Q, A$ and $P \subseteq P'$ hold, using the Consequence rule. Even better, one could show that $\Gamma, \Theta \vdash P''\ c\ Q, A$, where $P''$ is the weakest precondition of $Q$ for $c$. The weakest precondition is said to be the most lenient assumption on the initial state such that $Q, A$ will hold after the execution of the command $c$. Any precondition such that the formula holds will be stronger than $P''$.

Weakest precondition calculus, also know as predicate transformer semantics (Dijkstra [5]), is a reformulation of Hoare logic. It constitutes a strategy to reduce the problem of proving a Hoare formula to the problem of proving an HOL assertion, which is called the verification condition. Since assertions are expressed as sets, reasoning about the conditions is expressed using set operations.

**Definition 9** (Weakest precondition calculus)**.** *Let $c$ be a command, $Q$ and $A$ assertions and $wp^{\Gamma,\Theta}(c, Q, A)$ the weakest precondition of $Q, A$ for $c$. The weakest precondition calculus for* `SOLI` *is inductively defined as follows*

$$
\begin{aligned}
&wp^{\Gamma,\Theta}\ (\mathbf{Skip},\ Q,\ A)\ =\ Q\\
&wp^{\Gamma,\Theta}\ (\mathbf{Revert},\ Q,\ A)\ =\ A\\
&wp^{\Gamma,\Theta}\ (\mathbf{Upd}\ f,\ Q,\ A)\ =\ \{s.\ f\ s \in Q\}\\
&wp^{\Gamma,\Theta}\ (\mathbf{Seq}\ c_1\ c_2,\ Q,\ A)\ =\ wp^{\Gamma,\Theta}\ (c_1,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A),\ A)\\
&wp^{\Gamma,\Theta}\ (\mathbf{If}\ b\ c_1\ c_2,\ Q,\ A)\ =\ \{s.\ (s \in b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (c_1,\ Q,\ A))\ \wedge\ (s \notin b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))\}\\
&wp^{\Gamma,\Theta}\ (\mathbf{While}\ c,\ Q,\ A)\ =\ \{s.\ (s \in b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (\mathbf{Seq}\ c\ (\mathbf{While}\ b\ c),\ Q,\ A))\ \wedge\ (s \notin b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))\}\\
&wp^{\Gamma,\Theta}\ (\mathbf{Call}\ f,\ Q,\ A)\ =\ P_f,\ \text{such that}\ \ f \in dom\ \Gamma\ \wedge\ (P_f,\ f,\ Q_f,\ A_f) \in \Theta\\
&wp^{\Gamma,\Theta}\ (\mathbf{DynCom}\ c,\ Q,\ A)\ =\ \bigcap_s wp^{\Gamma,\Theta}\ (c\ s,\ Q,\ A)\\
&wp^{\Gamma,\Theta}\ (\mathbf{Handle}\ c_1\ c_2,\ Q,\ A)\ =\ wp^{\Gamma,\Theta}\ (c_1,\ Q,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A)).
\end{aligned}
$$

A strategy to generate verification conditions based on this calculus and corresponding formulation of rules is described in 4.4.

## 4.3 Soundness and completeness

In this section, the concepts of semantic validity and derivability of Hoare formulas are compared. Moreover an attempt to prove their equivalence is described with respect to semantics described in 3.2. The soundness proof follows easily after introducing the notion of execution with limit on nested procedure calls. The completeness proof, and here the correct term is relative completeness, uses the notion of weakest precondition. The formal definition for validity regarding partial correctness follows.

**Definition 10** (Validity - Partial Correctness)**.**

$$\Gamma \vDash P\ c\ Q, A \quad \textit{if} \quad \forall s\ t.\ \Gamma \vdash \langle c, s\rangle \Rightarrow t\ \wedge\ s \in Normal\ `P\ \longrightarrow\ t \in Normal\ `Q\ \cup\ Rev\ `A.$$

Here, $s \in Normal\ `P$ means that there is an $s'$ such that $s = Normal\ s'$ and $s' \in P$. Executing the statement $c$ from normal state $s$ such that $P$ holds and execution ends in state $t$ implies that the precondition must hold for $t$, it can be either a normal state and $Q$ hold or in a revert state and $A$ hold.

### 4.3.1 Soundness

The goal is to prove that if a formula is derivable then it must be valid. The proof follows by induction on the derivation and consists in showing that every rule preserves validity. Since for most cases the construction of rules 4.1 follows the execution rules, except the ones for loops and procedure calls, the proof follows easily assuming the validity of their premises or is straightforward in the case of axioms. For loops additional induction on the loop body execution is introduced. In the case of calls, for instance recursive calls, it must be taken into account the set of assumptions $\Theta$ and also the depth of recursion. However the definitions of validity and big-step semantics are not rich enough to approach these properties.

First, the notion of validity is extended with the set of assumptions.

**Definition 11** (Validity with context)**.**

$$\Gamma, \Theta \models P\ c\ Q, A \quad \textit{if} \quad \forall \langle P, f, Q, A\rangle \in \Theta.\ \Gamma \models P\ (\textbf{Call}\ f)\ Q, A \ \longrightarrow\ \Gamma \models P\ c\ Q, A.$$

Also, an additional set of big-step rules to deal with depth of recursion is defined, where n is a limit on nested procedure calls.

**Definition 12** (Big-step semantics with limit)**.** *The big-step semantics for* `SOLI` *with limit on the depth of recursion are formalized by predicate*

$$\Gamma \vdash \langle c,\ s\rangle \overset{n}{\Rightarrow} t$$

*where*

$$\begin{aligned} \Gamma &:: \mathit{fname} \rightharpoonup\ '\mathit{s}\ \mathit{com} \\ c &:: '\mathit{s}\ \mathit{com} \\ s,\ t &:: '\mathit{s}\ \mathit{state}. \end{aligned}$$

The rules are equivalent to the normal big-step rules 3.1, except for the *Call* statement, where the limit $n$ is decremented.

$$\frac{\Gamma \vdash \langle \mathit{the}\ (\Gamma\ f),\ \text{Normal}\ s\rangle \overset{n}{\Rightarrow} t}{\Gamma \vdash \langle \textbf{Call}\ f,\ \text{Normal}\ s\rangle \xRightarrow{n+1} t}\ (Call_n)$$

This set of rules is monotonic regarding the limit $n$, which follows by induction on the execution depth $n$.

**Lemma 1** (Monotonicity)**.**

$$\Gamma \vdash \langle c, s\rangle \overset{n}{\Rightarrow}\ t\ \wedge\ n \leq m\ \longrightarrow\ \Gamma \vdash \langle c, s\rangle \overset{m}{\Rightarrow}\ t$$

Now validity can be established regarding the limit on nested recursive calls.

**Definition 13** (Validity with limit)**.**

$$\Gamma \overset{n}{\models} P\ c\ Q, A \quad \textit{if} \quad \forall s\ t.\ \Gamma \vdash \langle c, s\rangle \overset{n}{\Rightarrow} t\ \wedge\ s \in \mathit{Normal}\text{\`{}}\ P\ \longrightarrow\ t \in \mathit{Normal}\text{\`{}}\ Q\ \cup\ \mathit{Rev}\text{\`{}}\ A.$$

Finally the notions of validity with context and validity with limit are joined, leading to a definition which suits the needs to reason about recursive procedure calls.

**Definition 14** (Validity with limit and context)**.**

$$\Gamma, \Theta \overset{n}{\models} P\ c\ Q, A \quad \textit{if} \quad \forall \langle P, f, Q, A\rangle \in \Theta.\ \Gamma \overset{n}{\models} P\ (\textbf{Call}\ f)\ Q, A \longrightarrow\ \ \Gamma \overset{n}{\models} P\ c\ Q, A.$$

The required conditions to show that Hoare rules preserve validity are now established. An auxiliary lemma to the soundness result is stated.

**Lemma 2.**

$$(\forall n.\ \Gamma, \Theta \overset{n}{\models} P\ c\ Q, A) \quad \longrightarrow \quad \Gamma, \Theta \models P\ c\ Q, A$$

*Proof.* Using the monotonicity property, it can be shown by induction on rule structure that

$$\Gamma \vdash \langle c,\ s\rangle \Rightarrow t\ =\ (\exists n.\ \Gamma \vdash \langle c,\ s\rangle \overset{n}{\Rightarrow} t) \tag{4.11}$$

and therefore it can be inferred that a formula is valid if it is valid for all recursion depths,

$$\Gamma \models P\ c\ Q, A\ =\ (\forall n.\ \Gamma \overset{n}{\models} P\ c\ Q, A). \tag{4.12}$$

Taking context $\Theta$ into account the result follows.

□

A first result about soundness of Hoare formulas is proved regarding validity with recursion limit and context.

**Proposition 1** (Soundness within limit and context)**.**

$$\Gamma, \Theta \vdash P\ c\ Q, A \quad \longrightarrow \quad (\forall n.\ \Gamma, \Theta \overset{n}{\models} P\ c\ Q, A)$$

*Proof.* For each rule, the proof follows by structural induction. For most rules the proof is straightforward, except for *While* and *CallRec*. Here, only those cases and the *Seq* case are shown.

- c = **Seq** $c_1$ $c_2$: Consider as induction hypothesis 4.13 and 4.14, which assume validity of the premises of the rule.

$$\Gamma, \Theta \overset{n}{\models} P\ c_1\ R, A \tag{4.13}$$

$$\Gamma, \Theta \overset{n}{\models} R\ c_2\ Q, A \tag{4.14}$$

  We want to show that validity is preserved by the rule.

$$\Gamma, \Theta \overset{n}{\models} P\ (\textbf{Seq}\ c_1\ c_2)\ Q, A$$

That is, by unfolding the validity definition:

$$\begin{aligned} \forall s_1 s_3.\ (\forall \langle P, f, Q, A\rangle \in \Theta.\ \ \Gamma \overset{n}{\models} P\ (\textbf{Call}\ f)\ Q, A \wedge \\ \langle \textbf{Seq}\ c_1\ c_2,\ \mathit{Normal}\ s_1\rangle \overset{n}{\Rightarrow}\ s_3\ \wedge s_1 \in \mathit{Normal}\ 'P\ \longrightarrow \\ s_3 \in\ \mathit{Normal}\ 'Q\ \cup\ \mathit{Rev}\ 'A) \end{aligned}$$

Fixing $s_1$ and $s_3$, assume

$$\langle \textbf{Seq}\ c_1\ c_2,\ \mathit{Normal}\ s_1\rangle \overset{n}{\Rightarrow}\ s_3 \tag{4.15}$$

$$s_1 \in \mathit{Normal}\ `P \tag{4.16}$$

From 4.15 $s_2$ is obtained such that

$$\langle c_1,\ \mathit{Normal}\ s_1\rangle \overset{n}{\Rightarrow}\ s_2 \tag{4.17}$$

$$\langle c_2,\ s_2\rangle \overset{n}{\Rightarrow}\ s_3 \tag{4.18}$$

By unfolding the definition of validity in 4.13 and having both 4.17 and 4.16 follows:

$$s_2 \in \mathit{Normal}\ `R\ \cup\ \mathit{Rev}\ `A.$$

There are two cases to consider

- $s_2$ is *Normal*: from 4.18 and 4.14 follows

$$s_3 \in \mathit{Normal}\ `Q\ \cup\ \mathit{Rev}\ `A.$$

- $s_2$ is *Rev*: from 4.18 follows

$$s_3 \in \mathit{Rev}\ `A.$$

- c = **While** $b$ $c$: assuming the following hypothesis,

$$\Gamma, \Theta \overset{n}{\models} (P \cap b)\ c\ P, A \tag{4.19}$$

the goal is to show,

$$\Gamma, \Theta \overset{n}{\models} P\ (\textbf{While}\ b\ c)\ (P \cap -b), A$$

which by unfolding the definition of validity corresponds to showing

$$\forall \langle P, f, Q, A\rangle \in \Theta.\ \ \Gamma \overset{n}{\models} P\ (\textbf{Call}\ f)\ Q, A\ \wedge \tag{4.20}$$

$$\Gamma \vdash \langle \textbf{While}\ b\ c,\ \mathit{Normal}\ s\rangle \overset{n}{\Rightarrow} t\ \wedge \tag{4.21}$$

$$s \in P\ \ \longrightarrow \tag{4.22}$$

$$t \in \mathit{Normal}\ `(P \cap -b)\ \cup\ \mathit{Rev}\ `A. \tag{4.23}$$

There are two cases two consider the inital state for execution

- $s \notin b$: using the *WhileFalse* execution rule follows that $t = \mathit{Normal}\ s$, and by 4.22 that $t \in \mathit{Normal}\ `P$, furthermore $t \in \mathit{Normal}\ `(P \cap -b)$.
- $s \in b$: since a recursive loop is generated, it is necessary to use a nested induction on loop

execution

$$\Gamma \vdash \langle \textbf{While}\ b\ c,\ \mathit{Normal}\ s\rangle \overset{n}{\Rightarrow} t\ \wedge \tag{4.24}$$

$$s \in P \quad \longrightarrow \tag{4.25}$$

$$t \in \mathit{Normal}\ `(P \cap -b)\ \cup\ \mathit{Rev}\ `A. \tag{4.26}$$

From 4.25 and by using the *WhileTrue* execution rule, $r$ is obtained such that

$$\Gamma \vdash \langle c,\ \mathit{Normal}\ s\rangle \overset{n}{\Rightarrow} r \tag{4.27}$$

$$\Gamma \vdash \langle \textbf{While}\ b\ c,\ r\rangle \overset{n}{\Rightarrow} t. \tag{4.28}$$

Using hypothesis 4.19 and considering execution 4.27 and the fact that $s \in P \cap b$, it follows that

$$r \in \mathit{Normal}\ `P \cup \mathit{Rev}\ `A.$$

Again, there are two cases to consider for state $r$.

* $r \in \mathit{Rev}\ `A$: from 4.28 follows that $t = r$ hence

$$t \in \mathit{Rev}\ `A.$$

* $r \in \mathit{Normal}\ `P$: by applying the nested induction hypothesis with execution 4.28 follows

$$t \in \mathit{Normal}\ `(P \cap -b) \cup \mathit{Rev}\ `A.$$

- c = **Call** $f$: Under the hypothesis of

$$\langle P, f, Q, A\rangle \in S \tag{4.29}$$

and

$$\forall \langle P, f, Q, A\rangle \in S.\ f \in \mathit{dom}\ \Gamma\ \wedge\ (\forall n.\ \Gamma, (\Theta \cup S) \models^{n} P\ (\mathit{the}\ (\Gamma\ f))\ Q, A), \tag{4.30}$$

it is to be shown

$$\Gamma, \Theta \models^{n} P\ (\textbf{Call}\ f)\ Q, A.$$

By unfolding the definition of validity with limit and context 14 and since the call can be generalized for all specifications in $S$ because hypothesis 4.29 holds, this is equivalent to prove the following inclusion

$$\forall \langle P, f, Q, A\rangle \in \Theta.\ \ \Gamma \models^{n} P\ (\textbf{Call}\ f)\ Q, A \quad\longrightarrow\quad \forall \langle P, f, Q, A\rangle \in S.\ \ \Gamma \models^{n} P\ (\textbf{Call}\ f)\ Q, A \tag{4.31}$$

The proof follows by induction on $n$.

  - $n = 0$: follows trivially since there's no procedure execution if the recursion limit is 0.
  - $n = m + 1$: It is known that

$$\forall \langle P, f, Q, A\rangle \in \Theta.\ \ \Gamma \models^{m+1} P\ (\textbf{Call}\ f)\ Q, A.$$

By monotonicity lemma (1) follows

$$\forall \langle P, f, Q, A\rangle \in \Theta.\ \ \Gamma \models^{m} P\ (\textbf{Call}\ f)\ Q, A,$$

and by induction hypothesis,

$$\forall \langle P, f, Q, A\rangle \in S.\ \ \Gamma \models^{m} P\ (\textbf{Call}\ f)\ Q, A.$$

Then

$$\forall \langle P, f, Q, A\rangle \in (\Theta \cup S).\ \ \Gamma \models^{m} P\ (\textbf{Call}\ f)\ Q, A. \tag{4.32}$$

By unfolding the validity definition to the hypothesis 4.30 and having 4.32,

$$\forall \langle P, f, Q, A\rangle \in S.\ \ \Gamma \models^{m} P\ (\mathit{the}\ (\Gamma f))\ Q, A.$$

Using the execution rule with recursion limit follows

$$\forall \langle P, f, Q, A\rangle \in S.\ \ \Gamma \models^{m+1} P\ (\mathit{Call}\ f)\ Q, A.$$

□

The intended soundness result is finally achieved by joining proposition 1 and lemma 2.

**Proposition 2** (Soundness within context)**.**

$$\Gamma, \Theta \vdash P\ c\ Q, A \ \longrightarrow\ \Gamma, \Theta \models P\ c\ Q, A$$

### 4.3.2 Completeness

For a complete proof system, every true formula is provable within the system. And, since this is the case of an axiomatic deductive system, it follows that the set of all provable formulas is recursively enumerable. On the other hand, consider the Hoare formula

$$\Gamma \models \mathit{true}\ \textbf{Skip}\ Q, A.$$

This formula is valid iff

$$\forall s.\ s \in \mathit{true} \ \longrightarrow\ s \in \mathit{wp}^{\Gamma,\Theta}(\textbf{Skip},\ Q,\ A)$$

which is equivalent to

$$\forall s.\ s \in \mathit{true} \ \longrightarrow\ s \in Q.$$

Hence, having a recursively enumerable set of true formulas for system 4.1 could be reduced to having an recursively enumerable set of true formulas for assertions in HOL. Now, this is impossible according to Gödel's incompleteness theorem and therefore the introduced proof system cannot be complete.

Moreover, consider the Hoare formula

$$\Gamma \models \mathit{true}\ c\ \mathit{false}, \mathit{false}.$$

This formula is valid if and only if $c$ terminates for no initial state. The set of true formulas cannot be recursively enumerable because that would mean the halting problem would be decidable. Again, the proof system cannot be complete.

This incompleteness arises precisely due to the incompleteness of HOL, the language used for assertions. Though Hoare logic is not complete, due to its inheritance of HOL, its relative completeness can be proved. Cook [16] introduces the notion of relative completeness by separating incompleteness of the assertion language from incompleteness due to inadequacies in the axioms and rules for the programming language constructs. Completeness is proved on the assumption that there is an oracle which can be inquired about the validity of an HOL assertion, hence the denomination of relative completeness.

The proof relies on the concept of weakest precondition. From the definition of weakest precondition 9, the following lemma can be inferred.

**Lemma 3.**

$$\Gamma, \Theta \models P\ c\ Q, A \quad \longrightarrow \quad (s \in P \quad \longrightarrow \quad s \in wp^{\Gamma,\Theta}(c,\ Q,\ A))$$

A useful weakest precondition property regarding the derivation of a formula and its precondition is also proven.

**Lemma 4.**

$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c, Q, A)\ c\ Q, A$$

*Proof.* The proof follows by showing that the rules in system 4.1 preserve this property using structural induction on $c$.

- $c = \mathbf{Skip}$. By the *Skip* axiom
$$\Gamma, \Theta \vdash \ Q\ \mathbf{Skip}\ Q, A. \tag{4.33}$$
For the *Upd* and *Revert* cases the proof is equivalent.
- $c = \mathbf{Seq}\ c_1\ c_2$. By induction hypothesis follows
$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c_1,\ wp^{\Gamma,\Theta}(c_2,\ Q,\ A),\ A)\ c_1\ wp^{\Gamma,\Theta}(c_2,\ Q, A), A \tag{4.34}$$
$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c_2,\ Q,\ A)\ c_2\ Q, A. \tag{4.35}$$
Hence by the *Seq* rule the result follows
$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c_1,\ wp^{\Gamma,\Theta}(c_2,\ Q,\ A),\ A)\ (\mathbf{Seq}\ c_1\ c_2)\ Q, A. \tag{4.36}$$
The proof for the *Handle* case is equivalent.
- $c = \mathbf{If}\ b\ c_1\ c_2$. By induction hypothesis
$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c_1,\ Q,\ A)\ c_1\ Q, A \tag{4.37}$$
$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c_2,\ Q,\ A)\ c_2\ Q, A. \tag{4.38}$$
It is known that
$$wp^{\Gamma,\Theta}(c_1,\ Q,\ A) \ = \ wp^{\Gamma,\Theta}(c,\ Q,\ A) \cap b \tag{4.39}$$
$$wp^{\Gamma,\Theta}(c_2,\ Q,\ A) \ = \ wp^{\Gamma,\Theta}(c,\ Q,\ A) \cap -b. \tag{4.40}$$

So

$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c,\ Q,\ A) \cap b\ c_1\ Q, A \tag{4.41}$$

$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(c,\ Q,\ A) \cap -b\ c_2\ Q, A, \tag{4.42}$$

and by the *If* rule

$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}(\textbf{If}\ b\ c_1\ c_2,\ Q,\ A)\ (\textbf{If}\ b\ c_1\ c_2)\ Q, A. \tag{4.43}$$

- $c = \textbf{While}\ b\ c_1$. The following semantic equivalence holds

$$\textbf{While}\ b\ c_1\ \equiv\ \textbf{If}\ b\ (\mathit{Seq}\ c_1\ (\textbf{While}\ b\ c_1))\ \textbf{Skip}. \tag{4.44}$$

It is also possible to infer

$$\begin{aligned} wp^{\Gamma,\Theta}(\textbf{If}\ b\ (\mathit{Seq}\ c_1\ (\textbf{While}\ b\ c_1))\ \textbf{Skip},\ Q,\ A)\ &= \{s.\ (s \in b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (\textbf{Seq}\ c_1\ (\textbf{While}\ b\ c_1)),\ Q,\ A)) \land \\ &\quad (s \notin b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (\textbf{Skip},\ Q,\ A))\}\ = \\ &\quad \{s.\ (s \in b\ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (\textbf{Seq}\ c_1\ (\textbf{While}\ b\ c_1)),\ Q,\ A)) \\ &\quad \land\ (s \notin b\ \longrightarrow\ s \in Q\}\ = \\ &\quad wp^{\Gamma,\Theta}(\textbf{While}\ b\ c_1) \end{aligned}$$

By the *If* case follows that

$$\Gamma, \Theta \vdash \ wp^{\Gamma,\Theta}\ (\textbf{While}\ b\ c_1,\ Q,\ A)\ (\textbf{While}\ b\ c_1)\ Q, A. \tag{4.45}$$

- $c = \textbf{Call}\ f$. By induction hypothesis

$$(wp^{\Gamma,\Theta}\ (\mathit{the}\ (\Gamma\ f),\ Q,\ A),\ f,\ Q,\ A) \in S \tag{4.46}$$

and

$$\begin{gathered} \forall (wp^{\Gamma,\Theta}\ (\mathit{the}\ (\Gamma\ f),\ Q,\ A),\ f,\ Q,\ A) \in S.\ f \in \mathit{dom}\ \Gamma\ \land \\ \Gamma, (\Theta \cup S) \vdash\ wp^{\Gamma,\Theta}\ (\mathit{the}\ (\Gamma\ f),\ Q,\ A)\ (\mathit{the}\ (\Gamma\ f))\ Q, A. \end{gathered} \tag{4.47}$$

Suppose $P_f$ is the precondition for the specification of function $f$. Since tuples in $S$ are function specifications, it follows that

$$wp^{\Gamma,\Theta}\ (\mathit{the}\ (\Gamma\ f),\ Q,\ A) = P_f \tag{4.48}$$

and

$$(P_f,\ f,\ Q,\ A) \in S. \tag{4.49}$$

Having 4.49 and 4.47, and using the *CallRec* rule follows

$$\Gamma, \Theta \vdash\ P_f\ (\textbf{Call}\ f)\ Q, A, \tag{4.50}$$

which is equivalent to

$$\Gamma, \Theta \vdash\ wp^{\Gamma,\Theta}(\textbf{Call}\ f,\ Q, A)\ (\textbf{Call}\ f)\ Q, A. \tag{4.51}$$

- $c = \textbf{DynCom}\ c'$. By induction hypothesis

$$\forall s \in wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A).\ \ \Gamma, \Theta \vdash\ wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A)\ \ (c'\ s)\ \ Q, A. \tag{4.52}$$

Since

$$\forall s \in wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A).\ (\bigcap_s wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A) \subseteq wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A)), \tag{4.53}$$

the precondition can be strengthened

$$\forall s \in wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A).\ \ \Gamma, \Theta \vdash \bigcap_s wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A)\ \ (c'\ s)\ \ Q, A. \tag{4.54}$$

Having 4.54, by the *DynCom* rule follows

$$\Gamma, \Theta \vdash \bigcap_s wp^{\Gamma,\Theta}(c'\ s,\ Q,\ A)\ \ (\mathbf{DynCom}\ c')\ \ Q, A, \tag{4.55}$$

which is equivalent to

$$\Gamma, \Theta \vdash\ wp^{\Gamma,\Theta}(\mathbf{DynCom}\ c',\ Q, A)\ \ (\mathbf{DynCom}\ c')\ \ Q, A. \tag{4.56}$$

□

Finally, the (relative) completeness proposition can be inferred.

**Proposition 3** (Completeness)**.**

$$\Gamma, \Theta \models P\ c\ Q, A \ \longrightarrow\ \Gamma, \Theta \vdash P\ c\ Q, A$$

*Proof.* Assuming the validity holds, from lemma 3 follows that $s \in P$ implies that $s \in wp^{\Gamma,\Theta}(c, Q, A)$. Therefore the precondition of $\Gamma, \Theta \vdash wp^{\Gamma,\Theta}(c, Q, A)\ c\ Q, A$ can be strengthened, resulting in $\Gamma, \Theta \vdash P\ c\ Q, A$. □

## 4.4 Computation of verification conditions

In this section a strategy for the backward propagation method for Hoare logic is approached. Backward propagation of assertions can be made through the computation of the weakest precondition for each rule. The automation of this method can be achieved, but with the exception of some features. The invariant of a while loop must be supplied explicitly and the proof of the verification condition plus side conditions cannot be fully automated, although some simplifications can be made.

To facilitate the automation, more specifically stating an invariant, the concept of annotated command is introduced. Its syntax is defined by the polymorphic datatype *´s acom*, where *´s* is the state space type, *´s ⇒ ´s* a state-update function and *fname* the type of function names. The syntax is equivalent to normal commands except for the **While** command, which is annotated with the corresponding invariant, as an assertion.

**Definition 15** (Syntax for annotated commands)**.** *The syntax for annotated commands is defined by the polymorphic datatype ´s acom.*

$$
\begin{aligned}
{}'s\ acom \quad := \quad & \textbf{Skip} \mid \textbf{Upd}\ {}'s \Rightarrow {}'s \mid \textbf{Seq}\ {}'s\ acom\ {}'s\ acom \mid \textbf{If}\ {}'s\ bexp\ {}'s\ acom\ {}'s\ acom \\
& \mid \textbf{While}\ {}'s\ assn\ {}'s\ bexp\ {}'s\ acom \mid \textbf{Dyncom}\ {}'s \Rightarrow {}'s\ acom \mid \textbf{Call}\ fname \\
& \mid \textbf{Revert} \mid \textbf{Handle}\ {}'s\ acom\ {}'s\ acom
\end{aligned}
$$

The weakest precondition calculus for annotated commands is the same as for normal commands except for **While** where it becomes the loop invariant.

**Definition 16** (Weakest precondition calculus for annotated commands)**.** *The weakest precondition calculus for annotated commands is inductively defined as follows*

$$
\begin{aligned}
& wp^{\Gamma,\Theta}\ (\textbf{Skip},\ Q,\ A) \ = \ Q \\
& wp^{\Gamma,\Theta}\ (\textbf{Revert},\ Q,\ A) \ = \ A \\
& wp^{\Gamma,\Theta}\ (\textbf{Upd}\ f,\ Q,\ A) \ = \ \{s.\ f\ s \in Q\} \\
& wp^{\Gamma,\Theta}\ (\textbf{Seq}\ c_1\ c_2,\ Q,\ A) \ = \ wp^{\Gamma,\Theta}\ (c_1,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A),\ A) \\
& wp^{\Gamma,\Theta}\ (\textbf{If}\ b\ c_1\ c_2,\ Q,\ A) \ = \ \{s.\ (s \in b \ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (c_1,\ Q,\ A)) \ \wedge\ (s \notin b \ \longrightarrow\ s \in wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))\} \\
& wp^{\Gamma,\Theta}\ (\textbf{While}\ I\ b\ c,\ Q,\ A) \ = \ I \\
& wp^{\Gamma,\Theta}\ (\textbf{Call}\ f,\ Q,\ A) \ = \ P_f,\ \text{such that}\ \ f \in dom\ \Gamma\ \wedge\ (P_f,\ f,\ Q_f,\ A_f) \in \Theta \\
& wp^{\Gamma,\Theta}\ (\textbf{DynCom}\ c,\ Q,\ A) \ = \ \bigcap_s wp^{\Gamma,\Theta}\ (c\ s,\ Q,\ A) \\
& wp^{\Gamma,\Theta}\ (\textbf{Handle}\ c_1\ c_2,\ Q,\ A) \ = \ wp^{\Gamma,\Theta}\ (c_1,\ Q,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A)).
\end{aligned}
$$

The goal for a verification condition computation is to follow a strategy which simulates a derivation in the proof system for Hoare logic. Therefore, the $vc$ computation for each command can be obtained using the structure of each rule in the proof system.

To simplify the generation of unnecessary conditions, verification conditions are calculated independently from preconditions. An auxiliary function $vc^{\Gamma,\Theta}_{aux}$ computes the conditions generated from the structure of the rules. The main function $vc$ is then constituted by the conditions generated by $vc^{\Gamma,\Theta}_{aux}$ and a condition which verifies that the precondition implies the weakest precondition of the program.

**Definition 17** (Verification condition I)**.** *The verification condition function vc is defined as*

$$vc\ (\Gamma, \Theta \vdash\ P\ c\ Q, A) \ = \ P \subseteq wp^{\Gamma,\Theta}\ (c,\ Q,\ A)\ \cup\ vc^{\Gamma,\Theta}_{aux}\ (c,\ Q,\ A),$$

*where the auxiliary verification condition* $vc^{\Gamma,\Theta}_{aux}$ *is inductively computed as*

$$
\begin{aligned}
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{Skip},\ Q,\ A)\ =\ \emptyset\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{Revert},\ Q,\ A)\ =\ \emptyset\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{Upd}\ f,\ Q,\ A)\ =\ \emptyset\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{Seq}\ c_1\ c_2,\ Q,\ A)\ =\ vc_{aux}^{\Gamma,\Theta}\ (c_1,\ wp^{\Gamma,\Theta}(c_2,\ Q,\ A),\ A)\ \cup\ vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A)\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{If}\ b\ c_1\ c_2,\ Q,\ A)\ =\ vc_{aux}^{\Gamma,\Theta}\ (c_1,\ Q,\ A)\ \cup\ vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A)\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{While}\ I\ b\ c,\ Q,\ A)\ =\ (I \cap b) \subseteq wp^{\Gamma,\Theta}\ (c,\ I,\ A)\ \cup\ vc_{aux}^{\Gamma,\Theta}\ (c,\ I,\ A)\ \cup\ (I \cap -b) \subseteq Q\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{Call}\ f,\ Q,\ A)\ =\ Q_f \subseteq Q\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{DynCom}\ c,\ Q,\ A)\ =\ \bigcap_{s}\ vc_{aux}^{\Gamma,\Theta}\ (c\ s,\ Q,\ A)\\
&vc_{aux}^{\Gamma,\Theta}\ (\mathbf{Handle}\ c_1\ c_2,\ Q,\ A)\ =\ vc_{aux}^{\Gamma,\Theta}\ (c_1,\ Q,\ wp^{\Gamma,\Theta}(c_2,\ Q,\ A))\ \cup\ vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A).
\end{aligned}
$$

However, upon the verification of a program with any number of function calls, one has to use assume their specification is valid and belongs to the set of assumptions.

In order to do so, let $c$ be the program to be verified and consider the set $S$ which contains the specification for every function call that is generated by executing $c$. Upon the verification of $c$, the body of every function in $S$ is verified. The set $S$ is added to the set of assumption so, in case these bodies contain other calls, the specification can be assumed.

A verification condition suitable for every program is formalized.

**Definition 18** (Verification condition II)**.** *Let $S$ be the set of specifications for every function whose call is generated by the execution of $c$. The verification condition function $VC$ for $c$ is defined as*

$$VC\ (\Gamma, \Theta \vdash\ P\ c\ Q, A)\ =\ P \subseteq wp^{\Gamma,\Theta}\ (c,\ Q,\ A)\ \cup\ vc_{aux}^{\Gamma,\Theta}\ (c,\ Q,\ A)\ \cup \bigcup_{\langle P,f,Q,A\rangle \in S} vc\ (\Gamma, (\Theta \cup S) \vdash\ P\ (\mathit{the}\ (\Gamma\ f))\ Q, A).$$

## 4.4.1 An alternative formulation of rules

In order to implement these verification condition computations, which explicitly separate the main verification condition, the inclusion of precondition in the weakest precondition of the program, from auxiliary conditions generated from the structure of the rules, the main rules in 4.1 are modified to a structure that will be referred as weakest precondition style.

First, the *vc* function is applied to each case to obtain the premises of the modified rules. For the simpler cases **Skip**, **Revert** and **Upd**, the rule is obtained straightforwardly.

$$
\begin{aligned}
&vc\ (\Gamma, \Theta \vdash\ P\ \mathbf{Skip}\ Q,\ A)\ =\ P \subseteq Q\\
&vc\ (\Gamma, \Theta \vdash\ P\ \mathbf{Revert}\ Q,\ A)\ =\ P \subseteq A\\
&vc\ (\Gamma, \Theta \vdash\ P\ (\mathbf{Upd}\ f)\ Q, A)\ =\ P \subseteq \{s.\ f\ s \in Q\}
\end{aligned}
$$

From these computations the intuition for the rules follows. Note that the derivation of the new rules from the initial set of rules is proved in Isabelle using the *Consequence* rule.

$$\frac{P \subseteq Q}{\Gamma, \Theta \vdash P \ \textbf{Skip} \ Q, A} \ (Skip')$$

$$\frac{P \subseteq A}{\Gamma, \Theta \vdash P \ \textbf{Revert} \ Q, A} \ (Revert')$$

$$\frac{P \subseteq \{s.\ f\ s\ \in Q\}}{\Gamma, \Theta \vdash P \ (\textbf{Upd}\ f) \ Q, A} \ (Upd')$$

**Figure 4.2:** Weakest precondition style rules for Skip, Revert and Upd

For the **Seq** case, the intermediate assertion corresponds to the weakest precondition for the second command. Let $R = wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A)$. The unfolding of the $vc$ computation leads to no modification to the rule. This makes sense since the rule introduces a relation, as an assertion, between two statements.

$$\begin{aligned}
vc\ (\Gamma, \Theta \vdash\ P\ (\textbf{Seq}\ c_1\ c_2)\ Q, A) &= P \subseteq wp^{\Gamma,\Theta}\ (\textbf{Seq}\ c_1\ c_2,\ Q,\ A) \cup \\
&\quad vc_{aux}^{\Gamma,\Theta}\ (c_1,\ wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A),\ A) \cup vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A) \\
&= P \subseteq wp^{\Gamma,\Theta}\ (c_1,\ wp^{\Gamma,\Theta}(c_2,\ Q,\ A),\ A) \cup vc_{aux}^{\Gamma,\Theta}\ (c_1,\ R,\ A) \cup \\
&\quad R \subseteq wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A) \cup vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A) \\
&= vc\ (\Gamma, \Theta \vdash\ P\ c_1\ R, A) \cup vc\ (\Gamma, \Theta \vdash\ R\ c_2\ Q, A)
\end{aligned}$$

For the **If** case, let $P_1 = wp^{\Gamma,\Theta}\ (c_1,\ Q,\ A)$ and $P_2 = wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A)$ in the $vc$ computation. The rule follows such by stating a weakest precondition which depends on $P_1$ and $P_2$.

$$\begin{aligned}
vc\ (\Gamma, \Theta \vdash\ P\ (\textbf{If}\ b\ c_1\ c_2)\ Q, A) &= P \subseteq wp^{\Gamma,\Theta}\ (\textbf{If}\ b\ c_1\ c_2,\ Q,\ A) \cup vc_{aux}^{\Gamma,\Theta}\ (c_1,\ Q,\ A) \cup vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A) \\
&= P \subseteq \{s.\ (s \in b \longrightarrow s \in wp^{\Gamma,\Theta}\ (c_1,\ Q,\ A)) \wedge (s \notin b \longrightarrow s \in wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A))\} \cup \\
&\quad vc_{aux}^{\Gamma,\Theta}\ (c_1,\ Q,\ A) \cup vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A) \\
&= P \subseteq \{s.\ (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\} \cup \\
&\quad vc_{aux}^{\Gamma,\Theta}\ (c_1,\ Q,\ A) \cup P_1 \subseteq wp^{\Gamma,\Theta}\ (c_1,\ Q,\ A)) \cup vc_{aux}^{\Gamma,\Theta}\ (c_2,\ Q,\ A) \cup \\
&\quad P_2 \subseteq wp^{\Gamma,\Theta}\ (c_2,\ Q,\ A)) \\
&= P \subseteq \{s.\ (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\} \cup \\
&\quad vc\ (\Gamma, \Theta \vdash\ P_1\ c_1\ Q, A) \cup vc\ (\Gamma, \Theta \vdash\ P_2\ c_2\ Q, A)
\end{aligned}$$

$$\frac{P \subseteq \{s.\ (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\} \qquad \Gamma, \Theta \vdash P_1\ c_1\ Q, A \qquad \Gamma, \Theta \vdash P_2\ c_2\ Q, A}{\Gamma, \Theta \vdash P\ (\textbf{If}\ b\ c_1\ c_2)\ Q, A} \ (If')$$

**Figure 4.3:** Weakest precondition style rule for If

For the **While** case, the rule follows simply by unfolding definitions in the $vc$ computation.

$$
\begin{aligned}
vc\ (\Gamma,\Theta \vdash\ P\ (\textbf{While}\ b\ c)\ Q, A) &= P \subseteq wp^{\Gamma,\Theta}\ (\textbf{While}\ b\ c,\ Q,\ A)\ \cup\ (I \cap b) \subseteq wp^{\Gamma,\Theta}\ (c,\ I,\ A)\ \cup\ vc^{\Gamma,\Theta}_{aux}\ (c,\ I,\ A) \cup \\
&\quad (I \cap -b) \subseteq Q \\
&= P \subseteq I\ \cup\ vc\ (\Gamma,\Theta \vdash\ (I \cap b)\ c\ I, A)\ \cup\ (I \cap -b) \subseteq Q
\end{aligned}
$$

$$
\frac{P \subseteq I \qquad \Gamma,\Theta \vdash (I \cap b)\ c\ I, A \qquad (I \cap -b)\ \subseteq Q}{\Gamma,\Theta \vdash P\ (\textbf{While}\ b\ c)\ Q, A}\ (While')
$$

**Figure 4.4:** Weakest precondition style rule for While

The remaining rules such as *Handle* and *DynCom* are not taken into account since they are only used as auxiliary statements for calling procedures and reverting state changes. The rules for these more complex language structures are approached in 4.5.

Using this system of rules stated in a weakest precondition style, the method for proving Hoare formulas follows a structured and simplified strategy, in a sense that it mimics a recursive computation and that unnecessary conditions are not generated. It follows a path with the main goal of proving the implication of the precondition in the generated weakest precondition, always separating this path from the ones for auxiliary verification conditions. An implementation of a tactic for the automatic application of these rules can be written in ML, which can then be used in Isabelle/HOL proofs.

### 4.4.2 Factorial as an example

As an example for the explicit application of this method, a proof for the imperative factorial program 3.3 is presented. A factorial definition $fac$, which is used in the specification assertions, is defined in HOL.

$$
\begin{aligned}
&fac\ ::\ nat \Rightarrow nat\ where \\
&fac\ i = (if\ i \leq 0\ then\ 1\ else\ fac\ (i-1)\ \times\ i)
\end{aligned}
\tag{4.57}
$$

The goal is to prove the derivation 4.58 by a successive backwards application of the weakest precondition rules, resulting in verification conditions to be proved.

$$
\Gamma,\Theta \vdash \{|\ \acute{}a = i|\}\ fac\_imp\ \{|\ \acute{}res =\ fac\ i|\}, \{\}
\tag{4.58}
$$

After unfolding the $fac\_imp$ definition, the *Seq* rule is applied, followed by the *Upd'* rule and subset inclusion simplifications which results in the subgoal

1. $\Gamma, \Theta \vdash \{| \ s.\ a\ s = i\ s|\}$

$\quad Upd\ (b\_update\ (\lambda\_.1));;$

$\quad While\{s.\ 0 < a\ s\}\ (Upd\ (s.\ s(|b := b\ s \times a\ s|));;\ Upd\ (s.\ s(|a := a\ s - 1|))$

$\quad \{s.\ s(|res := b\ s\ |) \in \{s.\ b\ s = fac\ (i\ s)\}\}, \{\}.$

The *Seq* rule is, again, applied, followed by the *While'* rule where the invariant is supplied:

$$I = \{s.\ b\ s \times fac(a\ s) = fac(i\ s)\}$$

This results in 3 subgoals:

1. $\Gamma, \Theta \vdash \{s.\ b\ s \times fac(a\ s) = fac(i\ s)\} \cap \{s.\ 0 < a\ s\}$

   $Upd\ (s.\ s(|b := b\ s \times a\ s|));;\ Upd\ (s.\ s(|a := a\ s - 1|)$

   $\{s.\ b\ s \times fac(a\ s) = fac(i\ s)$
2. $\{s.\ b\ s\ \times\ fac(a\ s) = fac(i\ s)\} \cap -\{s.\ 0 < a\ s\} \subseteq \{s.\ s(|res := b\ s\ |) \in \{s.\ b\ s = fac\ (i\ s)\}\}$
3. $\Gamma, \Theta \vdash \{| \ s.\ a\ s = i\ s|\}\ Upd\ (b\_update\ (\lambda\_.1))\ \{s.\ b\ s \times fac(a\ s) = fac(i\ s).$

For the first subgoal, the *Seq* rule is applied, next to successive *Upd'* rules. For the third, the *Upd'* rule is applied once, which results in the final verification conditions:

1. $\{s.\ b\ s \times fac(a\ s) = fac(i\ s)\} \cap \{s.\ 0 < a\ s\} \subseteq$

   $\{s.\ s(|b := b\ s \times a\ s|) \in \{s.\ s(|a := a\ s - 1|) \in \{s.\ b\ s \times fac(a\ s) = fac(i\ s)\}\}\}$
2. $\{s.\ b\ s\ \times\ fac(a\ s) = fac(i\ s)\} \cap -\{s.\ 0 < a\ s\} \subseteq \{s.\ s(|res := b\ s\ |) \in \{s.\ b\ s = fac\ (i\ s)\}\}$
3. $\{| \ s.\ a\ s = i\ s|\} \subseteq \{s.\ s(|b := 1|) \in \{s.\ b\ s \times fac(a\ s) = fac(i\ s)\}\}.$

The first and third conditions follow by simplification using the $fac$ definition. The second condition is proved using an auxiliar lemma which states

$$0 < a\ x \implies b\ x \times a\ x \times fac(a\ x - 1) = b\ x \times (fac(a\ x - 1) \times a\ x). \tag{4.59}$$

## 4.5 Rules for derived statements

In this section the rules for additional language features described in 3.6, such as exception handling and procedure calls, are derived.

Since the *Require* command is modelled as a conditional statement, the rule is derived using 4.3. The auxiliary conditions result in $\Gamma, \Theta \vdash P_1$ **Skip** $Q, A$ and $\Gamma, \Theta \vdash P_2$ **Revert** $Q, A$, which can be omited by taking $P_1$ as $Q$ and $P_2$ as $A$ by the axioms *Skip* and *Revert.*

$$\frac{P \subseteq \{s.\ (s \in b\ \longrightarrow\ s \in Q)\ \wedge\ (s \notin b\ \longrightarrow\ s \in A)\}}{\Gamma, \Theta \vdash P\ (Require\ b)\ Q, A}\ (Require')$$

The *Init* statement corresponds to a regular execution of the body ending in a state for which the regular postcondition $Q$ holds or, in the case of an exception being thrown, the execution ends in a state such that by reverting all state changes the exceptional postcondition $A$ holds. To do so, a dependence on the initial state $s$ is introduced in the premise.

$$\frac{\forall s \in P.\ \ \Gamma, \Theta \vdash P\ bdy\ Q, \{t.\ rvrt\ s\ t \in A\}}{\Gamma, \Theta \vdash P\ (Init\ bdy\ rvrt)\ Q, A}\ (Init)$$

## 4.5.1 Calling a function

As described in 3.6.2 a *call* statement allows to model parameter passing, resetting local variables and returning a result.

So, whenever a *call* statement occurs, we want to apply a rule that uses the specification of the procedure so we can then apply the *CallRec* rule (or one of its variants) to deal with the possibility of recursion.

In order to do so, the specification must be adapted to the current context. Let $\langle P'\ Z,\ f,\ Q'\ Z,\ A'\ Z\rangle$ be the specification of $Call\ f$. A suitable instance of the auxiliary variable $Z$ must be found such that the precondition of the specification $P'\ Z$ holds.

The execution starts in state $s$ such that the precondition $P$ holds, then $Z$ is found such that $pass\ s \in P'\ Z$. It is known that $t \in Q'\ Z$ because of the specification, where $t$ is the result after execution the body of the called procedure. $R\ s\ t$ is the assertion, which depends on the initial state $s$ and $t$, that holds after returning from the call and before returning a result. This will lead to the final result such that $Q$ holds.

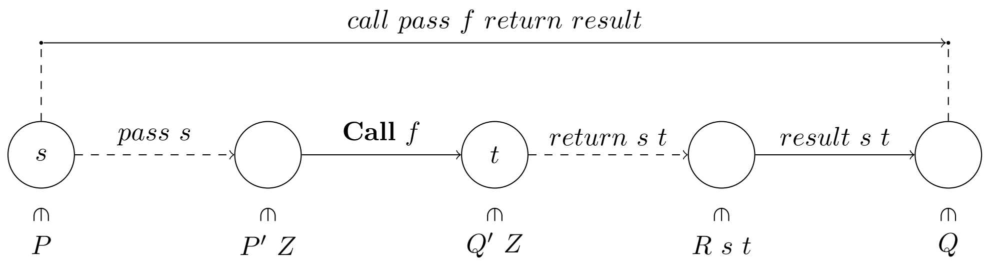

**Figure 4.5:** Control flow for *call* with assertions

To derive such a rule, the Kleymann style consequence rule is used, the explicity of $Z$ makes it easier to adapt the specification to the current context.

The weakest precondition consists in finding the suitable instance of $Z$ and stating the relation between the states in $Q'\ Z$ and $R\ s\ t$. The specification of the procedure is presented as an Hoare formula. The states $s$ and $t$ are universally quantified since a dependence on those states is created.

$$\frac{\begin{array}{c} P \subseteq \{s.\ \exists Z.\ pass\ s \in P'\ Z\ \wedge\ (\forall t.\ t \in Q'\ Z\ \longrightarrow\ return\ s\ t \in R\ s\ t)\} \\ \forall Z.\ \Gamma,\Theta \vdash (P'\ Z)\ (Call\ f)\ (Q'\ Z), A \qquad\qquad \forall st.\ \Gamma,\Theta \vdash (R\ s\ t)\ (result\ s\ t)\ Q, A \end{array}}{\Gamma,\Theta \vdash P\ (call\ pass\ f\ return\ result)\ Q, A}\ (CallRec')$$

With this rule, the method for reasoning about procedure calls can now be fully explained. When such is encountered, if it is in the form of an *call* statement, the weakest precondition style *CallRec'* rule is applied to adapt the statement to its specification, the statement becomes presented in the simple form $Call\ f$.

When the call is in the form $Call\ f$, if the called procedure presents any form of recursion, the *CallRec* rule, or one of its more specific versions (see 4.1.2), is used. Those include versions explicit about auxiliary variables and procedures, therefore being specific for a single recursive procedure *CallProcRec1*, a set of recursive procedure or even mutual recursion *CallProcRec*. The body is unfolded and whenever a recursive call appears, it is know that its specification was added to Θ and therefore the *Asm* rule can be applied.

In the case of a non recursive procedure, the *CallBody* rule is applied which simply unfolds the procedure body.

In conclusion, consider a recursive version of factorial. The function body $fac\_rec$ contains the statement $call\_fac$ which is defined as a *call* statement and corresponds to a recursive call to $fac\_rec$ for $\acute{a} - 1$.

$$\begin{array}{rl} fac\_rec & ::\ st\ com\ where \\ fac\_rec & \equiv \textbf{IF}\ (\acute{a} = 0)\ \textbf{THEN}\ \acute{r} ::= 1 \\ & \qquad\qquad\qquad \textbf{ELSE}\ (call\_fac\ ;;\ \acute{r} ::= \acute{a} \times \acute{r}) \\ & \\ call\_fac & ::\ st\ com\ where \\ call\_fac & \equiv call\ (\lambda s.\ s(\!|a := a\ s - 1|\!)) \\ & \quad fac\_rec \\ & \quad (\lambda st.\ t(\!|a := a\ s|\!)) \\ & \quad (\lambda st.\ \textbf{Upd}\ (\lambda u.\ u(\!|r := r\ t|\!))) \end{array}$$

**Figure 4.6:** Factorial program in `SOLI` - recursive version

The goal is to verify the factorial body for all possible values for $\acute{a}$. In order to do so, the auxiliary variable $n$ is introduced and generally quantified.

$$\forall n.\ \Gamma,\Theta \vdash\ \{\!|\acute{a} = n|\!\}\ \ \textbf{Call}\ fac\_rec\ \ \{\!|\acute{r} = fac\ n|\!\}, \{\}$$

Since this is a situation of a single recursive procedure with an explicit auxiliary variable, the *CallProcRec1* rule is applied. The factorial specification is added to the set of assumptions for all instances of the auxiliary variable $n$ and the body of the called procedure is unfolded. The body is verified and upon appearance of the statement $call\_fac$, the *CallRec'* rule is applied, followed by the *Asm* rule which uses the added assumption to verify the recursive call.

# 5

# Application to real-world smart contracts

In this chapter SOLI is tested on actual smart contracts, in particular some vulnerable ones. An electronic voting contract, a Token with an overflow bug and a simplified version of DAO are analyzed.

## 5.1 Electronic voting

One of the applications for smart contracts is electronic voting. In this example the *Ballot* contract[1], which features automatic and transparent vote counting and delegate voting, is presented. This contract illustrates a vast majority of Solidity's features.

Structs are complex variables which are constituted by a set of variables, in SOLI they are represented as records. The *Ballot* contract has a $Voter$ that contains the weight of the voter, which is accumulated by delegation, a boolean that states whether the person already voted, in case of vote delegation, the delegate's address, and the index of the voted proposal. It also contains a *Proposal* struct which contains the proposal name and corresponding vote count.

$$\begin{aligned} \textbf{record}\ \ & Voter \ = \\ weight \ &:: \ nat \\ voted \ &:: \ bool \\ delegate \ &:: \ address \\ vote \ &:: \ nat \end{aligned} \qquad \begin{aligned} \textbf{record}\ \ & Proposal \ = \\ name \ &:: \ char\ list \\ voteCount \ &:: \ nat \end{aligned} \tag{5.1}$$

As global variables the contract contains an address $chairperson$, a mapping $voters$ between addresses and $Voter$ structs and a list of *Proposal* structs, which are stored in the *st* record.

$$\begin{aligned} \textbf{record}\ \ st \ &= \ env + \\ chairperson \ &:: \ address \\ voters \ &:: \ address \Rightarrow Voter \\ proposals \ &:: \ Proposal\ list \end{aligned} \tag{5.2}$$

The contract is composed by the constructor function, the $giveRightToVote$ function, which increases the weight of a voter (who hasn't already voted) to 1 and can only be called by $chairperson$, the *delegate* function which can be used by a voter to delegate his vote, the *vote* function which gives the person's vote weight to a proposal, the *winningProposal* function which computes the winning proposal by finding

[1] https://solidity.readthedocs.io/en/v0.5.12/solidity-by-example.html

the maximum value of *voteCount* in the proposal list and, finally, the *winnerName* function which uses the previous computation to return the name of the winner proposal.

*INIT* is defined as an *init* statement to revert all state changes in case of exception, that is, resetting the global variables to their initial values. Every function of the contract begins with this statement.

This example is focused on the verification of the *winnerName* function, which returns the name of the winning proposal by calling the *winningProposal* function which returns the corresponding index. This function finds the maximum value of *voteCount* in the list of proposals using a loop. It introduces the necessity of supplying an invariant and to verify that, while the list is gone through, the current maximum is correctly computed. This computation compares the current value of the list with the previous maximum. The verification requires a definition on the maximum of a list and additional lemmas on the matter to be introduced.

In order to internally call the *winningProposal* function, *call_wp* is defined as a *call* statement where the result value *winningProposal_out* is passed to the variable $r$. The body function $\Gamma$ is defined so that *the* ($\Gamma$ *winningProposal*) = *winningProposal_com*.

```
winningProposal_com :: loc com
winningProposal_com ≡
       INIT( ´winningProposal_out ::= 0;;
               ´winningVoteCount ::= 0;;
               ´p ::= 0;;
               WHILE (´p < length ´proposals) DO
                      IF (´winningVoteCount < voteCount ´proposals[´p]) THEN
                             ´winningVoteCount ::= voteCount ´proposals[´p];;
                             ´winningProposal_out ::= ´p;;
                      ´p ::= ´p + 1)

winnerName_com :: loc com
winnerName_com ≡
         INIT( call_wp;;
                   ´winnerName_out ::= name ´proposals[´r])

call_wp ::  st com
call_wp ≡ call (λs. s) winningProposal (λst. t)
                      (λst. Upd (λu. u(|r := winningProposal_out t|)))
```

**Figure 5.1:** *winningProposal* and *winnerName* functions

The verification consists in showing that the returned value $r$ from the *winningProposal* function corresponds to the maximum vote count of the list and that the output of the *winnerName* function is the corresponding name. The $max'$ function was defined to retrieve the maximum of a list of natural numbers.

While reasoning about the current element of the list, located at index $p$, the list selector function is used. However, it is not defined for the empty list. If the list of proposals is empty, at some point in the verification, since $p$ is initialized as 0, the first element of the list is selected from the empty list and therefore the verification does not succeed.

The assumption that the list must be not empty is added. The assumption can be either be added to the precondition or to the actual function as a *Require* statement. Since this is assumed, the winning proposal is initialized as the list's first element which will be compared with the next element by initializing $p$ as 1. These modifications are added to the *winningProposal* function.

```
$winningProposal\_com$ :: $loc\ com$
$winningProposal\_com$ $\equiv$
    $INIT($ REQUIRE $(length\ ´proposals > 0);;$
        $´winningProposal\_out ::= 0;;$
        $´winningVoteCount ::= voteCount\ ´proposals[0];;$
        $´p ::= 1;;$
        WHILE $(´p < length\ ´proposals)$ DO
            IF $(´winningVoteCount < voteCount\ ´proposals[´p])$ THEN
                $´winningVoteCount ::= voteCount\ ´proposals[´p];;$
                $´winningProposal\_out ::= ´p;;$
          $´p ::= ´p + 1)$
```

**Figure 5.2:** *winningProposal* updated function

The properties regarding the correctness of the function outputs to be verified can be stated as the regular postcondition of a Hoare formula. The initial values for global variables are stored in the auxiliar variables *chair*, *vtrs* and *prop*.

$$\begin{aligned}\Gamma, \Theta \vdash\ & \{\!|\ chair = ´chairperson\ \wedge\ vtrs = ´voters\ \wedge\ prop = ´proposals\ |\!\}\\ & winnerName\_com\\ & \{\!|\ max'\ (map\ voteCount\ prop) = (map\ voteCount\ prop)[r]\ \wedge\\ & ´winnerName\_out = name\ prop[r]\ |\!\},\\ & \{\!|\ ´chairperson = chair\ \wedge\ ´voters = vtrs\ \wedge\ ´proposals = prop\ |\!\}\end{aligned} \tag{5.3}$$

The invariant $I$ for the function loop is supplied which states the limits that should be verified on the value of $p$ and that, while the loop iterates through the list, the current $winningVoteCount$ is the maximum of the $p$ first elements of the list.

$$\begin{aligned}I = \{\!|\ & 1\ \leq\ ´p\ \leq length\ prop\ \wedge\\ & ´winningVoteCount = max'(take\ ´p\ (map\ voteCount\ prop)[´winningProposal\_out]\ |\!\}\end{aligned}$$

The application of the verification method results, after simplification, in two conditions.

$$
\begin{aligned}
&1.\ \acute{}proposals \neq \{\} \implies \\
&\quad 1 \leq length\,\acute{}proposals\ \wedge\ voteCount\,\acute{}proposals[0] = max'\ (take\ 1\ (map\ voteCount\,\acute{}proposals)) \\
&2.\ \acute{}p < length\ \acute{}proposals \implies \\
&\quad (\ max'\ (take\,\acute{}p\ (map\ voteCount\,\acute{}proposals))\ <\ voteCount\,\acute{}proposals[\acute{}p]\ \longrightarrow \\
&\quad max'\ (take\ (\acute{}p+1)\ (map\ voteCount\,\acute{}proposals)) = voteCount\,\acute{}proposals[\acute{}p]) \\
&\quad (\ \neg\ max'\ (take\,\acute{}p\ (map\ voteCount\,\acute{}proposals))\ <\ voteCount\,\acute{}proposals[\acute{}p]\ \longrightarrow \\
&\quad max'\ (take\ (\acute{}p+1)\ (map\ voteCount\,\acute{}proposals)) = max'\ (take\,\acute{}p\ (map\ voteCount\,\acute{}proposals)))
\end{aligned}
\tag{5.4}
$$

To verify these conditions, some auxiliary lemmas are introduced. The first verification condition is generated from the simplification of the inclusion of the precondition in the weakest precondition of the program. The condition is proven using lemmas 5 and 6, together with the fact that $voteCount\,\acute{}proposals[0] = (map\ voteCount\,\acute{}proposals)[0]$. Both these lemmas are proven by induction on the structure of $l$.

**Lemma 5.**

$$l \neq \{\} \implies 1 \leq length\ l.$$

**Lemma 6.**

$$l \neq \{\} \implies l[0] = max'\ (take\ 1\ l).$$

In order to prove the second condition, which results from the invariant, lemma 7 is presented. It states, under the assumption that a list is not empty, that the maximum of the list is the maximum between the list of its first $n-1$ members and its last element.

**Lemma 7.**

$$\{x_1, \dots, x_n\} \neq \{\} \implies max'\ (\{x_1, \dots, x_n\}) = max\ (max'\ \{x_1, \dots, x_{n-1}\})\ x_n.$$

It follows by induction on the structure of the list using the definition of $max'$ and functions *last* and *butlast*. For $l = \{x_1, \dots, x_n\}$, $butlast\ l = \{x_1, \dots, x_{n-1}\}$ and $last\ l = x_n$.

Consider lemma 8.

**Lemma 8.** *Let* $\acute{}p$ *be an uint, then*

$$\acute{}p < length\ l \implies l \neq \{\}.$$

It follows that $\acute{}p < length\ \acute{}proposals \implies \acute{}proposals \neq \{\}$. Lemma 7 can now be used to prove the condition by taking $\{x_1, \dots, x_n\}$ as $take\ (\acute{}p+1)\ (map\ voteCount\,\acute{}proposals)$.

## 5.2 Ethereum tokens

Ethereum not only has its own currency, ether, but also has tokens which can act like currency. Most of Ethereum tokens follow the ERC20 standard, which defines a set of functions to be implemented by the tokens to allow integration between them. This set includes the basic operations of transferring tokens, approving allowance of tokens to a delegate account and retrieving the total supply, the balance of an address, or the allowance of an address to another.

Solidity, and specially token smart contracts, are prone to underflows and overflows since the EVM works with 256-bit unsigned integers and, therefore, all operations are performed modulo 256.

As an example of a vulnerable implementation of an ERC20 token, the Hexagon ($HXG$) token[2] will be taken into account. Amongst its global variables the contract contains a mapping *balanceOf* between addresses and their balances, a mapping *allowances* between addresses and another mapping between addresses and the allowed value. It also contains an uint *burnPerTransaction* which is set to 2.

The $Hexagon$ contract contains a $transfer$ function which is used by other contract functions to transfer a value of tokens $val$, between an address $from$ and the address *to*. During the transaction there's a fee *burnPerTransaction*, which is charged to the $from$ account. That fee is burned by being sent to the 0x0 address.

```
$transfer$ :: $loc\ com$
$transfer$ $\equiv$
    $INIT$ (**REQUIRE** ($´to \neq ´adr0$);;
            **REQUIRE** ($´balanceOf\ ´frm \geq ´val + ´burnPerTransaction$);;
            **REQUIRE** ($´balanceOf\ ´to + ´val \geq ´balanceOf\ ´to$);;
            $´balanceOf ::= ´balanceOf\ (´frm := ´balanceOf\ ´frm - (´val + ´burnPerTransaction))$;;
            $´balanceOf ::= ´balanceOf\ (´to := ´balanceOf\ ´to + ´val)$;;
            $´balanceOf ::= ´balanceOf\ (´adr0 := ´balanceOf\ ´adr0 + ´burnPerTransaction)$;;
            $´currentSupply ::= ´currentSupply - ´burnPerTransaction$)
```

**Figure 5.3:** $transfer$ function from the $Hexagon$ contract

In order for this function to meet its specification it should be the case that, after the function is executed, the balance of address $from$ decreases by $val + 2$, the balance of address *to* increases by $val$ and the balance of address $adr0$ increases by 2. In case of failure of one of the require statements, the balances should maintain their initial values. These conditions are expressed by formula 5.5 where some auxiliary variables are introduced to store initial values.

[2] https://etherscan.io/address/0xB5335e24d0aB29C190AB8C2B459238Da1153cEBA#code

$$\begin{aligned}
\Gamma,\Theta \vdash \ & \{\!|\ {}^{\grave{}}burnPerTransaction = 2 \ \wedge\ from = {}^{\acute{}}frm \ \wedge\ t = {}^{\acute{}}to \ \wedge\ a = {}^{\acute{}}adr0 \ \wedge \\
& \quad bal\_from = {}^{\grave{}}balanceOf\ from \ \wedge\ bal\_to = {}^{\grave{}}balanceOf\ t \ \wedge\ bal\_a = {}^{\grave{}}balanceOf\ a \ \wedge \\
& \quad supply = {}^{\acute{}}currentSupply \ \wedge\ from \neq a \ \wedge\ from \neq t \ \wedge\ a \neq t\ |\!\} \\
& \quad transfer \\
& \{\!|\ uint\ ({}^{\grave{}}balanceOf\ from) = uint\ bal\_from - (uint\ {}^{\acute{}}val + 2) \ \wedge \\
& \quad uint\ ({}^{\grave{}}balanceOf\ t) = uint\ bal\_to + uint\ {}^{\acute{}}val \ \wedge uint\ ({}^{\grave{}}balanceOf\ a0) = uint\ bal\_a + 2 |\!\}, \\
& \{\!|\ {}^{\grave{}}balanceOf\ from = bal\_from \ \wedge\ {}^{\grave{}}balanceOf\ t = bal\_to \ \wedge\ {}^{\grave{}}balanceOf\ a = bal\_a |\!\}
\end{aligned} \tag{5.5}$$

Note that the conditions in the regular postcondition are stated using the $uint$ Isabelle function which casts an unsigned integer to an integer. This allows to check that an operation does not underflow or overflow. For instance, for the addition operation, it is known that

$$\begin{aligned}
uint\ (x+y) \ = \ & if\ (uint\ x + uint\ y < 2^{256}) \\
& then\ uint\ x + uint\ y \\
& else\ uint\ x + uint\ y - 2^{256}.
\end{aligned} \tag{5.6}$$

Therefore the addition of $x$ and $y$ does not overflow if $uint\ (x+y) = uint\ x + uint\ y$.

However, looking at the $transfer$ function, there's no condition that prevents overflow when adding ${}^{\acute{}}val$ and ${}^{\grave{}}burnPerTransaction$. The $uint\_arith$ Isabelle tactic is used in the proof to unfold this definition, which gets stuck with a verification condition which depends precisely on that condition. One is not able to prove 5.5 since there's no way to guarantee

$$uint\ (val + 2) = uint\ val + 2 \tag{5.7}$$

and therefore prove

$$\begin{aligned}
uint\ (balanceOf\ from) &= uint\ (bal\_from - (val + 2)) \\
&= uint\ bal\_from - (uint\ val + 2)
\end{aligned} \tag{5.8}$$

This vulnerability can be exploited. Suppose the $transfer$ function is called by an attacker with $val$ equal to $2^{256} - 2$. It follows that $val + burnPerTransaction = 2^{256} - 2 + 2 = 0$ and therefore the second require statement's guard will become $balanceOf\ {}^{\acute{}}frm \geq 0$ which is always true. The balance of ${}^{\acute{}}frm$ is then decreased by 0 and the balance of ${}^{\acute{}}to$ increased by $2^{256} - 2$.

To solve this issue a require statement can be added to the $transfer$ function which checks if ${}^{\acute{}}val\ +\ {}^{\grave{}}burnPerTransaction < 2^{256}$.

An additional good practice is to use the solidity library $SafeMath$, which prevents overflow and underflow for every arithmetic operation.

Similar results regarding addition overflow were obtained for the $SMT$ and $SCA$ tokens, in the $transferProxy$ and $transferMulti$ functions, respectively. A result regarding multiplication overflow was also analyzed in the $batchTransfer$ function present in the $BEC$ token contract.

## 5.3 Reentrancy

A *DAO* is a Decentralized Autonomous Organization built using the Ethereum blockchain. Its goal is to codify the rules and decision making procedures of an organization, eliminating the need for documents and people in charge, creating a structure with decentralized control. A *DAO* has associated tokens, used to give the investors the right to vote on projects.

In 2016, an hacker exploited a bug in the *DAO* contract which resulted in the loss of approximately $50 million in ether. This was the first reentrancy attack which consisted in draining funds using the attacker's fallback function.

A fallback function is a contract's function, with no arguments or return values, which is automatically executed whenever a call is made to the contract and none of its other functions match the given function identifier or no data is supplied. This is the case when the contract receives ether, with no data specified.

The vulnerability consisted in the fact that *DAO*'s *withdraw* function uses $call.value()$ to send ether to the caller's account. Now, this triggers its fallback function, which contains arbitrary code defined by the owner.

To deepen the technical aspects of this attack, a simplified version of the *DAO* contract, *babyDao*, is approached. This version consists in a mapping *credit* between addresses and their respective balances, defined as a global variable, the constructor function, a *donate* function, a *getCredit* function and a *withdraw* function. The purpose of the *withdraw* function is to drain all caller's funds in the contract and send this value in ether to its account. This is modelled as the *call* statement *call_value* defined so that the values of some environment variables, *msg_sender*, *msg_value* and *address_this*, are updated, the balance of *user* is increased by *msg_value* and the balance of *babyDao* is decreased by the same amount. Then, the fallback function's code is executed and to return from the call, the global variables values are restored. After this operation, the credit of *user* is updated to 0. The body function Γ is defined so that $the\ (\Gamma\ withdraw) = withdraw\_com$ and $the\ (\Gamma\ fallback) = fallback\_com$.

$$
\begin{aligned}
withdraw\_com &:: loc\ com \\
withdraw\_com &\equiv INIT( \\
&\quad \mathbf{IF}\ ({}^\acute{}credit\ user > 0)\ \mathbf{THEN} \\
&\qquad\quad call\_value;; \\
&\quad {}^\acute{}credit ::= {}^\acute{}credit(user := 0))
\end{aligned}
$$

**Figure 5.4:** *withdraw* function

$$\begin{aligned}
&call\_value \ :: \ loc\ com\\
&call\_value \ \equiv call\ (\lambda s.\ s(\!|\ msg\_sender := address\_this\ s,\ ,\ msg\_value := (credit\ s)\ user\\
&\qquad address\_this := user,\\
&\qquad gs := (gs\ s)(user := ((gs\ s)\ user)(\!|balance := balance\ ((gs\ s)\ user)\\
&\qquad\qquad + (credit\ s)\ user|\!),\\
&\qquad\qquad babyDao := ((gs\ s)\ babyDao)(\!|balance := balance\ ((gs\ s)\ babyDao)\\
&\qquad\qquad - (credit\ s)\ user|\!))|\!))\\
&\quad fallback\\
&\quad (\lambda st.\ s(\!|credit := credit\ t,\ gs := gs\ t|\!))\\
&\quad (\lambda st.\ \mathbf{Skip})
\end{aligned}$$

**Figure 5.5:** *call_value* definition

To write the specification for *withdraw*, which consists in stating that the credit of *user* is drained to 0, the balance of *user* is increased by its credit value and the balance of *babyDao* is decreased by the same amount, the auxiliary variables *c*, *b* and *bdao* are introduced.

$$\begin{aligned}
\forall c\ b\ bdao.\ \Gamma,\Theta \vdash &\{\!|\ c = \acute{}credit\ user\ \wedge\ b = balance\ (\acute{}gs\ user)\ \wedge\ bdao = balance\ (\acute{}gs\ babyDao)\ |\!\}\\
&withdraw \\
&\{\!|\ \acute{}credit\ user = 0\ \wedge\ balance\ (\acute{}gs\ user) = b + c\ \wedge\ balance\ (\acute{}gs\ babyDao) = bdao - c\ |\!\}, \{\}
\end{aligned} \tag{5.9}$$

Note that *gs* denotes the world state function which maps addresses to their accounts. In the case of a so called friendly fallback function, which is modelled by a **Skip** statement, the specification 5.9 for *withdraw* holds.

However, suppose an attacker writes a *fallback* function for its own account which besides increasing the attacker's balance and decreasing the balance of *babyDao*, contains code that checks whether the balance of *babyDao* will remain bigger or equal than 0 after another possible withdraw, and if so, *withdraw* is called.

$$\begin{aligned}
&fallback\ :: loc\ com\\
&fallback\ \equiv INIT(\\
&\qquad \mathbf{IF}\ (balance\ (\acute{}gs\ babyDao) - \acute{}credit\ user \geq 0)\\
&\qquad\quad \mathbf{THEN}\ call\_withdraw)
\end{aligned}$$

**Figure 5.6:** Malicious *fallback* function

Note that *call_withdraw* is defined as *Call* statement which updates environment variables, calls *withdraw* and restores the values for global variables.

Suppose the attacker has some credit *c* and *bdao* is the total balance of *babyDao*. When the attacker calls the *withdraw* function, *call_value* transfers ether to the attacker, triggering its *fallback* function which may create another call to *withdraw*. This causes the attacker to receive the same amount of ether again and enter a recursive loop until all possible ether has been drained from *babyDao* without causing the function to fail, that is, when the guard of the conditional statement in the *fallback* function evaluates to false.

Note the attacker's credit is only set to 0 after *call_value*, and therefore, after all these recursive calls. The withdraw function ends up being called $\left\lfloor \frac{bdao}{c} \right\rfloor$ times and the attacker increases its value by $\left\lfloor \frac{bdao}{c} \right\rfloor \times c$.

A proof for this reentrancy attack is accomplished in Isabelle, using the recursive features defined for the semantics of `SOLI`.

In order to avoid this kind of attack two practices should be followed. The first consists in always updating the contact's funds, in this case the *credit* mapping, before sending those funds to the caller. The second is avoid using $call.value()$, but instead use $transfer()$ or $send()$ which are both limited to 2300 gas, which wouldn't allow heavy operations such as multiple external calls.

# 6

# Conclusions

## 6.1 Contributions

The main contribution from this thesis is the development of an imperative language and respective semantic systems, regarding a considerable subset of Solidity, based on a set of existent imperative languages in Isabelle/HOL. The main additions were the modelling of Solidity calls, both internal and external, and the possible ways to throw an exception, that is, calling a revert operation, and then proceeding to the actual state reversion. Both features use a combination of the basic `SOLI` commands, but the essential point was the use of dynamic commands, which allow to refer to states in certain steps of execution. During calls, the processes of returning and resetting global variables include the update of environment variables regarding the current message and transaction.

The relative completeness proof, uses an auxiliary lemma that involves the concept of weakest precondition. There isn't always a usual definition of this concept for every command in `SOLI`, therefore some computations for the *wp* calculus had to be introduced. Connected to the *wp* calculus, some *vc* computation cases are proposed. For instance, the weakest precondition for *Call* is assumed to be the precondition for the specification of the called function and for *DynCom* the intersection of the weakest preconditions for every instance of the dynamic command. Consequently some cases in the proof of the auxiliary lemma are also introduced.

The main advantage about using a proof assistant is the richness with which properties about programs can be expressed. From the *ballot* example, it can be seen how invariants increase the complexity of a proof, but also how that complexity can be tackled using auxiliary properties. Also, a list of several Tokens was analyzed and in most cases, upon a correct specification, the tactic *uint_arith* is able to find, or at least give a hint of, overflows and underflows. Finally, the possibility of recursion allows to model reentrancy situations and fallback function attacks.

## 6.2 Future Work

Delegate calls and gas modeling are some Solidity features to be added to the current language. The issue is that gas is measured using the total cost of operations used by EVM. There is the possibility of establishing a bound for Solidity commands, but a lot of accuracy would be lost. A possible approach would be to connect `SOLI` with a EVM Isabelle/HOL implementation, establishing a compilation from a language to another.

A lot of syntax is yet to be enhanced, not only to improve readability but to simplify and automate program specification. This can be achieved using ML, a language which can be written on top of Isabelle/HOL. Related to this subject, a ML tactic for the automatic application the weakest precondition style rules according to the described backward propagation method This tactic would generate the simplified verification conditions generated from the method.

# Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] V. Buterin, "Ethereum: A next-generation cryptocurrency and decentralized application platform." [Online]. Available: https://bitcoinmagazine.com/articles/ethereum-next-generation-cryptocurrency-decentralized-application-platform-1390528211

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2019. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[4] Y. Hirai, "Formal verification of deed contract in ethereum name service," 2016. [Online]. Available: https://yoichihirai.com/deed.pdf

[5] E. W. Dijkstra and C. S. Scholten, Predicate Calculus and Program Semantics. Berlin, Heidelberg: Springer-Verlag, 1990.

[6] B. Xavier, "Formal analysis and gas estimation for ethereum smart contracts," 2018.

[7] T. Nipkow, L. Paulson, and M. Wenzel, "Isabelle/HOL - A Proof Assistant for Higher-Order Logic," Lecture Notes on Computer Science, vol. 2283, 2002.

[8] T. Nipkow and G. Klein, "Concrete Semantics with Isabelle/HOL," 2018. [Online]. Available: http://www.concrete-semantics.org/concrete-semantics.pdf

[9] M. Wenzel, G. Bauer, and T. Nipkow, "Miscellaneous Isabelle/Isar examples," 1999.

[10] T. Nipkow and L. P. Nieto, "Owicki/Gries in Isabelle/HOL," in Proceedings of the Second International Conference on Fundamental Approaches to Software Engineering, ser. FASE '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 188–203.

[11] N. Schirmer, "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. dissertation, Technische Universität München, 2006.

[12] C. A. R. Hoare, “An axiomatic basis for computer programming,” in Communications of the ACM, 1969.

[13] ——, “Procedures and parameters: An axiomatic approach,” in Symposium on Semantics of Algorithmic Languages, 1971.

[14] T. Kleymann, “Hoare Logic and Auxiliary Variables,” Formal Aspects of Computing, vol. 11, pp. 541–566, 1999.

[15] M. J. Frade and J. S. Pinto, “Verification conditions for source-level imperative programs,” Computer Science Review, vol. 5(3), pp. 252–277, 2011.

[16] S. A. Cook, “Soundness and completeness of an axiom system for program verification,” SIAM J. Comput., vol. 7, pp. 70–90, 1978.

[17] G. Winskel, The Formal Semantics of Programming Languages: An Introduction. Cambridge, MA, USA: MIT Press, 1993.

[18] “Solidity documentation,” accessed: October 2019. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/index.html

[19] OpenZeppelin, “Ethernaut.” [Online]. Available: https://ethernaut.openzeppelin.com

# A

**Isabelle theories**

## A.1 Command syntax

```
theory Com
  imports Main
begin

type_synonym 's bexp = 's set
type_synonym 's assn = 's set
typedecl fname

datatype 's com = Skip
             | Upd 's ⇒ 's
             | Seq 's com  's com
             | If 's bexp  's com  's com
             | While 's bexp  's com
             | DynCom 's ⇒'s com
             | Call fname
             | Handle 's com  's com
             | Revert

definition Require :: 's bexp ⇒'s com where
    Require b ≡ If b Skip Revert

definition init :: 's com ⇒ ('s ⇒'s ⇒'s) ⇒'s com where
    init bdy rvrt ≡ DynCom (λs. Handle bdy (DynCom (λt.
                    Seq (Upd (rvrt s)) Revert)))

definition call:: ['s⇒'s, fname, 's⇒'s⇒'s, 's ⇒'s ⇒('s com)] ⇒'s com where
    call init f return c ≡ DynCom (λs. (Seq ((Seq (Upd init) (Call f)))
                          (DynCom (λt. Seq (Upd (return s)) (c s t)))))

end
```

## A.2 Big-step semantics

```
theory Big_step
    imports Syntax_trans
begin

datatype  's state = Normal 's | Rev 's

type_synonym 's body = fname ⇀ 's com

inductive
    big_step :: ['s body, 's com, 's state, 's state] ⇒ bool
    for Γ :: 's body
where
    Skip: Γ⊢⟨Skip, Normal s⟩ ⇒ Normal s |
    Upd: Γ⊢⟨Upd f, Normal s ⟩ ⇒ Normal (f s) |
    Seq: ⟦ Γ⊢⟨c1, Normal s1⟩ ⇒ s2 ; Γ⊢⟨c2, s2⟩ ⇒ s3 ⟧ ⟹
        Γ⊢⟨c1;;c2, Normal s1⟩⇒ s3 |
    IfTrue: ⟦ s∈b; Γ⊢⟨ c1, Normal s ⟩ ⇒ t ⟧ ⟹
        Γ⊢⟨If b c1 c2, Normal s⟩ ⇒ t |
    IfFalse: ⟦ s∉b; Γ⊢⟨ c2, Normal s ⟩ ⇒ t ⟧ ⟹
        Γ⊢⟨If b c1 c2, Normal s⟩ ⇒ t |
    WhileFalse: s∉b ⟹ Γ⊢⟨While b c, Normal s⟩ ⇒ Normal s |
    WhileTrue: ⟦ s ∈ b; Γ⊢⟨c, Normal s⟩ ⇒ s'; Γ⊢⟨ While b c, s'⟩ ⇒ t ⟧ ⟹
        Γ⊢⟨ While b c, Normal s⟩ ⇒ t |
    DynCom: Γ⊢⟨(c s), Normal s⟩ ⇒ t ⟹ Γ⊢⟨ DynCom c, Normal s⟩ ⇒ t |
    Call: ⟦ Γ⊢⟨the (body f), Normal s⟩ ⇒t ⟧ ⟹ Γ⊢⟨Call f, Normal s⟩ ⇒ t |
    HandleRevert: ⟦ Γ⊢⟨c1, Normal s⟩ ⇒ Rev r; Γ⊢⟨c2, Normal r⟩ ⇒ t⟧ ⟹
        Γ⊢⟨Handle c1 c2, Normal s⟩ ⇒ t |
    HandleNormal:  Γ⊢⟨c1, Normal s⟩ ⇒ Normal t ⟹
        Γ⊢⟨ Handle c1 c2, Normal s⟩ ⇒Normal t |
    Revert: Γ⊢⟨Revert, Normal s⟩ ⇒ Rev s |
    RevState: Γ⊢⟨c, Rev s⟩ ⇒ Rev s

lemma RequireTrue:
    ⟦ s∈b; Γ⊢⟨Skip, Normal s⟩ ⇒ Normal s ⟧ ⟹
        Γ⊢⟨Require b, Normal s⟩ ⇒ Normal s
<Proof>
```

```
lemma RequireFalse:
    ⟦ s∉b; Γ⊢⟨Revert, Normal s⟩ ⇒ Rev s ⟧ ⟹
        Γ⊢⟨Require b, Normal s ⟩ ⇒ Rev s
<Proof>

lemma ExecCall:
    ⟦ Γ⊢⟨the (Γ f), Normal (pass s)⟩ ⇒ Normal t;
        Γ⊢⟨result s t, Normal (return s t)⟩ ⇒ u ⟧ ⟹
        Γ⊢⟨call pass f return result, Normal s⟩ ⇒u
<Proof>

lemma ExecNormal:
    Γ⊢⟨bdy, Normal s⟩ ⇒ Normal t ⟹
        Γ⊢⟨init bdy rvrt, Normal s⟩ ⇒ Normal t
<Proof>

lemma ExecRev:
    Γ⊢⟨bdy, Normal s⟩ ⇒ Rev t ⟹
        Γ⊢⟨init bdy rvrt, Normal s⟩ ⇒ Rev (rvrt s t)
<Proof>

end
```

## A.3 Hoare logic

```
theory Hoare
  imports Big_step
begin

type_synonym 's assn = 's set

type_synonym 's assmpt = ('s assn × fname × 's assn × 's assn)

definition
    hoare_valid :: 's body ⇒ 's assn ⇒ 's com ⇒ 's assn ⇒ 's assn ⇒ bool
where
    Γ⊨ P c Q,A ≡ (∀s t. Γ⊢⟨c,s⟩ ⇒ t ⟶ s ∈ Normal ` P ⟶
        t ∈ Normal ` Q ∪ Rev ` A)
```

```
inductive
    hoare :: 's body ⇒ 's assmpt set ⇒ 's assn ⇒'s com ⇒ 's assn ⇒
        's assn ⇒ bool
    for Γ :: 's body
where
    Skip: Γ,Θ⊢ Q Skip Q, A |
    Upd: Γ,Θ⊢ {s. f s ∈ Q} (Upd f) Q,A |
    Seq: ⟦ Γ,Θ⊢ P c1 Q, A; Γ,Θ⊢ Q c2 R,A ⟧ ⟹
        Γ,Θ⊢ P (c1;;c2) R,A |
    If: ⟦ Γ,Θ⊢ (P ∩ b) c1 Q, A; Γ,Θ⊢ (P ∩ -b) c2 Q, A ⟧ ⟹
        Γ,Θ⊢ P (If b c1 c2) Q,A |
    While: Γ,Θ⊢ (P ∩ b) c P, A ⟹
        Γ,Θ⊢P (While b c) (P ∩ -b), A |
    DynCom: ∀s ∈ P. Γ,Θ⊢ P (c s) Q,A ⟹ Γ,Θ⊢ P (DynCom c) Q,A |
    CallRec: ⟦(P,f,Q,A)∈ Specs; ∀(P,f,Q,A) ∈ Specs. f ∈ dom Γ ∧
        Γ,(Θ ∪ Specs)⊢ P (the (Γ f)) Q, A ⟹
        Γ,Θ⊢ P (Call f) Q,A |
    Handle: ⟦ Γ,Θ⊢ P c1 Q,R; Γ,Θ⊢ R c2 Q,A ⟧ ⟹ Γ,Θ⊢(Handle c1 c2) Q,A |
    Revert: Γ,Θ⊢ A Revert Q,A |
    Asm: (P,f,Q,A)∈Θ ⟹ Γ,Θ⊢ P (Call f) Q,A |
    Conseq: ∀s ∈ P. ∃P' Q' A'. Γ,Θ⊢ P' c Q',A' ∧ s ∈ P' ∧ Q'⊆ Q ∧ A'⊆ A ⟹
        Γ,Θ⊢ P c Q,A


lemma strengthen_pre:
    ⟦ Γ,Θ⊢ P c Q,A ; P'⊆P ⟧ ⟹ Γ,Θ⊢ P' c Q,A
<Proof>


lemma weaken_post:
    ⟦ Γ,Θ⊢ P c Q,A; Q ⊆ Q'; A ⊆ A' ⟧ ⟹ Γ,Θ⊢ P c Q',A'
<Proof>


lemma ConseqK:
    ⟦ ∀Z. Γ,Θ⊢ (P' Z) c (Q' Z),(A' Z);
    ∀ s ∈ P. (∃ Z. s∈P' Z ∧ (Q' Z ⊆ Q) ∧ (A' Z ⊆ A))⟧ ⟹ Γ,Θ⊢ P c Q,A
<Proof>
```

```
%weakest precondition style rules
lemma Upd':
    P ⊆{s. f s ∈ Q} ⟹ Γ,Θ⊢ (Upd f) Q,A
<Proof>

lemma If':
    assumes c1: Γ,Θ⊢ B c1 Q, A
    assumes c2: Γ,Θ⊢ C c2 Q, A
    assumes wp: P ⊆ {s. (s ∈ b ⟶ s ∈ B) ∧ (s ∉ b ⟶ s ∈ C)}
    shows Γ,Θ⊢ P (If b c1 c2) Q, A
<Proof>

lemma Skip':
    P ⊆ Q ⟹ Γ,Θ⊢ P Skip Q,A
<Proof>

lemma Revert':
    P ⊆ A ⟹ Γ,Θ⊢ P Revert Q,A
<Proof>

lemma While':
    ⟦ P ⊆ I; Γ,Θ⊢ (I ∩ b) c I,A; (I ∩ -b) ⊆ Q ⟧ ⟹ Γ,Θ⊢ (While b c) Q,A
<Proof>

lemma Require':
  assumes wp: P ⊆ {s. (s ∈ b ⟶ s ∈ Q) ∧ (s ∉ b ⟶ s ∈ A)}
  shows Γ,Θ⊢ P (Require b) Q,A
<Proof>

%rules for call
lemma CallProcRec:
    assumes deriv_bodies: ∀p∈Procs.
    ∀Z. Γ,(Θ∪(⋃p∈Procs. ⋃Z. {(P p Z, p, Q p Z, A p Z)}))
        ⊢ (P p Z) (the (Γ p)) (Q p Z),(A p Z)
  assumes Procs_defined: Procs ⊆ dom Γ
  shows ∀p∈Procs. ∀Z. Γ,Θ⊢ (P p Z) (Call p) (Q p Z),(A p Z)
<Proof>
```

```
lemma CallProcRec1:
    assumes deriv_body:
    ∀Z. Γ,(Θ∪(⋃Z. {(P Z, p, Q Z, A Z)}) ⊢ (P Z) (the (Γ p)) (Q Z),(A Z)
    assumes p_defined: p ∈ dom Γ
    shows ∀Z. Γ,Θ⊢ (P Z) (Call p) (Q Z),(A Z)
<Proof>

lemma CallBody:
    assumes body: Γ f = Some body
    assumes deriv_body: Γ,Θ⊢ P' body Q,A
    assumes WP: P ⊆ P'
    shows Γ,Θ⊢ P (Call f) Q,A
<Proof>

%additional rules
lemma CallRec':
    assumes c: ∀s t. Γ,Θ⊢ (R s t) (result s t) Q,A
    assumes body: ∀Z. Γ,Θ⊢ (P' Z) (Call f) (Q' Z), A
    assumes pass: P ⊆ {s. ∃Z. pass s ∈ P' Z ∧
                                (∀t. t ∈ Q' Z ⟶ return s t ∈ R s t)}
    shows Γ,Θ⊢ P (call pass f return result) Q,A
<Proof>

lemma Call':
    assumes c: ∀s t. Γ,Θ⊢ (R s t) (result s t) Q,A
    assumes body: Γ f = Some body
    assumes body_der: ∀s. Γ,Θ⊢ (P' s) body {t. return s t ∈ R s t}, A
    assumes pass: P ⊆ {s. pass s ∈ P' s}
    shows Γ,Θ⊢ P (call pass f return result) Q,A
<Proof>

lemma init_exec:
    assumes body: ∀s ∈ P. Γ,Θ⊢ bdy Q, {t. rvrt s t ∈ A}
    shows Γ,Θ⊢ P (init_exec bdy rvrt) Q, A
<Proof>
```

## A.4 Ballot

```
theory Ballot
  imports Hoare Env
begin

record Voter =
    weight :: nat
    voted :: bool
    delegate :: address
    vote :: nat

record Proposal =
    name :: char list
    voteCount :: nat

%state variables
record st =  env +
    voters :: address ⇒ Voter
    proposals :: Proposal list
    chairperson :: address

record loc  = st +  %add local varibles
   %constructor
   propNames :: (char list) list  %arg
   i :: nat

   %giveRightToVote
   voter :: address  %arg

   %deleg
   t :: address %arg
   senderVd :: Voter
   to :: address
   del :: Voter
   n :: nat
   propd :: Proposal
   %vote
   senderVv :: Voter
   pr :: nat %arg
   propv :: Proposal

   %winningProposal
   winningVoteCount :: nat
   p :: nat
   winningProposal_out :: nat  %res

   %winnerName
   r :: nat  %stores call result
   winnerName_out  :: char list %res


%auxiliary functions
definition
    INIT :: loc com ⇒ loc com
where
    INIT bdy = init_exec bdy (λ s t.
                t⦃voters := voters s, proposals := proposals s,
                chairperson := chairperson s⦄)

definition
    call_wp :: fname  loc com
where
    call_wp f = call (λs. s) f (λ s t. t)
                     (λ s t. (Upd (λ u. u⦃r:= winningProposal_out t⦄)))

consts fncts :: (fname * loc com) list
definition
    Γ :: fname ⇀ loc com
where
    Γ = map_of fncts


%functions
definition
```

```
    ballot :: loc com
where
    ballot ≡ INIT(
        ´chairperson ::= ´msg_sender;;
        ´voters ::= ´voters(´chairperson:= (´voters ´chairperson)⦇weight:=1⦈);;
        ´i ::=0;;
        (WHILE  (´i < (length ´propNames)) DO
             (´proposals ::= ´proposals@[⦇name= ´propNames! ´i, voteCount=0⦈];;
             ´i ::= ´i + 1)))

definition
    giveRightToVote :: loc com
where
    giveRightToVote ≡ INIT(
        (REQUIRE (´msg_sender = ´chairperson));;
        (REQUIRE ( ¬ voted (´voters ´voter)));;
        (REQUIRE ((weight (´voters ´voter)) = 0));;
        ´voters ::= ´voters(´chairperson:= (´voters ´chairperson)⦇weight:=1⦈)))

definition
    deleg :: loc com
where
    deleg ≡ INIT(
        ´to ::= ´t;;
        ´senderVd ::= ´voters(´msg_sender);;
        (REQUIRE (¬voted ´senderVd));;
        (REQUIRE (´to ≠ ´msg_sender));;
        (WHILE ( delegate (´voters ´to) ≠ adr0) DO
            ´to ::= (delegate (´voters ´to));;
            (REQUIRE (´to ≠ ´msg_sender)));;
        ´senderVd ::= ´senderVd⦇voted:=True⦈;;
        ´senderVd ::= ´senderVd⦇delegate := ´to⦈;;
        ´del ::= ´voters(´to);;
        (IF (voted ´del) THEN (
            ´n ::= vote ´del;;
            ´propd ::= ´proposals!´n;;
            ´proposals ::= list_update ´proposals ´n
            (´propd⦇voteCount:= voteCount ´propd + weight ´senderVd⦈))
        ELSE (´del::= (´del(|weight:= weight ´del + weight ´senderVd |)))))

definition
    vote :: loc com
where
    vote ≡ INIT(
        sendr ::= ´voters(´msg_sender);;
        (REQUIRE (weight ´sendr ≠ 0));;
        (REQUIRE (¬ voted ´senderVv));;
        ´sendr ::= ´sendr(|voted := True, vote := ´pr|);;
        ´prop ::= ´proposals!´pr;;
        ´proposals ::= list_update ´proposals ´pr
                (´prop(|voteCount:= voteCount ´prop + weight ´senderVv|)))

definition
    winningProposal_com :: loc com
where
    winningProposal_com ≡ INIT(
        (´winningProposal_out ::= 0;;
        ´winningVoteCount ::= 0;;
        ´p ::= 0;;
        (WHILE (´p < (length ´proposals)) DO(
            (IF (´winningVoteCount < (voteCount (´proposals!´p))) THEN
            ´winningVoteCount ::= voteCount (´proposals!´p);;
            ´winningProposal_out ::= ´p ELSE Skip);;
          ´p ::= ´p + 1))))"

definition
    winnerName :: loc com
where
    winnerName ≡ INIT(
        call_wp winningProposal;;
        ´winnerName_out ::= name (´proposals!´r))
```

```
%1 - modifying the initialization of p
definition
    winningProposal'_com :: loc com
where
    winningProposal'_com ≡ INIT(
        (´winningProposal_out ::= 0;;
        ´winningVoteCount ::= voteCount ´proposals!0;;
        ´p ::= 1;;
        (WHILE (´p < (length ´proposals)) DO(
            (IF (´winningVoteCount < (voteCount (´proposals!´p))) THEN
            ´winningVoteCount ::= voteCount (´proposals!´p);;
            ´winningProposal_out ::= ´p ELSE Skip);;
          ´p ::= ´p + 1))))"

definition
    winnerName' :: loc com
where
    winnerName' ≡ INIT(
        call_wp winningProposal';;
        ´winnerName_out ::= name (´proposals!´r))


%2 -  modifying the initialization of p + adding a require
definition
    winningProposal2_com :: loc com
where
    winningProposal2_com ≡ INIT(
        REQUIRE (length ´proposals > 0);;
        (´winningProposal_out ::= 0;;
        ´winningVoteCount ::= voteCount ´proposals!0;;
        ´p ::= 1;;
        (WHILE (´p < (length ´proposals)) DO(
            (IF (´winningVoteCount < (voteCount (´proposals!´p))) THEN
            ´winningVoteCount ::= voteCount (´proposals!´p);;
            ´winningProposal_out ::= ´p ELSE Skip);;
          ´p ::= ´p + 1))))"
```

```
definition
    winnerName2 :: loc com
where
    winnerName2 ≡ INIT(
        call_wp winningProposal2;;
        ´winnerName_out ::= name (´proposals!´r))


overloading fncts ≡ fncts
begin
    definition
        fncts == [(winningProposal, winningProposal_com),
                  (winningProposal', winningProposal'_com),
                  (winningProposal2, winningProposal2_com)]
end


%auxiliary lemmas
primrec
    max' :: nat list ⇒ nat
where
    max' [] = 0 |
    max'(x#xs) = max x (max' xs)

lemma
    aux1_1: l=li@[ls] ⟹ (butlast l = li ∧ last l = ls)
    apply(simp)
done

lemma
    aux1_2: l ≠ [] ⟹ max' l = max (max' (butlast l)) (last l)
    apply(induction l)
    apply(auto)
done

lemma
    aux1: li@[l] ≠ [] ⟹ max' (li@[l]) = max (max' li) l
    apply(simp add: aux1_1 aux1_2)
```

```
done


lemma
    aux_true: ⋀x. (p x) < length (proposals x) ∧ (proposals x)≠[] ∧
        (winningVoteCount x) < voteCount ((proposals x)!(p x)) ∧
        winningVoteCount x =  max' (take (p x) (map voteCount (proposals x)))
        ⟹ max' (take (Suc (p x)) (map voteCount (proposals x))) =
            voteCount ((proposals x)!(p x))
    apply(simp add: take_Suc_conv_app_nth)
    apply(simp add: aux1)
    apply(auto)
done


lemma
    aux_false: ⋀x. (p x) < length (proposals x) ∧ (proposals x)≠[] ∧
        ¬(winningVoteCount x) < voteCount ((proposals x)!(p x)) ∧
        winningVoteCount x =  max' (take (p x) (map voteCount (proposals x)))
        ⟹ max' (take (Suc (p x)) (map voteCount (proposals x))) =
             (max'(take (p x) (map voteCount (proposals x))))
    apply(simp add: take_Suc_conv_app_nth)
    apply(simp add: aux1)
    apply(auto)
done


lemma
    aux2: ⋀pr p. p < length (pr) ⟹ pr ≠ []
    apply(auto)
done


lemma
    aux3: l ≠ [] ⟹ Suc 0 ≤ length l
    apply(induct l, auto)
done


lemma
    aux4: l ≠ [] ⟹ (l ! 0) = max' (take (Suc 0) l)
    apply(induct l, auto)
done
```

```
lemma
    aux5: ⋀x. (proposals x) ≠ [] ⟹
        voteCount (proposals x ! 0) = ((map voteCount (proposals x))!0)
    apply(simp)
done

%PROOFS
%original function - the verification doesn't finish due to the empty
%list case, the list selector function is not defined for the empty list
lemma
    winnerName_proof:
    shows Γ,Θ⊢ {´proposals ≠ []} winnerName
            {max' (map voteCount ´proposals) = (map voteCount ´proposals)!´r ∧
            ´winnerName_out = name(´proposals!´r)},
            {´proposals ≠ []}
    apply(unfold winnerName_def)
    apply(unfold INIT_def, rule init_exec, rule ballI)
    apply(rule Seq')
    apply(rule Upd)
    apply(unfold call_wp_def)
    apply(rule Call')
    apply(rule allI, rule allI, rule Upd', rule subset_refl)
    apply simp
    apply(rule allI, unfold winningProposal_com_def)
    apply(unfold INIT_def, rule init_exec, rule ballI)
    apply(rule Seq')
    apply(rule While' [where P = {| (0 ≤ ´p) ∧ (´p ≤ (length ´proposals)) ∧
            ´winningVoteCount = max' (take ´p (map (voteCount) ´proposals)) ∧
            ´winningVoteCount =
                (map (voteCount) ´proposals)!(´winningProposal_out)|}])
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule If')
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Upd', rule subset_refl)
    apply(rule Skip', rule subset_refl)
    prefer 3
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Upd', rule subset_refl)
    apply(simp_all)
    apply(rule subsetI)
    apply(clarsimp)
    apply(simp add: aux2 aux_false)
    apply(simp add: aux2 aux_true)
    apply(clarsimp, rule subsetI)
    apply(simp add: aux3)
    apply(simp add: aux4 aux5)
sorry

%1 - modifying the initialization of p + condition in the precondition
lemma
    winnerName'_proof:
    shows Γ,Θ⊢ {|´proposals ≠ []|} winnerName'
            {|max' (map voteCount ´proposals) = (map voteCount ´proposals)!´r ∧
            ´winnerName_out = name(´proposals!´r)|},
            {|´proposals ≠ []|}
    apply(unfold winnerName'_def)
    apply(unfold INIT_def, rule init_exec, rule ballI)
    apply(rule Seq')
    apply(rule Upd)
    apply(unfold call_wp_def)
    apply(rule Call')
    apply(rule allI, rule allI, rule Upd', rule subset_refl)
    apply simp
    apply(rule allI, unfold winningProposal'_com_def)
    apply(unfold INIT_def, rule init_exec, rule ballI)
    apply(rule Seq')
    apply(rule While' [where P = {| (1 ≤ ´p) ∧ (´p ≤ (length ´proposals)) ∧
            ´winningVoteCount = max' (take ´p (map (voteCount) ´proposals)) ∧
            ´winningVoteCount =
                (map (voteCount) ´proposals)!(´winningProposal_out)|}])
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule If')
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Upd', rule subset_refl)
    apply(rule Skip', rule subset_refl)
    prefer 3
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Upd', rule subset_refl)
    apply(clarsimp)
    apply(simp add: aux2 aux_false)
    apply(simp add: aux2 aux_true)
    apply(clarsimp, rule subsetI)
    apply(simp add: aux3)
    apply(simp add: aux4 aux5)
done

%2 - modifying the initialization of p + adding a require
lemma
    winnerName2_proof:
    shows Γ,Θ ⊢ {|prop = ´proposals ∧ chair = ´chairperson ∧ vtrs = ´voters|}
            winnerName2
            {|max' (map voteCount ´proposals) = (map voteCount ´proposals)!´r ∧
            ´winnerName_out = name(´proposals!´r)|},
            {|´proposals = prop ∧ ´chairperson = chair ∧ ´voters = vtrs |}
    apply(unfold winnerName2_def)
    apply(unfold INIT_def, rule init_exec, rule ballI)
    apply(rule Seq')
    apply(rule Upd)
    apply(unfold call_wp_def)
    apply(rule Call')
    apply(rule allI, rule allI, rule Upd', rule subset_refl)
    apply simp
    apply(rule allI, unfold winningProposal2_com_def)
    apply(unfold INIT_def, rule init_exec, rule ballI)
    apply(rule Seq')
    apply(rule While' [where P = {| (1 ≤ ´p) ∧ (´p ≤ (length ´proposals)) ∧
            ´winningVoteCount = max' (take ´p (map (voteCount) ´proposals)) ∧
            ´winningVoteCount =
                (map (voteCount) ´proposals)!(´winningProposal_out)|}])
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule If')
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Upd', rule subset_refl)
    apply(rule Skip', rule subset_refl)
    prefer 3
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Seq', rule Upd', rule subset_refl)
    apply(rule Require')
    apply(simp)
    apply(rule subset_refl)
    prefer 2
    apply clarsimp
    apply clarsimp
    apply(simp add: aux2 aux_true)
    apply(simp add: aux2 aux_false)
    apply clarsimp
    apply(simp add: aux3)
    apply(simp add: aux4 aux5)
done

end
```