ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

School of Information Sciences and Technology
Department of Informatics
Athens, Greece

Master Thesis
in
Computer Science

Blockchain-based algorithmic problem solving competitions

Efstathios Aliprantis

October 2021

Efstathios Aliprantis

Blockchain-based algorithmic problem solving competitions

October 2021

Supervisor: Asst. Prof. Eugenie Foustoucos

Athens University of Economics and Business

School of Information Sciences and Technology

Department of Informatics

Athens, Greece

Abstract

This thesis presents a blockchain-based platform where algorithmic problems can be posed as competitions, potentially, with a financial reward. A competition is won by the first individual to submit an algorithm that is both correct and bounded on time and space complexity, in accordance with the specification of the corresponding problem. Submitted algorithms must be accompanied by a formal proof of correctness which is mechanically validated by the platform. The platform is based on blockchain technology, specifically Ethereum, which ensures transparency on solution validation and allows for the automatic payment of competition rewards.

A complete system design is proposed and experimentally evaluated. The system consists of a web-based front end and a blockain-based backend. The blockain-based backend conducts competitions transparently and validates submitted algorithms and proofs mechanically. The proposed system design is evaluated by an experimental proof-of-concept implementation.

Furthermore, a theoretical setup is established in order to develop the methods and tools required to make the platform a reality. This theoretical setup includes a method for the formal specification of computational problems where problem specifications are formatted as special problem-definition algorithms. A method for proving algorithm correctness against such problem specifications is proposed based on formal program verification. A new programming language with a fully working compiler is developed for the representation of algorithms. A proof composer for proofs of algorithm correctness is developed based on a custom configuration of Hoare logic.

Περίληψη

Η παρούσα διπλωματική εργασία παρουσιάζει μια πλατφόρμα στην οποία μπορούν να διεξαχθούν διαγωνισμοί για την επίλυση αλγοριθμικών προβλημάτων, προαιρετικά, με χρηματικό έπαθλο. Νικήτρια ενός διαγωνισμού ανακηρύσσεται η πρώτη συμμετοχή που κρίνεται ορθή και ταυτόχρονα φράσσεται ως προς τη χρονική και χωρική πολυπλοκότητά της, βάσει των προδιαγραφών του υπό εξέταση αλγοριθμικού προβλήματος. Οι συμμετοχές αποτελούνται έναν αλγόριθμο-επιλυτή καθώς και μία απόδειξη ορθότητάς του, η οποία επαληθεύεται μηχανικά από την πλατφόρμα. Η πλατφόρμα βασίζεται σε τεχνολογία blockchain, συγκεκριμένα στο δίκτυο Ethereum, εξασφαλίζοντας διαφάνεια στην αξιολόγηση των συμμετοχών και επιτρέποντας την αυτόματη πληρωμή των χρηματικών επάθλων.

Η εργασία προτείνει και αξιολογεί πειραματικά μια πλήρη σχεδίαση συστήματος. Το σύστημα αποτελείται από μία διαδικτυακή διεπαφή και από ένα backend βασισμένο στο blockchain. Το backend είναι υπεύθυνο για τη διαφανή και αυτοματοποιημένη διεξαγωγή των διαγωνισμών καθώς και για την μηχανική επικύρωση των υποβαλλόμενων αλγορίθμων και αποδείξεων ορθότητας. Η προτεινόμενη σχεδίαση συστήματος αξιολογείται πειραματικά μέσω μια βασικής υλοποίησης.

Η εργασία, επίσης, εγκαθιδρύει μια θεωρητική βάση που καθιστά δυνατή την ανάπτυξη των απαραίτητων μεθόδων και εργαλείων ώστε να μπορέσει η πλατφόρμα να γίνει πραγματικότητα. Η θεωρητική αυτή βάση περιλαμβάνει μια μέθοδο για τον τυπικό ορισμό των προδιαγραφών ενός υπολογιστικού προβλήματος σύμφωνα με την οποία ένα υπολογιστικό πρόβλημα παρίσταται από έναν ειδικό αλγόριθμο-ορισμό. Συνακολούθως, αναπτύσσεται μια μέθοδος για την απόδειξη της ορθότητας αλγορίθμων σύμφωνα με τέτοιες τυπικά δοσμένες προδιαγραφές προβλημάτων. Μία νέα γλώσσα προγραμματισμού έχει σχεδιαστεί για το σκοπό της αναπαράσταση των αλγορίθμων. Η νέα αυτή γλώσσα συνοδεύεται από έναν πλήρως λειτουργικό compiler Επιπλέον, έχει αναπτυχθεί ένας βοηθός σύνθεσης αποδείξεων ορθότητας αλγορίθμων με βάση μια προσαρμοσμένη διαμόρφωση της λογικής Hoare.

Acknowledgements

I’d like to thank my supervisor Asst. Prof. Eugenie Foustoucos for inspiring the subject of this thesis while taking my background and interests into account, for her constant guidance and for her invaluable assistance in improving my theoretical background during the entire course of the Master’s program.

I’d like to thank the members of the committee, Assoc. Prof. Vangelis Markakis and Asst. Prof. Spyridon Voulgaris, for challenging this thesis to be sound in it’s theoretical aspects and well-designed in it’s engineering aspects.

Furthermore, I’d like to thank my friend Konstantinos Apergis who introduced me to the world of blockchains.

Finally, I’d like to thank my family for supporting me during the Master’s program.

Contents

1 Introduction

1.1 Motivation and Problem Statement

1.1.1 Motivation

Throughout history there have been numerous unsolved mathematical problems and for some of them a financial reward was promised to the first individual who can find a solution (see [Guy04] for a list). The main question that motivates this thesis is:

Can we make a platform where mathematical problems are posed as competitions with a potential financial reward that will be paid to the creator of the first correct solution?

Our additional requirements for such a platform would be that the validation for the correctness of proposed solutions must be automatic and error-free and in turn, any reward must automatically be paid to their creator. Furthermore, we would like to ensure that all solutions and their proofs are available for everyone.

The emerging technology of blockchain and smart contracts presents an opportunity for a platform fulfilling those requirements to be realized. Specifically, smart contracts are programs that are stored in a blockchain and thus unaltered. Smart contracts can receive data through transactions, validate that these data satisfy certain requirements and then automatically perform payments. Additionally, all transaction data on a public blockchain are publicly available. So, if mathematical questions, answers and proofs can become transaction data for a smart contract that is able to validate proofs, then our target competition platform has been realized.

Handling all the kinds of mathematical problems is of course a very ambitious goal. So, this thesis focuses on a specific problem kind: the algorithmic problems. Our view of algorithmic problems defines them according to the equation:

Algorithmic Problem = Computational Problem + Complexity Constraints

So, the questions we focus on have the form: *can we make an algorithm which solves this computational problem while satisfying certain time and space complexity constraints?*

1.1.2 Concept and Actors

Competitions have two actors roles:

Petitioner	The individual who poses a problem and initiates a competition
Creator	The individual who proposes a solution accompanied by a proof

There is no role of “reviewer” as solutions are validated by the platform.

We focus on algorithmic problems, thus petitioners provide definitions of algorithmic problems while creators provide algorithms. A definition for an algorithmic problem consist of the following:

- A definition for the underlying computational problem which will serve as a specification for the correctness of proposed algorithms
- A set of complexity constraints that must be satisfied by proposed algorithms

Creators submit algorithms in the form of a procedural program and accompany them with a proof for their correctness and their worst case asymptotic complexity. Proofs are composed with the utilization of formal program verification methods.

1.2 Thesis Structure

Chapter 2
In chapter 2 we give some background and present some related works about competition platforms, formal program verification, automated theorem proving systems, blockchains/smart contracts and their intersections.

Chapter 3
In chapter 3, we first establish a generic abstract framework that is able to formally describe the conducting of competitions regarding general mathematical questions. Subsequently, we specialize this abstract framework for the domain of algorithms, thus producing the theoretical ground for our algorithmic problem solving competition platform. The main result of this chapter is the proposition of a method for defining algorithmic problems which derives from the concept of the verifier algorithm used in the study of NP-completeness. This problem definition method is then enriched to express worst-case time and space complexity requirements. Finally, chapter 3 describes the process of validating an algorithm’s correctness and worst-case complexity against a given problem definition with the use of Hoare logic.

Chapter 4

In chapter 4, we present the design of our blockchain-based platform which corresponds to the theoretical framework introduced in chapter 3. Our platform comprises of a web-based front end and a purely blockchain-based back end. The front end includes a compiler for problem definitions and solver algorithms and a proof composer for composing proofs of correctness for solver algorithms with the use of Hoare logic. The blockchain-based back end includes smart contract components for conducting competitions, validating and storing compiled problem definitions and solver algorithms and validating proofs of correctness for solver algorithms. All these components are fully functional except for the last one, the validator for proofs of correctness, which is experimental work with limited capabilities that serves for the evaluation of the design.

Chapter 5

In chapter 5 we evaluate the concept experimentally. For the experiment we submit various problem definitions and solvers and a single proof of correctness. We measure the gas cost of the submissions. We also discuss the ease of use of the platform. Lastly, we attempt to reason about the feasibility of the concept for real world use cases.

Chapter 6

In chapter 6 we present our conclusion regarding the feasibility of the concept and the conclusions that arose from the evaluation. Furthermore, we propose future work that will lead to a more complete, efficient and easy to use platform.

Background and Related Work 2

2.1 Background

We shall present some background in the fields of competition platforms, formal program verification, automated theorem proving systems, blockchains/smart contracts and their intersections.

Automated theorem proving allows to automatically prove formal theorems or validate proofs of formal theorems. The automation of a proof is often regarded as guaranty for it's correctness. Present day systems for automated theorem proving include Isabelle/HOL[1] [NWP02] and Coq[2] [Bar+97].

Formal program verification allows to formally prove statements about programs. It began in the decade of 1960 with the works of Floyd [Flo67] and Hoare [Hoa69] and since then has been extended to handle more advanced program elements such as pointers [Bor00], global variables [Coo78] and recursive procedures [AB90]. Furthermore, formal program verification has been proposed to prove bounds for the time complexity of programs [Nie84]. Regarding reading material on formal program verification, Huth and Ryan [HR04] approach the task by it's foundations in Logic while Nipkow and Klein [NK14] utilize an automated theorem proving system to both introduce the theoretical aspects and present practical applications.

Smart contracts in blockchain is a relatively new, yet emerging, field that got popularized by the Ethereum[3] network introduced by Buterin [But13] and Wood [Woo14]. The Ethereum network is fully stable and used in production business systems.

Automated theorem proving on blockchain is starting to show some studies such as [Su18] and [Car+21], both proposing interactive blockhain-based theorem provers which utilize financial incentivization mechanisms. However, to the best of the author's knowledge, no work has been presented on performing formal program verification in the blockchain (note that the opposite i.e. performing formal program verification *of* smart contracts by the use of off-the-blockchain tools has had considerable attention).

[1]`https://isabelle.in.tum.de/`
[2]`https://coq.inria.fr/`
[3]`https://ethereum.org/en/`

Finally, in regards to competition platforms for scientific problems, the most notable one is Kaggle[4] which specializes in data mining problems.

2.2 Related Work

Brodal [Bro] lists various platforms for algorithm programming competitions. However, the listed platforms pose already solved problems as a means to challenge the contestants rather than tackling problems of interest with new algorithms.

When it comes to theorem proving in blockchain, we studied two related works. The first work by Su [Su18] proposes a new blockchain where transactions can prove mathematical theorems taking previous transactions as antecedents. The second work by Carré et al. [Car+21] uses the technology of smart contracts to validate proofs about formal statements. A notable fact about the later work is that it regards a proof as a tree, and requires a proof node to be formally validated in the blockchain only when it's truth is being doubted thus, trivial points in proofs will not require formal validation, effectively saving resources.

[4]`https://www.kaggle.com/competitions`

3 Theoretical Setup

This chapter presents the theoretical setup for automatically conducting algorithmic problem solving competitions. By formalizing the general concept of automated competitions, it appears that automated algorithmic problem solving competitions *mainly* require a) a method for defining computational problems and b) a method for proving the correctness of proposed algorithms.

A method for defining computational problems is proposed in this chapter, where computational problems are uniquely defined by a special problem-definition algorithm. Subsequently, a method for proving algorithm correctness against such problem definitions is developed with the help of Hoare logic.

3.1 Abstract Framework for Automated Competitions

In this section, we establish a generic and abstract framework that is able to formally describe the conducting of competitions regarding formally posed mathematical questions. This framework is abstract as it is not adequate to support any kind of competitions without further work. This framework is generic in the sense that it defines the generic concept of a competition and describes specific components that need to be defined/implemented for competitions of a certain kind to be supported. Namely, for any competition kind, these components are: a family of mathematical objects Q that contains all valid question/problems/specifications that can be posed, a family of mathematical objects A that contains all possible answers/solutions/participations that may be given in response to any $q \in Q$ and, finally, a proof validator mechanism v that is able to validate a proof for the suitability of a given $a \in A$ for a given $q \in Q$. In the next section, we fully define this triplet for algorithmic problem solving competitions.

This section first provides some core definitions and then describes the flow of competitions.

3.1.1 Definitions

Below we define the core notions for our abstract competition framework.

3.1.1.1 Problems

Definition 3.1 (Problem Kind)**.** A problem kind K is defined by a pair (Q_K, A_K) where Q_K is a family of objects which contains the representations of all questions that can be posed and A_K is a family of objects which contains the representations of all answers that may be given. A question $q \in Q_K$ specifies the requirements of a problem in K while an answer $a \in A_K$ aims to comply with the requirements of specific $q \in Q_K$.

► In other words, a problem kind consists of the definition of what is considered a valid question/problem/specification and what is considered a valid answer/solution/participation.

3.1.1.2 Proof validators

Definition 3.2 (Proof Validator)**.** A proof validator v that serves a problem kind $K = (Q_K, A_K)$ is a mechanism[1] capable of determining the suitability of an answer $a \in A_K$ for a question $q \in Q_K$. A proof validator receives a $q \in Q_K$, an $a \in A_K$ and a third object p which is a proof of correctness (suitability) of a for q. Effectively, v works as a partial function v: $Q_K \times A_K \times P_v \to \{True\}$ where P_v is the family of objects that v comprehends as proofs of correctness. Given $q \in Q_K$, $a \in A_K$ and $p \in P_v$, if $v(q, a, p) = True$ then v deems a to be correct (suitable) for q.

Let, now, v be a proof validator which serves a problem kind $K = (Q_K, A_K)$.

Definition 3.3 (Proof Validator Soundness)**.** v is *sound* if for every $q \in Q_K$ and $a \in A_K$ the existence of a p such that $v(q, a, p) = True$ necessarily means that a is correct for q.

► In other words, a proof validator is sound if it is never wrong whenever it deems an answer correct.

Definition 3.4 (Proof Validator Completeness)**.** v is *compete* if for every pair $q \in Q_K$ and $a \in A_K$ such that a is correct for q there exists p such that $v(q, a, p) = True$.

[1]In the simplest (pure) case, a proof validator is an algorithm, however, a mechanism that works interactively or is time dependant is also acceptable in the general case.

► In other words, a proof validator is complete if for every correct answer, that proof validator is able to deem it correct.

Definition 3.5 (Proof Validator Pureness)**.** v is pure if it's response for given (q, a, p) is a function exclusively of those (q, a, p) and does not depend on any other information neither is time depended.

► In case a proof validator is pure, it's function is simplified to the execution of an algorithm. In our framework, the usage of a pure proof validator is adequate in order to automate the conducting of competitions.

3.1.1.3 Competitions

Definition 3.6 (Competition Kind)**.** A competition kind K_c is defined by a pair $((Q_K, A_K), v)$ where (Q_K, A_K) is a problem kind and v is a proof validator which serves that problem kind.

► In other words, Competition Kind = Problem Kind + Proof Validator.

Definition 3.7 (Competition Declaration)**.** The declaration C_d of a competition is defined by a pair (q, K_c) where $K_c = ((Q_K, A_K), v)$ is a competition kind and $q \in Q_K$ is the question/problem/specification of the competition.

► In other words, Competition Declaration = Question + Competition Kind.

Definition 3.8 (Competition Progress)**.** The progress C_p of a competition with declaration $C_d = (q, ((Q_K, A_K), v))$ *at a specific point in time* consists of the sequence $((a_i, p_i))$ of the answers $a_i \in A_K$ submitted up until that point and the proofs p_i following them. The competition has successfully concluded if there exist $(a, p) \in C_p$ such that $v(q, a, p) = True$.

► In other words, a competition concludes when an answer and it's accompanying proof make the proof validator respond positively.

3.1.1.4 Corollary

By having a fully defined competition kind $K_c = ((Q_K, A_K), v)$, that is a formal notion of what is a question, a formal notion of what is an answer and a proof validator to judge answers, we can develop a system able to conduct competitions. Furthermore, if v is pure then these competitions can be automated to the maximum extend.

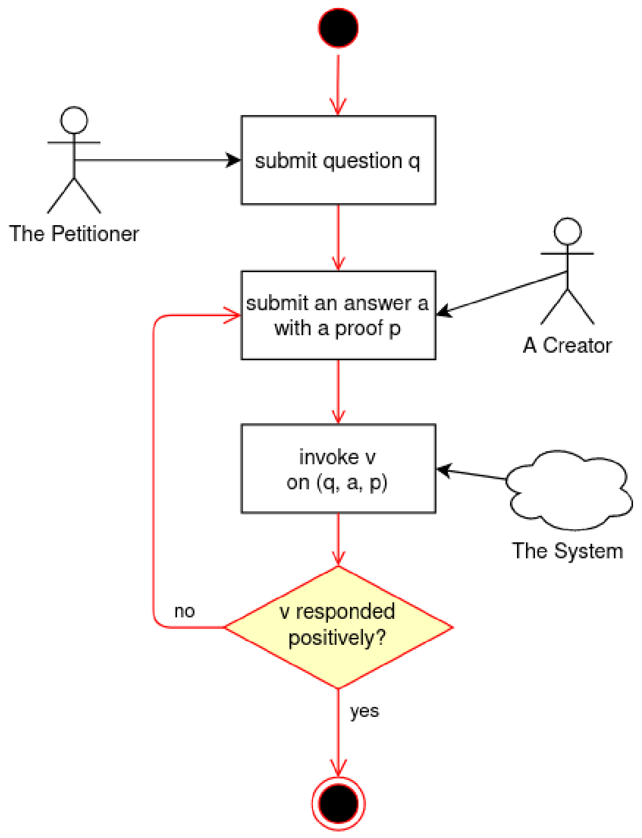

Figure 3.1: Flow of a competition

3.1.2 Competition flow

On a competition kind $K_c = ((Q_K, A_K), v)$, the main flow of a completion is as follows:

1. The **petitioner** submits the question q and starts the competition
2. A **creator** submits an answer a accompanied by a proof p
3. The **system** invokes the proof validator v on (q, a, p) and
   - if v responds positively the competition **concludes successfully**
   - otherwise, the system allows the re-execution of **step 2**

This flow is depicted in figure 3.1.

3.2 Algorithmic Problem Solving Competitions

For the remainder of this thesis, we focus on a single problem kind: the search for algorithms for computational problems, restricted by worst case time and space complexity constraints. More formally, let $K_A = (Q, A)$ be the problem kind under consideration where the elements of Q represent computational problems accompanied by complexity constraints and the elements of A represent solver algorithms for the elements of Q. In order for the generic competition framework of the previous section to be specialized for algorithmic problem solving competitions, we first define the format of problem definitions (objects in Q) and solver algorithms (objects in A) and then propose an appropriate proof validator mechanism. The proof validator mechanism we propose, validates the suitability of a solver algorithm against a problem definition with the help of a proof in Hoare logic.

This section is organized as follows: we first give some suitable definitions for computational problems in subsection 3.2.1, then we establish the format of solver algorithms in subsection 3.2.2, subsequently we establish the format of problem definitions in subsection 3.2.3 and, finally, we describe the proof validator mechanism in subsection 3.2.4.

3.2.1 Computational problems

3.2.1.1 Definition

Definition 3.9 (Computational Problem)**.** A computational problem is a binary relation $P \subseteq I_P \times S_P$ where I_P is the instance space of the problem and S_P is the solution space of the problem. The meaning of P is the following: for $i \in I_P$ and $s \in S_P$ if $(i, s) \in P$ then s is a correct solution for instance i. An $i \in I_P$ for which there is no $s \in S_P$ s.t. $(i, s) \in P$ is called a negative instance for P.

For example, the problem of finding the natural square root of a natural number is defined by the pairs $(0, 0), (1, 1), (4, 2), (9, 3), ...$ while natural numbers with no natural square root, such as 3 or 6, are negative instances for this problem. In this problem we have $I_P = S_P = \mathbb{N}_0$.

Decision problems are a problem class where the objective for any given instance is to decide whether that instance holds a specific property or not. We view decision problems as a special case of computational problems. Indeed, a decision problem can be viewed as a computational problem $P \subseteq I_P \times S_P$ consisting of all (i, s) pairs for which i holds the property and s always equals the boolean value $True$.

For example, the primality testing problem is defined for natural numbers > 1 by the pairs $(2, True)$, $(3, True)$, $(5, True)$, $(7, True)$, ... while composite numbers, such as 4 or 6, are negative instances for this problem. In this problem we have $I_P = \{i \in \mathbb{N} \mid i > 1\}$. Regarding the solution space, we may either consider $S_P = \{True\}$ or $S_P = \{True, False\}$.

<u>REMARK</u>: Other definitions for computational problems, such as the one in [Mac18], do not restrict the instances to a space I_P or the solutions to a space S_P but rather define computational problems over the space of all strings, $\Sigma*$. However, as we describe later in

this chapter, the solver algorithms in our platform are guarded against invalid instance representations and are guaranteed to be handled valid instances only. Furthermore, algorithms in our platform are written in a high level programming language which has data types such as integers and arrays, so the solutions returned by solver algorithms are bound to a data type and cannot be arbitrary strings. The consequences of this remark are addressed in detail in paragraph 3.2.3.3.

3.2.1.2 Extended format definition

In definition 3.9 we saw that a computational problem is not necessarily a total relation over it's instance space due to the existence of negative instances. As we desire to establish a standard behaviour over solver algorithms for handling negative instances, we *introduce* the extended format of a computational problem.

Definition 3.10 (Extended Format of Computational Problem)**.** For a given computational problem $P \subseteq I_P \times S_P$ we first define a set $NoSolution$ that is disjoint with S_P and the selection of it's contents can be arbitrary. Now, the extended format, denoted as P', shall be:

$$P' = \{(i, r) \,|\, (i, r) \in P \; or \; r \in NoSolution \; and \; i \; is \; negative\}$$

In other words, if N_P is the set of negative instance of P then:

$$P' = P \cup N_P \times NoSolution$$

For example, for the problem of the natural square root, if we select $NoSolution = \{-1\}$ we end up with the extended format consisting of $(0, 0), (1, 1), (2, -1), (3, -1), (4, 2)$....

As another example, for primality testing, if we select $NoSolution = \{False\}$ we end up with the extended format consisting of $(2, True)$, $(3, True)$, $(4, False)$, $(5, True)$, $(6, False)$, ...

3.2.2 Solver algorithms/Answer objects

In our platform, a correct solver algorithm for a problem $P \subseteq I_P \times S_P$ is required to compute the extended problem P' in order to handle negative instances. That is, a solver algorithm must implement a function $I_P \rightarrow S_P \cup NoSolution$ that maps every instance to either a correct solution $s \in S_P$ (i.e an s for which $(i, s) \in P$) or to an element of $NoSolution$ if i is a negative instance.

```
function solve(int n) -> int
  int r = 0
  result = -1
  while r < n
    if r*r == n
      result = r
      r = n
    else
      r = r+1
    end
  end
end
```

Program 3.1: Natural square root solver

For writing solver algorithms, we have designed the Ivee language which we fully describe in appendix A. The syntax of Ivee is quite straightforward and we believe the reader will easily comprehend Ivee programs without even consulting the appendix. We briefly note the following:

- Integer variables are declared with the `int` keyword and initialized with `0`
- Boolean variables are declared with the `boolean` keyword and initialized with `False`
- The special variable `result` denotes the result of a function

So, a solver is represented by an Ivee program which necessarily implements an appropriate function $solve\colon I_P \rightarrow S_P \cup NoSolution$. Program 3.1 is a correct solver for the natural square root problem when selecting $NoSolution = \{-1\}$ (i.e calculating the natural square root of a natural number or returning -1 if no such square root exists).

As we need the solver to be further processable for the purpose of proving it's correctness, the source code in plaintext is not a convenient format. We need a structured representation of the program's semantics. We obtain such a structured representation by parsing the program's source code, producing it's Abstract Syntax Tree (AST) and then enriching the AST with semantic information such data types. We refer to this representation as Semantic AST of a program. As Semantic ASTs can get lengthy, we provide a small example in figure 3.2 for program 3.2.

```
function add(int a, int b) -> int
  result = a + b
end
```

Program 3.2: Addition

With Semantic ASTs we have defined the format of the answer objects for algorithmic problem solving competitions. A creator wishing to submit an answer for a competition

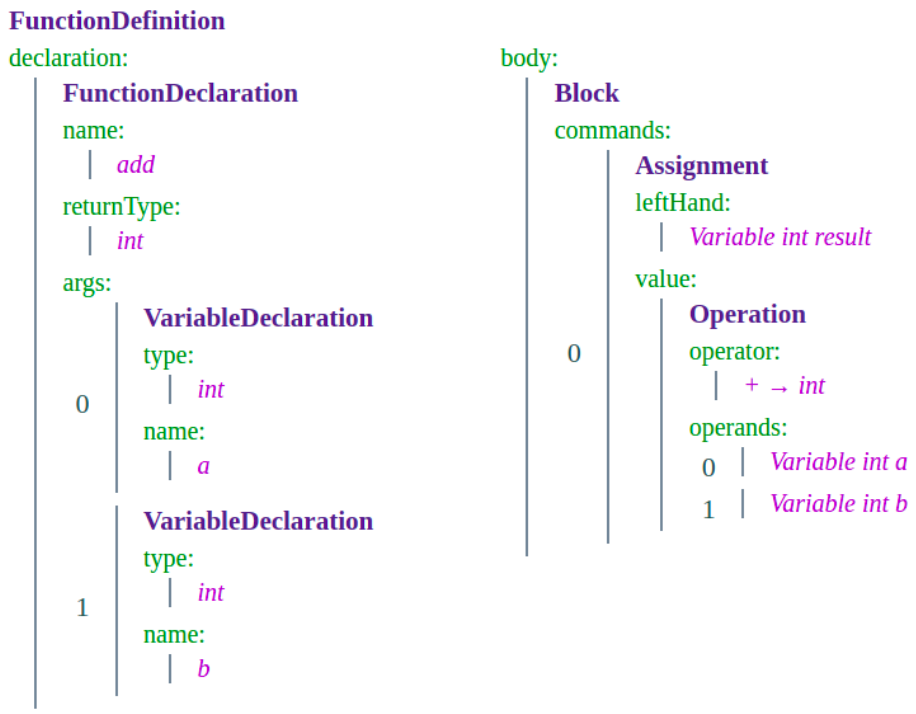

Figure 3.2: Semantic AST of program 3.2

will have to submit the Semantic AST of his solver. Section 4.1.2.2 presents the compiler we developed to transform the source source code of an Ivee program into it's Semantic AST.

3.2.3 Problem definitions/Question objects

The definition for computational problems we saw in subsection 3.2.1 regards a computational problem as a possibly *infinite* relation. This makes it infeasible to achieve a straightforward representation of a problem's definition and so the need for an alternative representation in the form of a finite object arises. The hard requirement for the format of such a problem definition object[2] will be to fully allow the determination of the problem's elements, that is, the instance-solution pairs. Furthermore, the format of this object must efficiently serve the purpose of validating candidate solver algorithms against it i.e. we must be able to use this object as a "judge" for candidate solvers. The *core idea* for "judging solvers" is the following: for any given instance i of a problem P, after passing the instance to a candidate solver and obtaining the solver's response s, use the problem definition object to *assert* that s is a correct solution for i or, if i is a negative instance, s is a negative response. To appropriately handle the case of negative instances, we shall be working with

[2]The term "problem specification" could be used as a synonym for "representation of the problem definition" in this context. However, we prefer the later as it makes the connection with previously introduced concepts clearer.

the extended problem format (see definition 3.10). We consider two approaches for the format of the problem definition objects presented below.

For a given computational problem P with extended format P', the problem definition object can either be:

1. a logical sentence satisfiable by all (i, s) pairs belonging to P' and only them
2. an algorithm that is able to determine whether a pair (i, s) belongs or not in P'

We promote the 2nd approach as easier to be implemented by petitioners for the following reasons:

- We empirically believe that implementing an algorithm is a skill far more common than composing logical sentences in a formal language
- The preceding point is further strengthened by the fact that our platform deals with algorithms so all participants should be familiar with algorithms
- Competition questions would be authored in the same language as competition answers thus reducing the knowledge requirements for petitioners

So, we focus exclusively on the 2nd option and propose a problem definition representation inspired by the notion of the verifier algorithm known from the study of NP-completeness.

3.2.3.1 Verifier-based problem definitions

The idea of an algorithm that takes as input a problem instance along with a candidate solution and decides whether the candidate solution is correct for that instance is well developed in the study of NP-completeness. This kind of algorithm is known as a verifier. Our platform utilizes the notion of the verifier as a means to formally define computational problems. However, as pointed out by MacCormick [Mac18], the largest part of the related literature focuses on verifiers for decision problems, contrary to the views of our platform which focuses on computational problems and regards decision problems as a special case. MacCormick [Mac18] proceeds to formulate a thorough and comprehensible definition for the verifier of a computational problem. We adopt their definition to our context and present it as definition 3.11. Definition 3.11 serves as a starting point in the process of introducing the format of problem definition objects (formally, the elements in Q) presented later in this chapter.

Definition 3.11 (Verifier for Computational Problems)**.** Let $P \subseteq I_P \times S_P$ be a computational problem. A *verifier* for P is a program $V(i, s, h)$ with the following properties:

- V receives three parameters: an instance $i \in I_P$, a candidate solution $s \in S_P$, and a hint h whose format is specific to V

- V halts on all inputs, returning either $True$ or $False$
- **Every non-negative instance can be verified:** If i is a non-negative instance, then $V(i, s, h) = True$ for *some* correct solution s and *some* hint h
- **Negative instances can never be verified:** If i is a negative instance, then $V(i, s, h) = False$ for *all* values of s and h
- **Incorrect proposed solutions can never be verified:** If s is not a correct solution (i.e $(i, s) \notin P$), then $V(i, s, h) = False$ for *all* h

In practice, a verifier V functions as follows:

$$V(i, s, h) = \begin{cases} True & \text{If } h \text{ provides sufficient evidence that } s \text{ is a correct response for } i \\ False & \text{Otherwise} \end{cases}$$

A verifier in this form does meet the hard requirement of fully allowing the determination of a problem's elements (i.e. instance-solution pairs). Indeed, the following statement connects membership in a problem P with the response of a verifier V for P:

$$(i, s) \in P \iff \textit{there exists } h\text{: } V(i, s, h) = True$$

However, this verifier form is not convenient for handing the case of negative instances because, as resulting from points 4 and 5 of definition 3.11, it is unable to distinguish a pair of a negative instance with a *correct* negative solver response from an incorrect proposed solution. In order to be able to use the verifier as an efficient "judge" for solver algorithms, it suffices to adopt the extended problem format which technically eliminates negative instances and effectively renders point 4 irrelevant. Indeed, the following statement connects membership in a problem P with the response of a verifier V' for the extended P' while also takes into account negative instances, where N_P is the set of negative instances and $NoSolution$ in an appropriate set for negative responses:

$$(i, s) \in P \textit{ or } (i, s) \in N_P \times NoSolution \iff \textit{there exists } h\text{: } V'(i, s, h) = True$$

We've now reached a point where in order to represent a computational problem $P \subseteq I_P \times S_P$ it suffices to follow these steps:

1. Select a set $NoSolution$, disjoint with S_P, in order to obtain P'
2. Write a program that implements the function $verify\colon I_P \times S_P \times H \to \{True, False\}$ which must function as verifier for P' where H is specific to the $verify$ function

Let us now take the problem of natural square roots of natural numbers as an example. We first select $NoSolution = \{-1\}$ in order to obtain the extended problem and then we write a verifier for the extended problem, presented as program 3.3. The function of this program is explained as follows: verification for a non-negative instance is trivial and does not require the hint parameter while verification for a negative instance takes advantage of the fact that if number i has no natural square root then there exists h such that $h \cdot h < i < (h+1) \cdot (h+1)$.

```
function verify(int i, int s, int h) -> boolean
  if s == -1
    result = h*h < i and (h+1)*(h+1) > i
  else
    result = s*s == i
  end
end
```

Program 3.3: Natural square root verifier

We proceed to evaluate the convenience of using a verifier for the purpose of validating the correctness of a candidate solver for a given computational problem.

Proposition 3.1 (Correctness against a verifier)**.** Let $P \subseteq I_P \times S_P$ be a computational problem, let *solve* be a candidate solver, let $verify$ be an appropriate verifier for the extended P'. In order to prove that *solve* is correct for P it suffices to show the following:

$$\text{For each } i \in I_P\text{: } solve(i) \text{ terminates and there exists } h \text{ s.t. } verify(i, solve(i), h) = True$$

The proving process resulting from proposition 3.1, although trivially correct, has an inherent incompatibility with the proof of correctness validation process presented in the next subsection 3.2.4. Specifically, it requires a hint object h to be supplied along each correct solution s or, otherwise, it requires a method to obtain such h. In, contrast to this requirement, the solver algorithm only calculates the solution. Enforcing the solver to calculate the hint is not desirable as this calculation will make the solver more complex without producing any information requested by the petitioner. To bypass this inconvenience, we introduce the *negated verifier* in the following paragraph.

3.2.3.2 Negated verifier-based problem definitions

Contrary to the “positive” verifier presented earlier, a negated one focuses on wrong answers: it uses the hint argument to prove that a proposed solution is *not correct*. Note that the definition of negated verifiers, which follows, handles negative instances by using the extended problem format, and so no explicit reference to negative instances is present.

Definition 3.12 (Negated Verifier)**.** Let $P \subseteq I_P \times S_P$ be a computational problem with extended format P'. A *negated verifier* for P is a program $NV(i, s, h)$ with the following properties:

- NV receives three parameters: an instance $i \in I_P$, a candidate solution $s \in S_P$, and a hint h whose format is specific to NV
- NV halts on all inputs, returning either $True$ or $False$
- **All *incorrect* solutions can be verified:** If $(i, s) \notin P'$, then $NV(i, s, h) = True$ for *some* h
- **Correct proposed solutions can never be verified:** If $(i, s) \in P'$, then $NV(i, s, h) = False$ for *all* h

To intuitively grasp the concept of the negated verifier, we can imagine that for a given solver there is a person, the "accuser", who is trying to prove that a the solver is not correct. To achieve this, it suffices to find an instance for which the solver returns a wrong solution and also provide a proof (hint) for their claim. A negated verifier is responsible to evaluate the validity of this "accusation".

More formally, the following statements connect membership and non-membership in a problem P with the response of a negated verifier NV for P, where N_P is the set of negative instances and $NoSolution$ in an appropriate set for negative responses:.

$$\begin{aligned} (i,s) \in P \; or \; (i,s) \in N_P \times NoSolution &\iff for\; all\; h\colon NV(i,s,h) = False \\ (i,s) \notin P \; and \; (i,s) \notin N_P \times NoSolution &\iff there\; exists\; h\colon NV(i,s,h) = True \end{aligned}$$

Let us again take the problem of natural square roots of natural numbers as an example. A negated verifier when $NoSolution = \{-1\}$ is presented as program 3.4. This program verifies the two possible kinds of errors a solver algorithm can make:

- The proposed solution is a yes-solution ($\neq -1$) but does not really equal the square root of the instance.
- The proposed solution is a no-solution ($= -1$) but the instance actually has a square root. In this case, the hint h must hold the actual square of the instance as a proof.

```
function negatedVerify(int i, int s, int h) -> boolean
  result = s == -1 and h*h == i or s != -1 and s*s != i
end
```

Program 3.4: Natural square root negated verifier

Table 3.1 gives a sample of responses for this negated verifier along with a brief explanation, where N_P is the set of negative instances of P.

i	s	h	Response	Explanation
4	2	0	False	$(i, s) \in P$
4	2	2	False	$(i, s) \in P$
5	2	0	*True*	$s \notin NoSolution, (i, s) \notin P$
5	-1	2	False	$s \in NoSolution,\ i \in N_P$
9	-1	2	False	$s \in NoSolution,\ i \notin N_P$ *and h not adequate*
9	-1	3	*True*	$s \in NoSolution,\ i \notin N_P$ *and h adequate*

Table 3.1: Sample responses of program 3.4

Having a negated verifier for a computational problem allows as to validate the correctness of a candidate solver.

Proposition 3.2 (Correctness against a negated verifier)**.** Let $P \subseteq I_P \times S_P$ be a computational problem, let *solve* be a candidate solver, let *negatedVerify* be an appropriate negated verifier for P. In order to prove that *solve* is correct for P it suffices to show the following:

For all $i \in I_P$ and h: $solve(i)$ terminates and $negatedVerify(i, solve(i), h) = False$

Proof. Proposition 3.2 produces proofs by contradiction for the correctness of solvers in the following manner. (We focus on problems where $I_P \neq \emptyset$ without affecting out arguments.) Let's assume that *solve* is not correct. There must be an $i \in I_P$ s.t. either $solve(i)$ does not terminate or $(i, solve(i)) \notin P'$. Consequently, if $solve(i)$ does terminate then $(i, solve(i)) \notin P'$ which in turn, from point 3 of the negated verifier definition (3.12), implies that there exists an h s.t. $negatedVerify(i, solve(i), h) = True$. So, if we proof that "for all $i \in I_P$ and h: $negatedVerify(i, solve(i), h) = False$ and $solve(i)$ terminates" we reach a contradiction. □

3.2.3.3 Data types and valid instances

Definition 3.9 for computational problems assumes that problem instances are drawn from a specific space I_P. The Ivee language we use for both problem definitions and solvers has data types such as booleans, integers, arrays etc. so in practice, problem instances are restricted to a specific data type. However, the intance space of a problem might not correspond exactly to an Ivee data type, and so a wider data type will have to be used. In the example of the natural square root (see programs 3.1, 3.4), we chose the `int` data type to represent problem instances but the actual instance space consists only of non-negative integers. As we are are only interested in the behaviour of solvers when handed valid instances, we establish a guard mechanism to ensure that invalid instances are filtered out. The guard takes the form of a function *validIntance*: `Ti` $\rightarrow \{True, False\}$ where `Ti` is the

```
function validInstance(int i) -> boolean
  result = i >= 0
end

function negatedVerify(int i, int s, int h) -> boolean
  result = s == -1 and h*h == i or s != -1 and s*s != i
end
```

Program 3.5: Natural square root problem definition with no complexity constraints

(possibly wider) data type used to represent instances. *validIntance* decides which elements of `Ti` belong in I_P by returning $True$ for them and only them (i.e. it is the characteristic function of I_P over `Ti`). An appropriate *validIntance* function must be provided in problem definitions along with the problem's negated verifier. Program 3.5 demonstrates both these functions and is able to fully define the Natural square root problem. (Note, that there is not corresponding "*validSolution*" function, even if the data type representing solutions is wider than S_P, as the negated verifier is fully responsible to check the validity of proposed solutions.)

3.2.3.4 Complexity constraints

The last issue that remains to be addressed is the specification of constraints for time and space complexity requirements. We address it by introducing three additional functions to the problem definition object, presented below.

- $size\colon I_P \to \mathbb{N}$ which receives an instance and returns it's size as a single parameter
- $maxSteps\colon \mathbb{N} \to \mathbb{N}$ which receives the size of an instance, as returned by *size*, and returns the maximum number of steps a solver is allowed to execute for instances of that size
- $maxSpace\colon \mathbb{N} \to \mathbb{N}$ which receives the size of an instance, as returned by *size*, and returns the maximum memory space a solver is allowed to utilize for instances of that size

We once again return to the example of the natural square root problem and demonstrate a full problem definition with complexity constraints in program 3.6. (Note that the size of an instance in that example is regarded to be the absolute value of the intense and not the number of it's bits.)

We will be using these functions to validate the worst case asymptotic complexity of candidate solvers (big-O) in a way outlined by propositions 3.3 and 3.4.

Proposition 3.3 (Time Complexity Bounds against `maxSteps`)**.** Given functions $size\colon I_P \to \mathbb{N}$, $maxSteps\colon \mathbb{N} \to \mathbb{N}$ and a method *stepcount* that counts the number of steps of a

```
function validInstance(int i) -> boolean
  result = i >= 0
end

function size(int i) -> int
  result = i
end

function maxSteps(int siz) -> int
  result = siz
end

function maxSpace(int siz) -> int
  result = 1
end

function negatedVerify(int i, int s, int h) -> boolean
  result = s == -1 and h*h == i or s != -1 and s*s != i
end
```

Program 3.6: Natural square root problem definition

computation, in order to prove that the worst case time complexity of a solver *solve* belongs to the class $O(maxSteps(size(i)))$ where $i \in I_P$, it suffices to show the following:

$$\text{There exist } n_0, c \text{ s.t. for all } i \in I_P\text{:}$$
$$size(i) < n_0 \text{ or } stepcount(solve(i)) \leq c \cdot maxSteps(size(i))$$

Proposition 3.4 (Space Complexity Bounds against `maxSpace`)**.** Given functions $size\colon I_P \to \mathbb{N}$, $maxSpace\colon \mathbb{N} \to \mathbb{N}$ and a method *spacecount* that counts space utilized in a computation, in order to prove that the worst case space complexity of a solver *solve* belongs to the class $O(maxSpace(size(i)))$ where $i \in I_P$, it suffices to show the following:

$$\text{There exist } n_0, c \text{ s.t. for all } i \in I_P\text{:}$$
$$size(i) < n_0 \text{ or } spacecount(solve(i)) \leq c \cdot maxSpace(size(i))$$

Both propositions come from the definition of asymptotic worst case complexity. Methods *stepcount* and *spacecount* will be discussed in subsection 3.2.4.3.

3.2.3.5 Complete problem definition

We have placed both the representation of computational problems and the representation of complexity constraints for an algorithmic problem in the form of a program. Of course, the actual problem definition object will have the form of the Semantic AST of such program rather than it's source code for the reasons explained in subsection 3.2.2. Wrapping this section up, we present the form the problem definitions (or in terms of the generic abstract framework of section , the question objects/elements in Q) in definition 3.13.

Definition 3.13 (Representation of Algorithmic Problem Definitions)**.** Given an algorithmic problem, that is a computational problem $P \subseteq I_P \times S_P$ accompanied by a pair of time and space complexity constraints and an implied set $NoSolution$ if P is not total over I_P, the representation of the problem's definition (i.e. the question object) will be the Semantic AST of a program with following functions:

- `validIntance(Ti) -> boolean`
- `size(Ti) -> int`
- `maxSteps(int) -> int`
- `maxSpace(int) -> int`
- `negatedVerify(Ti, Si, Hi) -> boolean`

Where `Ti` is a data type able to represent the elements of I_P, `Si` is a data type able to represent the elements of $S_P \cup NoSolution$ and `Hi` is a data type able to represent the hint argument that serves the function `negatedVerify`.

3.2.4 Proof validation through program verification

In this subsection we propose a method for validating a proof for a) the correctness b) the compliance with time complexity constraints c) the compliance with space complexity constraints of a candidate solver in accordance with a problem definition object (see definition 3.13 for problem definition objects). The validation process is performed on a proof object, distinct from the solver, and takes the form of formal program verification.

3.2.4.1 Formal program verification

Formal program verification is the process of ensuring that a program satisfies some requirements by using formal methods of mathematics. One of the most famous methods of formal program verification is Hoare logic [Hoa69]. For a given program `C`, Hoare logic is capable of proving that a set of properties, called postconditions, holds when `C` finishes execution. A proof that asserts the desired postconditions might be conditioned by a set of properties, called preconditions, that must hold before `C` executes. Figure 3.3 demonstrates these principles. The main entity in Hoare logic is the Hoare triple which takes the form `{P}C{Q}` where `P` is a precondition, `C` is a program (or a command, or a block of commands) and `Q` is a postdonction. Triple `{P}C{Q}` means that if `P` holds at the start of the execution of `C` then `Q` necessarily holds of the end of the execution. Preconditions and postconditions are logical statements that express truths about the variables of `C` and their relations. Additionally, a standard extension of Hoare logic presented in various sources, such as [NK14], is able to proof that a given program terminates. Finally, as it is formally established by Nielson [Nie84], extensions of Hoare logic are able to prove asymptotic upper bounds for the time

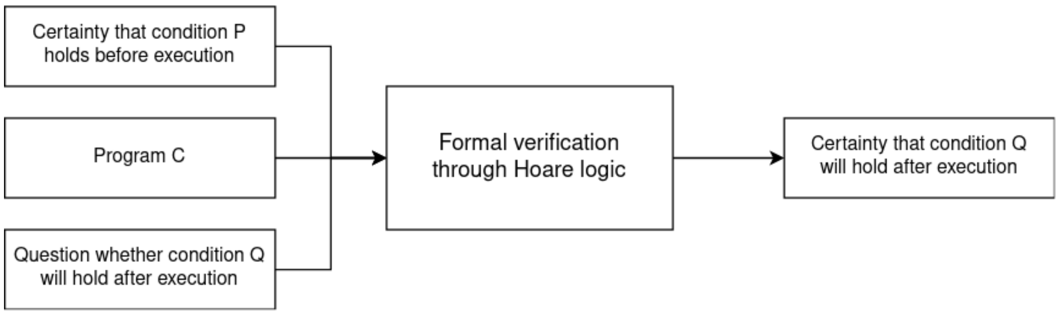
Certainty that condition P holds before execution
Program C
Question whether condition Q will hold after execution
Formal verification through Hoare logic
Certainty that condition Q will hold after execution

Figure 3.3: Simplified view of Hoare logic

complexity of programs. In the remainder of this section, we propose a proof validator for algorithmic problem solving competitions that is able to validate Hoare logic proofs which prove both the correctness and the upper bounds of the complexity of a solver algorithm in accordance with a problem definition.

3.2.4.2 Correctness

By examining the concepts of the solver algorithm and the negated varifier, we observe that the following "game" takes place among them: a solver proposes solutions while a negated verifier checks them. By this observation and in conjunction with proposition 3.2, we can prove the correctness of a solver by using formal program verification to prove a special postcondition on a a program that appropriately combines the problem definition program and the solver program. Specifically, this combination results in the program `correctness` (program 3.7) and the desired postcondition is that variable `verdict` always equals `True` when `correctness` finishes execution. Of course, it must also be proved that `correctness` terminates for every input which will in turn prove that the solver terminates for every valid instance (assuming a well defined negated verifier). By proving both the postcondition `verdict=True` and the termination of `correctness` we prove the statement of proposition 3.2 and thus the correctness of the solver under consideration.

```
function correctness(Ti i, Th h)
  boolean verdict
  if validInstance(i)
    Ts s = solve(i)
    verdict = not negatedVerify(i, s, h)
  else
    verdict = True
  end
end
```

Program 3.7: Format of program `correctness`

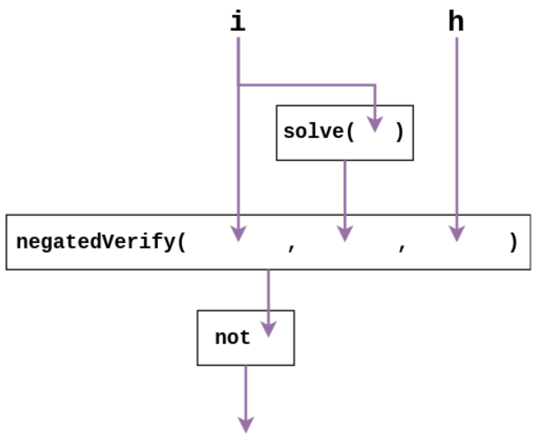

Figure 3.4: Computation flow of the correctness program for valid instances

Observe in program 3.7 that the condition `if validInstance(i)` guards against invalid instances been passed to the solver or the negated verifier as we are only interested on the behaviour of the solver for valid instances. Figure 3.4 demonstrated the computation flow in the case of a valid intance `i`.

REMARK: We assume that the functions `validInstance` and `solve` do not mutate their input. This assumption always holds when `Ti` is an immutable type such `boolean` or `int` but may not hold if `Ti` is, for example, an array type which can be modified in place. To handle a possible mutation of `i`, it suffices to create a deep copy before passing it to functions `validInstance` and `solve`.

3.2.4.3 Complexity

We work in similar fashion for the proofs of time and space complexity bounds. Specifically, for time complexity, we rely on the "method of program counters" as formally described by Nielson [Nie84]. According to this method, in order to reason about upper bounds for the time complexity of a program we first modifying that program by inserting a the command `time = time + 1` after *every* command of the original program. The variable `time` is initialized with 0 and effectively counts the number of steps of the computations performed by the program. We can then prove upper bounds for the time complexity of the program by asserting an appropriate postcondition regarding the value of `time` with the use of standard formal program verification. The method of programmed counters not only is proven to be sound by Nielson [Nie84][3] but also, contrary to other methods, has the advantage of allowing reasoning about the relation of the `time` special variable with the normal variables of the original programs. This property is useful as we need to compare the number of steps to the upper bounds returned by the `maxSteps` of a problem definition.

[3]It is worth noting that Nielson [Nie84] considers this method unnatural and proposes alternatives. However, she proves that the alternatives are not more powerful than this method

```
function timeComplexity(Ti i)
  boolean verdict
  if validInstance(i)
    int s = size(i)
    int max = maxTime(s)
    int cost = solveTime(i)
    verdict = s < N0 or cost <= C*m
  else
    verdict = True
  end
end
```

Program 3.8: Format of program `timeComplexity`

```
function solveTime(int n) -> int
  int _old_result
  int time = 0
  int r = 0
  time = time + 1
  _old_result = -1
  time = time + 1
  while r < n
    time = time + 1
    if r*r == n
      time = time + 1
      _old_result = r
      time = time + 1
      r = n
      time = time + 1
    else
      time = time + 1
      r = r+1
      time = time + 1
    end
  end
  result = time
end
```

Program 3.9: `solveTime` program corresponding to solver 3.1

In order to use the method of programmed counters, we appropriately combine the candidate solver and the problem definition into the program `timeComplexity`. We first modify the `solve` function of the candidate solver according to the method of programmed counters to obtain the function `solveTime` (see program 3.9 for an example). The `solveTime` returns the value of the special variable `time` instead of the actual solution. The format of the program `timeComplexity` is presented in program 3.8.

Note that the identifiers `N0` and `C` are mere placeholders: they must be replaced by positive values before proceeding to a proof. The desired posticondition is again `verdict=True`. A proof of the postcondition is acceptable for any `N0` ≥ 0 and `C` > 0. The termination of the program `timeComplexity` needs not be proved, it is implied by the termination of the program `correctness`.

Operation	Code example	Cost
new record object	`new Date(2020, 12, 23)`	1
new array of length n	`new int[]{n}`	n
new map of length n	`new map(int -> int){n}`	n

Table 3.2: Memory allocation costs for the Ivee language

The process of proving upper bounds for the space complexity of solvers is almost identical to the one for time complexity. The only difference is, instead of increasing a `time` variable by 1 after every command, we increase a `space` variable after every command that *allocates* memory by the amount of the newly allocated memory. Table 3.2 presents the memory allocation costs for the operations of the Ivee language, after ignoring constant factors. We obtain the program `spaceComplexity` in a completely analogous manner as with `timeComplexity`. The function `solveSpace` comes from the original `solve` function after modifying it to return the counted space usage instead of the actual solution. The format of `spaceComplexity` is presented in program 3.10.

```
function spaceComplexity(Ti i)
  boolean verdict
  if validInstance(i)
    int s = size(i)
    int max = maxTime(s)
    int cost = solveSpace(i)
    verdict = s < N0 or cost <= C*m
  else
    verdict = True
  end
end
```

Program 3.10: Format of program `spaceComplexity`

The postocondition is the same as in the case of time complexity.

REMARK: This process does not take into account memory *de-allocation* as Ivee does not currently support it. So, the memory cost may only increase during the execution of a solver. A way to avoid unnecessary memory costs would be to reuse existing previously allocated objects when possible.

3.2.4.4 Proof objects

As Hoare logic is based on deductive reasoning, a proof of correctness would consist of a series of logical steps, each resulting from the deduction schemes of Hoare or classical logic, ending at the desired postcondition. Since the proof for correctness, time complexity and space complexity are independent, a proof object must contain three different logical step sequences. Finally, a proof object must provide the values of `N0` and `C` selected for the proofs of complexity. Note that a proof object needs not contain the programs `correctness`, `timeComplexity` or `spaceComplexity` as they can be computed by the problem definition and the solver algorithm.

For more details on proof objects, consult appendix B, Hoare Logic Adaption.

3.2.4.5 Proof validator properties

We are now in position to introduce a proof validator for algorithmic problem solving competitions in accordance with the abstract proof validator concept we saw in 3.1.1.2. The proof validator will accept three parameters: a problem definition q, a solver algorithm a and a proof object p in the form described in this section. The proof validator will then validate the logical steps of the proof object to determine weather they lead to the desired postconditions by means of logical deductions.

This proof validator is **pure** (see def. 3.5), as it only takes into account the parameters q, a and p.

This proof validator is **sound** (see def. 3.3), as Hoare logic is a sound formal system as demonstrated by Cook [Coo78]. Regarding proofs for complexity bounds, the method of programmed counters, used intact for time complexity proofs and slightly adjusted for space complexity proofs, is proven to be sound by Nielson [Nie84].

This proof validator is *relatively* **complete** (see def. 3.4) in a sense that Hoare logic is relatively complete as suggested by Cook [Coo78]. In particular, completeness of any axiomatic proof system (such as Hoare logic) is prohibited by Gödel's Incompleteness Theorem [Göd31]. In practice, the completeness of Hoare logic, and by extension the completeness of our validator is limited by insurmountable restrictions in Logic itself. For a detailed presentation of the subject, the reader may consult [Win93].

3.2.4.6 Hoare logic for the Ivee language

The original Hoare logic by Hoare [Hoa69] was proposed for a language with restricted capabilities. Specifically, programs of that language could only have a single procedure with only integer variables. In the years following the original publication, multiple authors proposed extensions for handling more advanced program elements. We shall enumerate a set that is adequate for supporting the capabilities of the Ivee language. Cook [Coo78] handles global variables and sub-procedures, America and Boer [AB90] handle recursive procedures, Bornat [Bor00] handles pointers and by extension records, while multiple authors such as Nipkow and Klein [NK14] handle arrays and lists with a logic that can also be applied to mappings.

4 System Design and Implementation

This chapter presents a system design according the theoretical setup of the previous chapter. The system allows the conduction of competitions for algorithmic problems and comprises of a web-based front-end and a purely blockchain-based back-end.

This chapter first presents an overview of the system followed, by a per-component analysis and then proceeds to highlight some implementation aspects.

4.1 Design

4.1.1 Overview

The system is a realization of the the setup described in chapter 3. The main use cases of the system are the following:

- Compiling a problem definition
- Starting a competition for a problem
- Compiling a solver algorithm
- Composing a proof of correctness/complexity for a solver algorithm against a problem definition
- Submitting a solver algorithm for a problem

These use cases are served by the front end components and the smart contracts of the system. Table 4.1 presents how the concepts introduced theoretically in chapter 3 are mapped into smart contracts.

The system follows a component-based design approach. The system operates in two tiers: the web tier responsible for the front end interface and the blockhain tier responsible for the back end logic.

Abstract Framework (3.1)	Algorithmic Setup (3.2)	Smart Contract
Competition flow		Competition Manager
Question objects	Algorithmic problem definitions	Program Store
Answer objects	Solver algorithms	Program Store
Proof validator	Proof validation via program verification	Proof Validator

Table 4.1: Concept mapping

The system's components in both these tiers are the following:

Competition Manager	(*blockchain*) is the central component of the system
Compiler	(*web*) compiles programs (problem definitions and solvers) from source code to low level structured objects
Program Store	(*blockchain*) validates and stores program objects
Proof Composer	(*web*) assists the user to create proofs of correctness and encodes them into proof objects
Proof Validator	(*blockchain*) validates proofs of correctness

The main processes that take place place in the system is the starting of competitions and the submission of solvers.

Competition starting is depicted in figure 4.1. The process is originated by a petitioner (user) and utilizes the front end (Compiler), the Program Store and the Competition Manager. In more detail, the competition starting process has the following steps:

1. The petitioner writes the Problem Definition in the Ivee language
2. The Compiler compiles the Problem Definition into a Program Object
3. The Program Object is submitted to the Program Store
4. The Program Store validates the Program Object, stores it and produces a Question ID
5. The Question ID is reported to the petitioner
6. The petitioner starts the competition by submitting the Question ID and, optionally, pledges a reward
7. The Competition Manager invokes the Program Store to verify that a Program Object with the given Question ID has been submitted and validated
8. The Program Store verifies the presence, and thus the validity, of the Program Object
9. The Competition Manager creates a record with the details of the competition

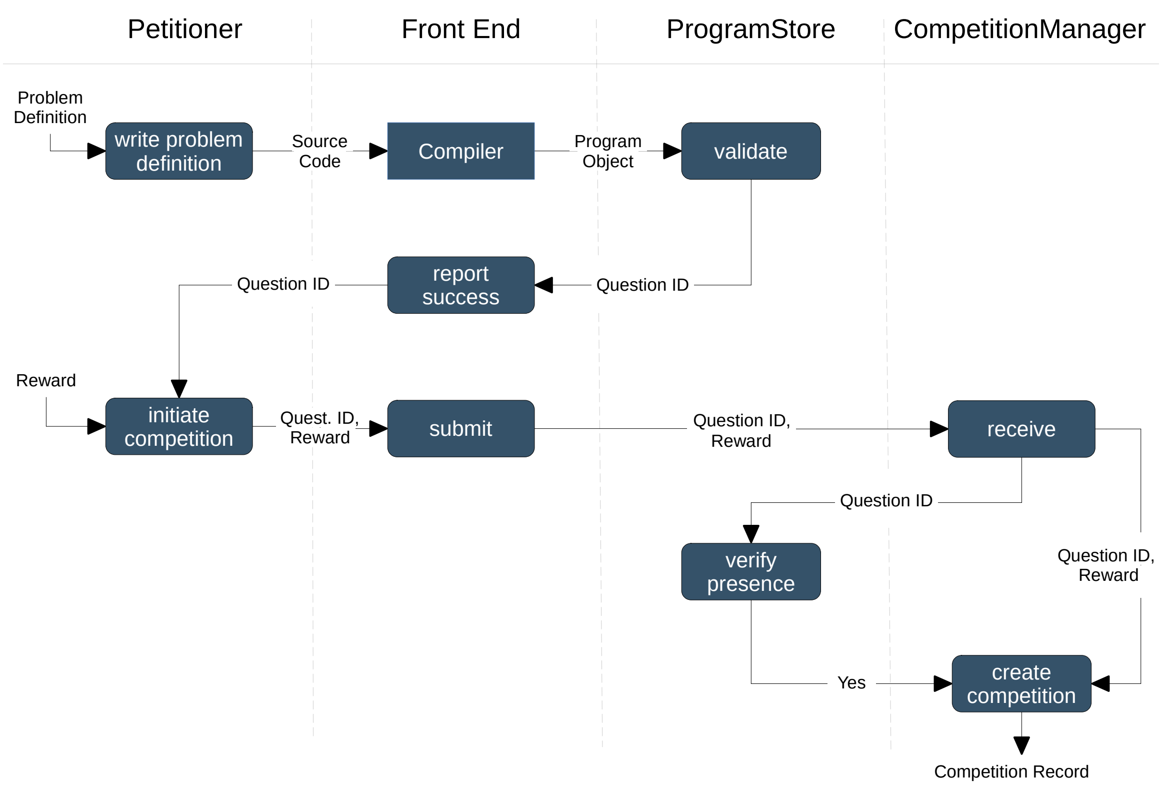

Figure 4.1: Competition starting process

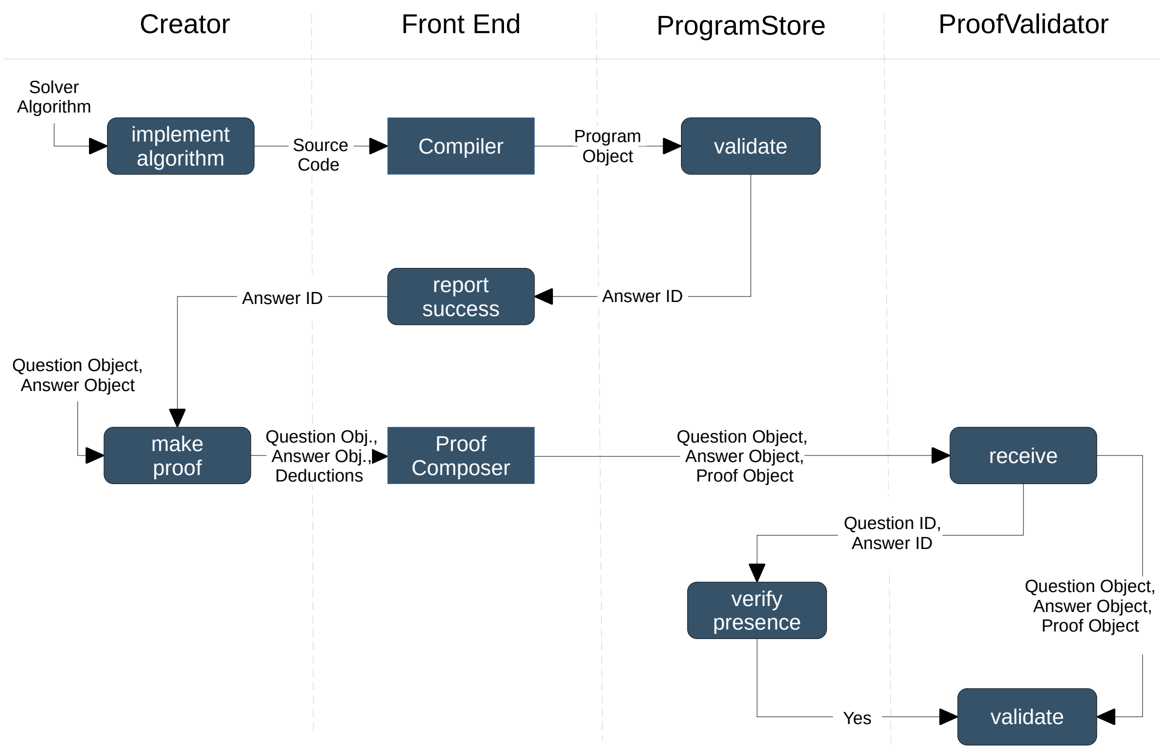

Figure 4.2: Answer submission process - One final step is omitted for simplicity: the Competition Manager closes the competition

Answer submission is depicted in figure 4.2. The process is originated by a creator (user) and utilizes the front end (Compiler and Proof Composer), the Program Store, the Proof Validator and the Competition Manager. In more detail, the answer submission process has the following steps:

1. The Creator writes the Solver Algorithm in the Ivee language
2. The Compiler compiles the Solver Algorithm into a Program Object
3. The Program Object is submitted to the Program Store
4. The Program Store validates the Program Object, stores it and produces an Answer ID
5. The Answer ID is reported to the Creator
6. The Creator makes the Deductions for a proof of correctness with the help of the Proof Composer
7. The Proof Composer encodes these deductions into a Proof Object and submits them to the Proof Validator along with the Question Object (the Program Object representing the Problem Definition) and the Answer Object (the Program Object representing the Solver Algorithm)
8. The Proof Validator re-calculates the Question ID and the Answer ID and then invokes the Program Store to verify that these programs have been submitted and validated
9. The Proof Validator validates the Proof Object
10. The Competition Manager closes the competition (not depicted)

4.1.2 Components

4.1.2.1 Competition Manager

The Competition Manager is the central component of the system, has the form of an Ethereum smart contract and is responsible for the conducting of competitions according to the flow described in 3.1.2.

The Competition Manager is agnostic of the format of question, answer and proof objects. When a competition starts, the question and answer stores (currently unified by the Program Store) and the proof validator for that competition must be specified. This happens *independently for each competition*, thus different competitions are free to use different question/answer stores and proof validators. This allows for future extensions and enhancements of the platform to be developed independently, without the need to revise the Competition Manager.

The Competition Manager enforces an interface for smart contracts that play the role of question/answer stores and proof validators.

Specifically, question/answer stores must implement the following function:

- `retrieveMeta(objectID) -> MetaRecord` which receives an object id and returns metadata about the object with that id. If not such object was submitted and validated, the function returns an empty response.

Proof validators must implement the following function:

- `checkProof(questionID, answerID) -> MetaRecord` which receives a question id/answer id pair and returns metadata about a *previously submitted, successfully validated* proof regarding those objects. If not such proof was submitted and validated, the function returns an empty response.

The `MetaRecord` objects must at least contain information about the owner of the corresponding object.

Finally, the Competition Manager itself has the following functions:

- `payable startCompetition(questionID, questionStoreAddress, answerStoreAddress, proofValidatorAddress) -> competitionID` which starts a competition with the specified question and configuration after first verifying that the question is present in the question store by invoking it's `retrieveMeta` function. Any amount paid to that function will be the competition's reward.
- `checkProof(competitionID, answerID, proofID)` which invokes the `checkProof` function of the competition's proof validator to check whether the specified proof has been validated for the specified answer against the question of the competition. If the proof validator responds positively, the competition closes and the owner of the proof is paid the reward.

This thesis presents a single question/answer store and a single, experimental, proof validator. However, this design promotes the extensibility and modularity of the system.

4.1.2.2 Compiler

The Compiler compiles programs written in the Ivee language which is used for both solver algorithms and problem definitions. The compiler's function is depicted in figure 4.3. In the first stage, the source code is processed by the parser which produces a Semantic AST, such as the one depicted in figure 3.2. This Semantic AST is passed to the Indexer which assigns a unique code to every node. Indexing also results to the deduplication of expressions as identical expressions are mapped to the same code even they occur multiple times. (Table

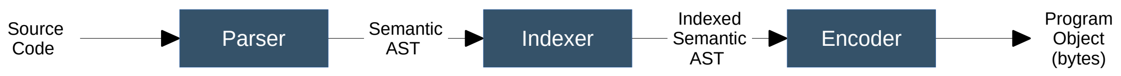

Figure 4.3: Function of the Compiler

Code	Expression	Production
1	`result`	Literal
2	`1`	Variable
3	`result + 1`	+(1, 2)
4	`(result + 1) * (result + 1)`	*(3, 3)
5	`n`	Variable
6	`(result + 1) * (result + 1) <= n`	<=(4, 5)

Table 4.2: Expression indexing and deduplication for program 5.2

4.2 demonstrates expression indexing and deduplication for program 5.2.) Finally, this Indexed Semantic AST is passed to the Encoder which produces a low level, structured representation suitable to be submitted as transaction data to smart contracts.

4.1.2.3 Program Store

The Program Store is a responsible for validating and storing program objects. These program objects are generated by the Compiler and represent both solver algorithms and problem definitions. In essence, the Program Store validates that a given program object has been supplied by a well behaving compiler as a means to prevent fraud.

4.1.2.4 Proof Composer

The Proof Composer is a front end, web-based, component responsible to assists the user to create proofs of correctness and complexity and to encode them into proof objects. The function of the Proof Composer is depicted in figure 4.4 for proofs of correctness (the behaviour is similar for proofs of complexity). The Composer consists of the Combiner, the Proof Assistant and the Encoder.

The Combiner takes the low level structured representations of the the problem definition (Question Object) and the solver algorithm (Answer Object) and combines them in a single program as described in paragraph 3.2.4.2.

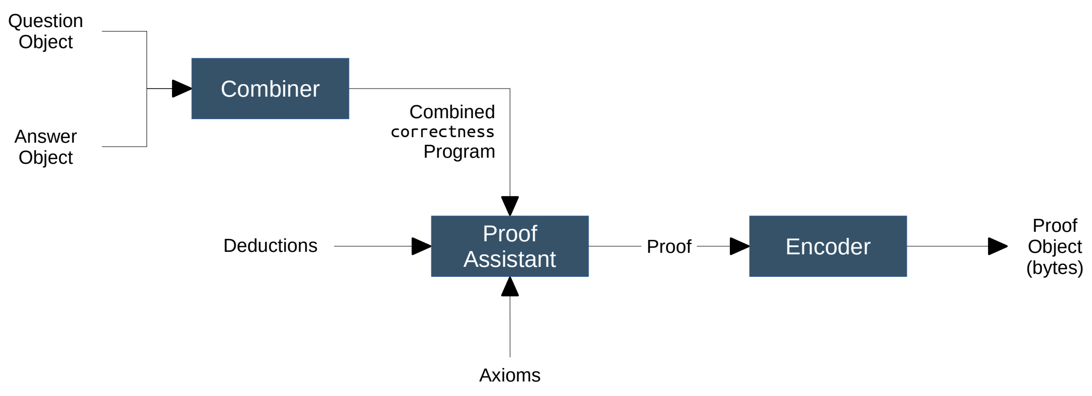

Figure 4.4: Function of the Proof Composer for correctness

Then, the assistant helps the user to make valid deductions in Hoare and classical logic, aiming to prove the target postcondition. For more information on the format of those those deductions, the reader may consult appendix B.

The Encoder, finally, produces a low level structured representation of the proof.

4.1.2.5 Proof Validator

The Proof Validator is a smart contract responsible for validating proof objects that are generated by the Proof Composer. A proof object is passed to the Proof Validator along with the related solver algorithm and problem definition. The Proof Validator must check that all deductions are valid and lead to the target postconditions.

4.2 Implementation

4.2.1 Features

Our proof of concept implementation consists of:

- A fully functional Competition Manager
- A fully featured compiler for the Ivee language
- A Program Store which implements the greatest part of the essential checks
- A Proof Composer for *correctness*, complexity proofs are not implemented, which
  - is limited to programs with only integer and boolean variables, do not procedures and do no have global variables
  - supports the full set of deductions presented in appendix B

    – disallows invalid deductions

- A Proof of Concept Proof Validator (PoCPV) for correctness which inherits the limitations of the Proof Composer implementation. PoCPV is limited to deductions of the following types:
    – Modus Ponens: classical form
    – Specification: specify a ∀ variable a to a given expression
    – Irrelevant Assignment: assert Hoare triples of the form `{P}x=E{P}` where the variable `x` does not occur in precondition *P*

  These types correspond to half the deduction steps in our experimental proof. Deductions of other types are passed though without checks.

4.2.2 Optimizations

The optimization effort focused entirely on reducing memory and storage consumption in smart contract code in order to reduce gas consumption. The most effective measure was to not store program objects and proof objects in the storage space of Ethereum but, instead, use their hash to check that a given object has been submitted and validated. (This hash is also used as ID for those objects.) As program objects need to be easily accessible by the Proof Composer, they are stored as events in the logs space of Ethereum, which is cheaper than the storage space. Proof objects persist only in transaction data.

An effort was also made to reduce the size of the transaction data. Specifically, both the Compiler and the Proof Composer encode objects by the use of *bytes* sequences and avoid the use of higher level data types. For example, in program objects, 7 bytes are used to encode an expression while 6 bytes are used to encode a command. Furthermore, the deduplication of expressions performed by the Indexed helps reduce the total number of expressions.

Finally, it must be noted that no effort has been made to optimize the time complexity of the code, although all executed algorithms are of constant or linear time.

5 Evaluation

This chapter presents the results of “running” the system for a sample of programs and a proof of correctness. The evaluation solely examines the blockchain tier (i.e. the Smart Contracts). Measurements of gas costs are obtained for various versions of the evaluated smart contracts.

The system is evaluated in regards to whether the gas costs stay within the block gas limit, the ease of use and, finally, the feasibility for real world use cases.

5.1 Experimental Setup

5.1.1 Evaluation samples

5.1.1.1 Programs

We evaluate the Program Store component in terms of gas cost by submitting the following programs:

- Small addition program 3.2
- Natural square root problem definition 3.6
- Simplified natural square root problem definition 5.1
- Natural square root “straight” verifier 3.3
- Natural square root problem solver 3.1
- Simplified natural square root solver 5.2

Where the simplified natural square root problem is described as follows: Given a natural number i find a natural number s s.t:

$$s \cdot s \leq i < (s+1)(s+1)$$

The difference between the natural square root problem and the simplified version is that the simplified version does not require a negative response for negative instances, it rather

```
function validInstance(int i) -> boolean
  result = i >= 0
end

function size(int i) -> int
  result = i
end

function negatedVerify(int i, int s, int h) -> boolean
  result = s*s <=i and (s+1)*(s+1) > i
end
```

Program 5.1: Simplified natural square root definition

```
function solve(int n) -> int
  while (result+1)*(result+1) <= n
    result = result + 1
  end
end
```

Program 5.2: Simplified natural square root solver

accepts the floor of the real valued square root of the instance. The full problem definition is given as program 5.1 and the corresponding solver is given as program 5.2.

5.1.1.2 Proof

We evaluate the PoCPV component by developing and submitting a proof for the correctness of solver 5.2 against definition 5.1. The main ideas behind the proof are based in Galm et al. [Gal+21] however, the full proof, in the format accepted by our platform, requires much more detail. Specifically, it takes up 105 deduction steps and is fully listed in appendix C. We do not proceed to proofs of time and space complexities as the PoCPV does not support them. However, the logic behind proofs of correctness in our platform is quite similar to the one for proof of complexity so we expect similar results.

5.1.2 Used tools

We use Ganache[1] version 2.5.4 as a local Ehtereum environment configured on the Muir Glacier hard fork. We use Remix[2] to deploy smart contracts. Solidity compiler version is 0.8.7 while our code requires version 0.8.6. Finally, we set the compilation optimization parameter to 200.

[1]https://www.trufflesuite.com/ganache
[2]https://remix.ethereum.org/

5.1.3 Process

We first deploy our smart contracts on Ganache and then invoke them by our web front end using web3.js. We perform multiple deployments in order to evaluate different versions of the smart contracts. We use the gas cost measures reported by Ganache. In order to determine the size of submitted objects we inspect those objects in the browser's JavaScript console.

5.1.4 Measurements

For each sample, we first determine it's size according to various size measurements and then submit it to the smart contract under evaluation to measure gas cost.

For program samples, we measure the following sizes:

- Number of expressions (exprs)
- Number of commands (comds)
- Number of functions (funs)
- Number of int literals (ints)
- Number of int strings (which come from identifier names) (strings)

We evaluate the Program Store in 4 different incremental versions:

1. Only receives the transaction data without doing any processing (Empty)
2. Only calculates the program ID without doing any validation (ID)
3. Performs all validations but does not issue an event with the Program Object (No Event)
4. Full version (Full)

For the proof samples, we measure the following sizes:

- Number of of used axioms (axioms)
- Number of generated preconditions (preconds)
- Number of deduction steps (steps)
- Number of deduction steps being fully validated by PoCPV (valsteps)
- Number of logical sentences (sents)
- Number of expressions (exprs)

We evaluate the PoCPV in 3 different incremental versions:

	Size						Cost			
Prog	exprs	comds	blocks	funs	ints	strings	Empty	ID	No Event	Full
3.2	4	1	1	1	1	4	28,118	34,832	176,113	194,076
5.2	6	2	2	1	1	2	28,518	35,332	205,862	224,430
3.1	10	7	4	1	2	4	30,454	38,694	227,863	250,604
3.3	15	3	3	1	1	5	30,694	38,858	242,829	265,492
5.1	20	5	5	5	2	9	36,562	49,257	282,837	318,262
3.6	25	5	5	5	2	9	36,954	49,662	298,693	334,388

Table 5.1: Gas cost for submitting programs

Size						Cost		
axioms	preconds	steps	valsteps	sents	expr	Empty	ID	Full
27	7	105	53	185	79	85,066	133,341	2,002,838

Table 5.2: Gas cost for validating the proof of appendix C

1. Only receives the transaction data without doing any processing (Empty)
2. Only calculates the question, answer and proof IDs without doing any validation (ID)
3. Full version (Full)

5.2 System evaluation

5.2.1 Cost evaluation

The measured gas costs are compared to the block gas limit which, at the time of writing, is 15,000,000 [MyC21].

The results for the evaluation of the Program Store are displayed in table 5.1. In the *No Event* and *Full* versions, a gas amount of 108,655 is fixed and associated with storing the program's metadata. These results indicate that the gas costs for a Program Store are in relatively safe levels and that a quite complicated algorithm would be needed to reach the block gas limit.

The results for the evaluation of the PoCPV are displayed in table 5.2. In the *Full* version a gas amount of 122,044 is fixed and associated with storing the proof's metadata. These results indicate that the gas costs for a Proof Validator are not safe regarding the block gas limit. We arrive to this conclusion because even a simple proof reaches quite high gas values, without even fully validating all steps. It is certain that, unless considerable optimizations are implemented on the Proof Validator, the validation of moderately complex proofs would need to break into multiple transactions. Thankfully this task is quite straightforward due to the deductive nature of proofs. Indeed, since deduction steps can only rely on previous steps, a single transaction may validate a number of steps and store a counter for the next transaction to pick up. This proposition relies on the fact that the *Empty* version uses insignificant gas and so re-submitting the whole proof object in each transaction is cheap.

5.2.2 Ease of use

A notable characteristic of the proof sample is it's length. Although it proves a simple program, the required deduction steps reached 105. This might be an insignificant size for a computer, but for human operators it make the process difficult and tedious.

Worth discussing is the relatively large number of axioms needed to prove a small program. 26 simple arithmetic theorems where used as axioms. This implies that moderately complex proofs would need a large number of ready-to-use theory in order to avoid re-proving trivial facts.

5.2.3 Feasibility evaluation

As there is no system similar to our platform, the most important evaluation question regards the feasibility of our concept for real world use cases. Having implemented the Program Store almost to full extend, and having obtained some first results in regards to proof validation, we have strong indications that the answer is positive.

5.2.4 Effect of other parameters

Some parameters that might have affected the results are:

- The optimization parameter of the Solidity compiler
- The gas efficiency of the evaluated smart contract code which can probably be further optimized
- The length of the evaluated proof which is probably far from the minimal

6 Conclusions and Future Work

6.1 Conclusions

We have proposed a blockchain-based platform design for conducting algorithmic problem solving competitions where proposed algorithms can be mechanically validated. The feasibility of this design has been demonstrated by a proof-of-concept implementation. As part of the design, we proposed a method for representing algorithmic problem definitions as procedural computer programs and showed that these representations can be effectively used to validate the correctness and complexity bounds of proposed algorithms.

The evaluation process revealed two main issues regarding the use of the platform, though none of them is irrecoverable. The first issue arises from the length of the proofs of correctness. The relatively large number of low level deduction steps makes it cumbersome for human users to compose proofs. The second issue arises from the relatively large number of axioms needed for a proof that only relies on simple arithmetic truths. This signals that more complex proofs would require a very large number of axioms or otherwise, they would have to prove many trivial facts thus making the proofs immensely lengthy.

6.2 Future Work

The main step of future work should be to extend the functionality of the Proof of Concept Proof Validator in order to support more program elements such as (recursive) function calls, global variables, records, dynamic memory allocation, arrays, maps, and real number arithmetic as these elements are essential even for basic algorithms.

To make proofs more easily composable, the proof composer can be enhanced to generate some of their low level details. This task may be assisted by an automated reasoning system such as Isabelle/HOL [NWP02] or Coq [Bar+97]. Another approach worth considering is the one suggested by Rodriguez [Rod16] which converts the process for composing a proof of correctness into a visual game.

The issue of proving trivial facts, can be tackled by a mechanism able to extend the set of axioms by logical reasoning. Facts proven by this mechanism would become straightly available and thus would not need re-proving. An interesting approach is followed by Carré et al. [Car+21] which regards proofs as a tree and demands low level proofs only for those nodes whose truth is being doubted by someone. This, approach however is interactive and thus not fully automatic.

Finally, optimizations on the gas consumption of proof validation are worth investigation. Even radical changes to the current setup can be explored, such as using formal verification systems other that Hoare logic.

Bibliography

[AB90] Pierre America and Frank de Boer. “Proving Total Correctness of Recursive Procedures”. In: *Inf. Comput.* 84.2 (Feb. 1990), pp. 129–162.

[Bar+97] Bruno Barras, Samuel Boutin, Cristina Cornes, et al. *The Coq Proof Assistant Reference Manual: Version 6.1*. Research Report RT-0203. Projet COQ. INRIA, May 1997, p. 214.

[Bor00] Richard Bornat. “Proving Pointer Programs in Hoare Logic”. In: *Mathematics of Program Construction*. Ed. by Roland Backhouse and José Nuno Oliveira. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 102–126.

[Bro] S. Gerth Brodal. *Algorithm Programming Competitions*. URL: `https://cs.au.dk/~gerth/code` (visited on May 30, 2021).

[But13] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. 2013. URL: `https://ethereum.org/en/whitepaper` (visited on May 30, 2021).

[Car+21] Sylvain Carré, Franck Gabriel, Clément Hongler, Gustavo Lacerda, and Gloria Capano. “Smart Proofs via Smart Contracts: Succinct and Informative Mathematical Derivations via Decentralized Markets”. In: *ArXiv* abs/2102.03044 (2021).

[Coo78] Stephen A. Cook. “Soundness and completeness of an axiom system for program verification”. In: *SIAM Journal of Computing* (1978).

[Flo67] Robert Floyd. “Assigning meanings to programs”. In: *Mathematical Aspects of Computer Science*. Ed. by J.T. Schwartz. Vol. 19. American Mathematical Society, 1967, pp. 19–32.

[Gal+21] Norbert Galm, Walter Guttmann, Farhad Mehta, Tobias Nipkow, and Leonor Prensa Nieto. *Hoare Logic*. 2021. URL: `https://isabelle.in.tum.de/library/HOL/HOL-Hoare/document.pdf` (visited on Sept. 29, 2021).

[Göd31] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für Mathematik und Physik* 38.1 (Dec. 1931), pp. 173–198.

[Guy04] Richard Guy. *Unsolved Problems in Number Theory*. Springer, New York, NY, 2004.

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[Mac18] John MacCormick. “Strategies for Basing the CS Theory Course on Non-Decision Problems”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 521–526.

[MyC21] MyCrypto. *The History of Ethereum’s Block Size Block Gas Limit*. 2021. URL: `https://blog.mycrypto.com/the-history-of-ethereums-block-size-block-gas-limit/` (visited on Oct. 25, 2021).

[Nie84] Hanne Riis Nielson. “Hoare Logic’s for Run-time Analysis of Programs”. English. Publisher Edinburgh University. PhD thesis. 1984.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. “Isabelle/HOL: A Proof Assistant for Higher-Order Logic”. In: 2002.

[Rod16] Dilia Rodriguez. *Verification Games: Crowd-Sourced Formal Verification*. Air Force Research Laboratory, Report AFRL-RI-RS-TR-2016-096. `https://apps.dtic.mil/sti/pdfs/AD1006471.pdf`. 2016.

[Su18] Borching Su. *MathCoin: A Blockchain Proposal that Helps Verify Mathematical Theorems In Public*. Cryptology ePrint Archive, Report 2018/271. `https://ia.cr/2018/271`. 2018.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993.

[Woo14] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. 2014. URL: `http://gavwood.com/Paper.pdf` (visited on May 30, 2021).

List of Acronyms

AST	Abstract Syntax Tree
API	Application Programming Interface
PoCPV	Proof of Concept Proof Validator
EVM	Ethereum Virtual Machine

List of Definitions and Propositions

List of Programs

List of Figures

List of Tables

A Specification of the Ivee Programming Language

A.1 High Level Description

A.1.1 Capabilities and limitations

Ivee is a high level procedural programming language with support for basic control structures (if-else, while), basic scalar data types (int, boolean, real), collection data types (arrays, maps) and user-defined custom data types (pascal-style records). Ivee supports dynamic memory allocation but, currently, does not support memory de-allocation.

Ivee does not support classes and lambda functions. This omission is intentional and aims to enforce the algorithmic/procedural programming paradigm.

A.1.2 Program structure

An Ivee program consists of two parts: a) data type definitions and b) function definitions and global variable declarations. Program A.1 demonstrates this structure.

```
# Data types
record Date
    int year
    int month
    int day
end

# Functions and global variables

int TOTAL_DAYS_PASSED # a global variable

# a function
function sameYear(Date a, Date b) -> boolean
    return a.year == b.year
end

real PI = 3.1415927 # another global variable
```

Program A.1: High level program structure

A.1.2.1 Scopes and function declarations

Variables declared in a function are local to that function while variables declared on the top level of a program are global. Functions are always global — no local function definitions are allowed. Furthermore, not variable of function may be used prior to it's declaration. Ivee allows for the pre-declaration of functions, by using the `declare` statement, so that they can be used prior to their implementation (this allows for mutually recursive functions).

```
declare function squareIt(int x) -> int

function addSquares(int a, int b) -> int
    return squareIt(a) + squareIt(b)
end

function squareIt(int x) -> int
    return x * x
end
```

Program A.2: Declare statement

A.1.3 Control structures

Ivee supports the standard if statement, with optional else-if and else branches, demonstrated by program A.3 and the standard while statement demonstrated by program A.4.

```
if [boolean expression]
    [block]
else if [boolean expression] # optional
    [block]
# [...more optional else if branches...]
else # optional
    [block]
end
```

Program A.3: If statement

```
while [boolean expression]
    [block]
end
```

Program A.4: While statement

The result of a function is specified by either using the `return` statement or assigning a value to the special `result` variable. The former breaks the flow of code execution and exits the functions immediately while the later continues with the subsequent command. In program A.5, functions `indexOfWithReturn` and `indexOfWithReturn` are not equivalent because the later returns the last index found.

```
function indexOfWithReturn(int val, int[] ar) -> int
    int index = 0
    while index < len(ar)
        if ar[index] == val
            return index
        end
    end
    return -1
end

# NOT equivalent to indexOfWithReturn
function indexOfWithResult(int val, int[] ar) -> int
    int index = 0
    while index < len(ar)
        if ar[index] == val
            result = index
        end
    end
    result = -1
end
```

Program A.5: Return vs result

A.1.4 Data types

A.1.4.1 Scalar types

Ivee supports the following scalar data types:

boolean	True/False values
int	Integer number values
real	Real number values

We do not specify upper/lower bounds for the `int` type neither precision for the `real` type. Both these number types may be assigned the semantics of computer arithmetic or mathematical arithmetic depending on the requirements of the work the Ivee language is being used in.

The `int` type is auto-convertible to the `real` type but *not* vice versa.

A.1.4.2 Container types

Ivee supports the following container type kinds:

array<E>	Fixed size collection of elements of type E indexed by the integer range [0, *size*-1]
map<K, V>	Fixed size collection of values of type V indexed by arbitrary keys of type K

Random access of elements of both these types should be considered to have $O(1)$ time cost. Container type variables are declared with specific E, K and V types while container type values are created with a specific size. Note there are no 2D arrays, arrays of arrays must be used instead. Program A.6 demonstrates container types.

```
int[] myIntArray # declare int array
myIntArray = new int[]{15} # create int array of size 15

int[][] myInt2DArray # declare int[] array
myInt2DArray = new int[][]{10} # create int[] array of size 10

map[real -> boolean] myMap # declare real -> boolean map
myMap = new map[real -> boolean]{20} # create real -> boolean map of size
    20
```

Program A.6: Container types

A.1.4.3 Record types

Ivee supports the declaration of used-defined custom record types. The declaration of a record type consist of a type name and a sequence of fields. A field is a pair consisting of the field name and the field type. The types the fields in a record type may refer to other record types or even the same record types leading to recursive data structures. Program A.7 demonstrates the declaration of record types and the creation of record objects.

A.1.4.4 Special types

Ivee contains the following special types:

None	Includes only the special value `None` which is assigned as default value to container and record type variables
Void	Implicitly set as the return type of functions which return nothing

```
# Simple record type
record Date
    int year
    int month
    int day
end

# Record type referring to other record type
record Submission
    int value
    Date submissionDate
end

# Recursive record type
record Node
    int value
    Node next
end

function testRecords()
    Submission answer = new Submission(42, new Date(1979, 10, 12))
end
```

Program A.7: Record types

It is not possible to declare a variable of a special type. However, the special value `None` can be accessed via the homonymous literal `None` and is assignable to container and record variables.

A.1.4.5 Default variable values

All variables take a default value upon declaration. Default values per variable type are given in table A.1.

Variable Type	Default Value
`boolean`	`False`
`int`	`0`
`real`	`0.0`
`array`	`None`
`map`	`None`
`record`	`None`

Table A.1: Default variable values in Ivee language

A.1.5 Builtin operators

Table A.2 summarizes the builtin operators of the Ivee language.

Operator	Return type	Operand types	Application	Description
`+`	`int`	`int, int`	infix	Integer addition
`-`	`int`	`int, int`	infix	Integer subtraction
`*`	`int`	`int, int`	infix	Integer multiplication
`/`	`int`	`int, int`	infix	Integer division
`%`	`int`	`int, int`	infix	Integer modulo
`-`	`int`	`int`	prefix	Integer negation
`+`	`real`	`real, real`	infix	Real addition
`-`	`real`	`real, real`	infix	Real subtraction
`*`	`real`	`real, real`	infix	Real multiplication
`/`	`real`	`real, real`	infix	Real division
`-`	`real`	`real`	prefix	Real negation
`==`	`boolean`	any same typed pair	infix	Equality
`!=`	`boolean`	any same typed pair	infix	Inequality
`<`	`boolean`	`int, int`	infix	Integer lesser
`<=`	`boolean`	`int, int`	infix	Integer lesser or equal
`>`	`boolean`	`int, int`	infix	Integer greater
`>=`	`boolean`	`int, int`	infix	Integer greater or equal
`<`	`boolean`	`real, real`	infix	Real lesser
`<=`	`boolean`	`real, real`	infix	Real lesser or equal
`>`	`boolean`	`real, real`	infix	Real greater
`>=`	`boolean`	`real, real`	infix	Real greater or equal
`and`	`boolean`	`boolean, boolean`	infix	Boolean and
`or`	`boolean`	`boolean, boolean`	infix	Boolean or
`not`	`boolean`	`boolean`	prefix	Boolean not
`len`	`int`	`array`	prefix	Array length

Table A.2: Builtin operators of the Ivee language

A.2 Language Elements

A.2.1 Lexical elements

Ivee has the following keywords: `function` `declare` `alias` `record` `new` `len` `if` `else` `while` `for` `in` `end` `return` `array` `map` `and` `or` `not` `True`, `False` `None` `int` `boolean` `real` `void`

Ivee has the following symbol operators: + - * / % == < > <= >=

Ivee has the following opening/closing parenthesis pairs: `()` `[]` `{}`

Ivee has the following special symbols: = `.` `,` ->

An `int` literal is a recognized by the regular expression: [0-9]+
An `real` literal is a recognized by the regular expression: [0-9]+\.[0-9]+
Of course, the recognition of real literals has priority over `int` literals.

Identifiers of type, field and variable names can contain ascii letters, digits and underscores and must not start with a number. Identifiers are recognized by the regular expression: [A-Za-z_][A-Za-z0-9_]+

Commands in Ivee are terminated by newlines. However, the contents of (matching) parentheses can safely include newlines and span multiple lines as part of the same command.

We skip the formal definition of the Ivee syntax as it is trivial work and can be inferred from the program examples.

A.2.2 Semantic elements

This subsection presents the elements of the semantics of an Ivee program. The fields (children) of each element are summarized in tables. The notation A $<$ B means that elements A are a special case of elements B. If an elements is abstract, it cannot occur in programs without been specialized.

We start this subsection from the most elementary notions and conclude with the notion of a full program.

Type (abstract)

Abstract notion of type (see A.1.4).

Fields	Kind	Description
`name`	string	Unique identifier for the type

Array Type < Type

Array types (see A.1.4.2).

Fields	Kind	Description
`elementType`	Type	Type of array's elements
`name`	string	Calculated as: "*elementType.name*[]"

Map Type < Type

Map types (see A.1.4.2).

Fields	Kind	Description
`keyType`	Type	Type of map's keys
`valueType`	Type	Type of map's values
`name`	string	Calculated as: "map(*keyType.name* -> *valueType.name*)"

Record Type < Type

Record types (see A.1.4.3).

Fields	Kind	Description
`fields`	list of Variables	Variables with unique names scoped within a Record Type
`name`	identifier	Unique identifier

Literal

Literals represent a constant value encoded as a lexical element of a program. Literals can be of type `int` which represents integer numbers, `real` which represents real numbers, or `boolean` which represents the values `True` and `False`.

Fields	Kind	Description
`type`	Type	Type of the represented value
`value`	string	The represented value

Variable

Variables are named placeholders for values.

Fields	Kind	Description
`type`	Type	Type of values the variable can hold
`name`	identifier	Unique identifier within the scope of the variable

Expression (abstract)

Abstract notion for something that is evaluated at runtime to produce a value.

Fields	Kind	Description
`type`	Type	Type of the resulting value

Literal Read < Expression

A Literal Read is an expression consisting of a single literal.

Fields	Kind	Description
`literal`	Literal	The literal to read
`type`	Type	Same as the type of the literal

Variable Access < Expression

A Variable Access is an expression consisting of a single variable. Can be used in write mode as the left hand of an assignment.

Fields	Kind	Description
`variable`	Variable	The variable to access
`type`	Type	Same as the type of the variable

Operation < Expression, Command

An operation is an expression that invokes a function or an operator. Because function invocations can have side effects, an Operation that invokes a function (not an operator) can occur standalone as a command.

Fields	Kind	Description
`operator`	Function Declaration	Operator to apply
`operands`	list of Expressions	The variable to access
`type`	Type	Same as the return type of the operator

Array Creation < Expression

An Array Creation is an expression which evaluates to a newly allocated array. An Array Creation must necessarily specify either the size of the array of the initial elements of the array.

Fields	Kind	Description
`type`	Array Type	Type of the array
`size?`	int Expression	Size of the array
`initializer?`	list of *type.elementType* Expressions	Elements of the array

Map Creation < Expression

An Map Creation is an expression which evaluates to a newly allocated map.

Fields	Kind	Description
`type`	Map Type	Type of the map
`size`	int Expression	Size of the map

Record Creation < Expression

An Record Creation is an expression which evaluates to a newly allocated record.

Fields	Kind	Description
`type`	Record Type	Type of the record
`fieldValues`	list of Expressions	Values of the record's fields

Array Access < Expression

An Array Access is an expression which reads an element of an array by index. Can be used in write mode as the left hand of an assignment.

Fields	Kind	Description
`array`	array Expression	Array to access
`index`	int Expression	Index of the array to access
`type`	Type	Same as the *elementType* of the array

Map Access < Expression

A Map Access is an expression which reads a value of a map by key. Can be used in write mode as the left hand of an assignment.

Fields	Kind	Description
`map`	map Expression	Map to access
`key`	map’s *keyType* Expression	Key of the map to access
`type`	Type	Same as the *valueType* of the map

Field Access < Expression

A Field Access is an expression which reads a field of a record. Can be used in write mode as the left hand of an assignment.

Fields	Kind	Description
`record`	record Expression	Record to access
`field`	Variable	Field of the record to access
`type`	Type	Same as the type of the field

Command (abstract)

Abstract notion for a statement that is executed at runtime.

Block

A block is a list of commands.

Fields	Kind	Description
`commands`	list of Commands	Commands of the block

Variable Declaration < Command

A Variable Declaration is a Command that declares and optionally initializes a Variable. Variable Declarations can occur in function bodies, declaring local Variables, or in the top level of programs, declaring global Variables.

Fields	Kind	Description
`variable`	Variable	Declared Variable
`initializer?`	Expression of type *variable.type*	Initializer of the variable

Assignment < Command

An Assignment is a command that assigns a new value to a Variable, Array element, Map key or Record field.

Fields	Kind	Description
`leftHand`	Expression	Variable, Array, Map or Field Access
`value`	Expression	Value to assigned to leftHand

Return Statement < Command

A Return Statement ends the execution of a function and (if the function is not Void) returns a result.

Fields	Kind	Description
`value`	Expression	Value to return

If Statement < Command

An If Statement executes on of two branches depending on the evaluation of a conditions. If the else branch is missing, the execution continues with the next command.

Fields	Kind	Description
`condition`	boolean Expression	Condition to evaluate
`body`	Block	Block to execute condition is `True`
`elseBody?`	Block	Block to execute if condition is `False`

NOTE: Else-if branches are just syntactic sugar for if statements nested in the else branch. Program A.8 demonstrates the semantics of else-if branches.

While Statement < Command

A While statement executes a block while a condition holds.

Fields	Kind	Description
`condition`	boolean Expression	Condition to evaluate
`body`	Block	Block to execute *while* condition is `True`

```
# this statement
if value < 0
  sign = -1
else if value > 0
    sign = 1
else
    sign = 0
end

# is translated to this
if value < 0
  sign = -1
else
  if value > 0
    sign = 1
  else
    sign = 0
  end
end
```

Program A.8: Else-if is syntactic sugar

Function Declaration

A Function Declaration represents the signature of a function of operator.

Fields	Kind	Description
`name`	string	Identifier of the function
`arguments`	list of Variables	Arguments of the function
`returnType`	Type	Return type of the function
`resultVariable`	Variable	Special result variable of the function of type *returnType*
`isOperator`	boolean	Whether this declaration regards an operator

Function Definition

A Function Definition consists of the declaration and the implementation of a function.

Fields	Kind	Description
`declaration`	Function Declaration	Declaration of the function
`body`	Block	Body of the function

Program

A Program consists of definitions of custom Record Types, declarations of global variables and definitions of functions.

Fields	Kind	Description
`recordTypes`	list of Record Types	Record types declared
`functions`	list of Function Definitions	Functions defined
`gobalVariables`	list of Variable Declarations	Global variables declared

B Hoare Logic Configuration

We adapt the Hoare logic for formally proving truths about programs in a special configuration that suits our purposes.

First, we scope logical theorems to code locations. A code location is essentially a line of code and consists of a code block and a command index. For example the following:
$x == False\ @1.2$
means that the variable x is equal to `False` in command 2 of code block 1.

Of course, we still maintain the us-coped theorems, which apply everywhere. Any deduction must take into account that the scope of the participating theorems are compatible i.e. they have the same scope or one of them is un-scoped.

A proof in this proposed configuration will consist of a sequences of such (scoped) theorems, each produced by the application of a certain deduction scheme.

A full proof in this configuration is listed in appendix C.

B.1 Proof Elements

Sentence

A Sentence is a logical formula that represents a truth about the Variables of a Program under examination. Furthermore, a Sentence may introduce logical Variables i.e extra Variables that do not exist in the Program. Logical Variables may be (universally or existentially) quantified.

In the simplest form, a Sentence expresses membership of certain Expressions (involving logical and Program Variables) in a relation. For example, the following Sentence states that the pair $(x, a + 1)$ belongs in relation $>$ (greater) i.e. that the value of x is greater than the value of $a + 1$:

$$x > a + 1$$

Sentences can be composed into more complex Sentences by logical operators. For example, the following Sentence uses the → (implication) operator to state that if a equals b then b equals a for all int a and int b.

$$\forall int\ a\ \forall int\ b\ ((a == b) \rightarrow (b == a))$$

The expressive potential of Sentences is fully presented below as an EBNF grammar. (Note that this EBNF grammar does not aim to define a language but rather to present the semantics of Sentences. So it does not deal with issues such as separator tokens, prefix or infix operator application etc.)

$Sentence = QuantifiedVariables, LogicalExpression$
$QuantifiedVariables = \{Quantifier, Variable\}$
$Quantifier = \exists \mid \forall$
$LogicalExpression = RelationMembership \mid LogicalOperation$
$RelationMembership = RelationSymbol, RelationMembers$
$RelationMembers = Expression, \{Expression\}$
$LogicalOperation = LogicalOperator, LogicalOperationArguments$
$LogicalOperationArguments = Sentence, \{Sentence\}$

Supported $RelationSymbols$ are the following built-in operators which operate on `int`, `boolean` and `real` values: `==`, `!=`, `<`, `<=`, `>`, `>=`, `and`, `or`, `not`. These operators are presented in table A.2.

Supported $LogicalOperators$, which operate on Sentences, are the following: ∧ (conjuction), ∨ (disjunction), ¬ (negation), → (implication).

A Sentence with a single relation membership and with no quantified variables will be called a **Fact**. A Fact with equality (==) as relation symbol will be called an **Identity**.

Code Location

A Code Location is essentially a Command in a Program. It is determined by the Block where the Command is located and the index of the Command within that Block.

Code Locations are represented with the notation `@BlockCode.CommandIndex`. Furthermore, there are two special types of code locations: a) *end-of-block*, which corresponds to the location right after the last command of block and b) *global*, which is used to denote that a Theorem is un-scoped and applies everywhere. Program B.1 notes the Code Locations of each command and the end-of-block code locations (@3.3, @4.2, @2.2, @1.4).

```
function solve(int n) -> int
  int r = 0 # @1.1
  result = -1 # @1.2
  while r < n # @1.3
    if r*r == n # @2.1
      result = r # @3.1
      r = n # @3.2
    else # @3.3
      r = r+1 # @4.1
    end # @4.2
  end # @2.2
end # @1.4
```

Program B.1: Code locations

Theorem

A Theorem is a Sentence proven to hold at a specific Code Location (or globally). If a theorem applied at a Code Location, we call it a *precondition* of the statement at that Code Location.

Deduction

A Deduction is the action of applying a Deduction Scheme (see B.2) on a set of input arguments to produce a new Theorem. Input arguments may be other Theorems, Code Locations, Expressions etc.

Step

A Step is a Theorem accompanied by the Deduction that produced it.

Proof

A proof is a sequence of Steps where each Step is produced by a Deduction that can only have Theorems of preceding Steps as input arguments. A Proof cannot be valid if it has an "open invariant" as described later in the Total While deduction scheme.

B.2 Deduction Schemes

This Hoare logic configuration recognizes deduction schemes belonging to the following groups:

Hoare	Standard Hoare logic deductions (as seen in [NK14])
Hoare simplifications	Deduction that simplify trivial Hoare logic deductions
Classical	Classical logic deductions
Boolean handling	Deductions for handling truths about boolean expressions
Computation	Deduction for computing the value of simple expressions involving literal values

Each deduction scheme is presented below by a table describing it's arguments and the format of the produced Theorem.

The notation `theorem`[a/b] means "`theorem` after replacing `b` by `a`".

Axiom (classical)

This is a pseudo-deduction used to state universally accepted truths.

Input Arguments		
Argument	Kind	Description
`axiom`	Sentence	A (universally acceptable) Sentence

Produced Theorem	
Sentence	`axiom`
Scope	Global

Assignment scheme (Hoare)

This is the standard Hoare logic assignment scheme re-configured for forward reasoning.

Input Arguments		
Argument	Kind	Description
`assignment`	Code Location	`x = E` statement
`precondition`	Theorem	Can only include `x` in sub-expressions equal to `E`

Produced Theorem	
Sentence	`precondition[x/E]`
Scope	Next of @`assignment`

If Condition Assertion (Hoare)

This deduction scheme asserts the condition an if statement to the if-body.

Input Arguments		
Argument	Kind	Description
`ifStatement`	Code Location	`if C then A else B`? statement

Produced Theorem	
Sentence	`C == True`
Scope	Start of `A`

Else Negated Condition Assertion (Hoare)

This deduction scheme asserts the negation of the condition an if statement to the else-body.

Input Arguments		
Argument	Kind	Description
`ifStatement`	Code Location	`if C then A else B` statement

Produced Theorem	
Sentence	`C == False`
Scope	Start of `B`

If scheme (Hoare)

This is the standard Hoare logic if scheme.

Input Arguments		
Argument	Kind	Description
`ifStatement`	Code Location	`if C then A else B?` statement
`ifPostcondition`	Theorem	Must apply @ end of `A`
`elsePostcondition`	Theorem	Must apply @ end of `B` or @`ifStatement` if no `B` and have the same Sentence as the `ifPostcondition`

Produced Theorem	
Sentence	`ifPostcondition`
Scope	Next of `ifStatement`

While Condition Assertion (Hoare)

This deduction scheme asserts the condition a while statement to the while-body.

Input Arguments		
Argument	Kind	Description
`whileStatement`	Code Location	`while C do A` statement

Produced Theorem	
Sentence	`C == True`
Scope	Start of `A`

Set Variant (Hoare)

Sets the variant expression for a while statement.

Input Arguments		
Argument	Kind	Description
`whileStatement`	Code Location	`while C do A` statement
`variable`	Variable	Logical Variable to hold the variant's start-of-loop value
`expression`	int Expression	Expression that calculates the variant

Produced Theorem	
Sentence	`variable == expression`
Scope	Start of `A`

Total While scheme (Hoare)

While scheme for total correctness. Proves that a while statement terminates.

Input Arguments		
Argument	Kind	Description
`whileStatement`	Code Location	`while C do A` statement
`variantDefinition`	Theorem	`variable == `*(int)* `expression` @ start of `A`
`variantNonNegative`	Theorem	`expression` >= 0 @ end of `A`
`variantDecreasing`	Theorem	`variable > expression` @ end of `A`

Produced Theorem	
Sentence	`C == True`
Scope	Next of `whileStatement`

Invariant Assertion (Hoare)

Asserts a precondition of a while statement to the while-body. The asserted theorem will be an "open invariant" required to be proved by the Invariant Proof scheme.

Input Arguments		
Argument	Kind	Description
`whileStatement`	Code Location	`while C do A` statement
`invariant`	Theorem	Must apply @`whileStatement`

Produced Theorem	
Sentence	`invariant`
Scope	Start of `A`
Opens	An invariant for `whileStatement`

Invariant Proof (Hoare)

Proves an invariant asserted to a while statement by the Invariant Assertion scheme. Requires a `terminationProof` obtained by the Total While scheme.

Input Arguments		
Argument	Kind	Description
`whileStatement`	Code Location	`while C do A` statement
`invariantStart`	Theorem	Must apply @ Start of `A`
`invariantEnd`	Theorem	Must be equal to `invariantStart` and apply @ end of `A`
`terminationProof`	Theorem	A theorem proved by Total While scheme for the `whileStatement`

Produced Theorem	
Sentence	`C == True`
Scope	Next of `whileStatement`
Resolves	The invariant that asserted `invariantStart`

Irrelevant Variable Assignment scheme (Hoare simplifications)

Forwards a precondition though an assignment when the left hand of the assignment does not occur in the precondition.

Input Arguments		
Argument	Kind	Description
`assignment`	Code Location	`x = E` statement
`precondition`	Theorem	Cannot include variable `x`

Produced Theorem	
Sentence	`precondition`
Scope	Next of @`assignment`

Non Self Referring Assignment scheme (Hoare simplifications)

A simplified version of the Assignment scheme for when the left hand does not occur in the right hand.

Input Arguments		
Argument	Kind	Description
`assignment`	Code Location	`x == E` statement where x does not occur in `E`

Produced Theorem	
Sentence	`x = E`
Scope	Next of @`assignment`

Pass to If Block (Hoare simplifications)

Inserts a precondition of an if statement to the if-block.

Input Arguments		
Argument	Kind	Description
`ifStatement`	Code Location	`if C then A else B`? statement
`precondition`	Theorem	Must apply `@ifStatement`

Produced Theorem	
Sentence	`precondition`
Scope	Start of `A`

Pass to Else Block (Hoare simplifications)

Inserts a precondition of an if statement to the else-block.

Input Arguments		
Argument	Kind	Description
`ifStatement`	Code Location	`if C then A else B` statement
`precondition`	Theorem	Must apply `@ifStatement`

Produced Theorem	
Sentence	`precondition`
Scope	Start of `B`

Modus Ponens (classical)

Modus Ponens.

Input Arguments		
Argument	Kind	Description
`antecedent`	Theorem	Any Theorem
`implication`	Theorem	Theorem of the form `antecedent` → `consequent`

Produced Theorem	
Sentence	`consequent`
Scope	Most specific scope between `antecedent` and `implication`

Specification (classical)

Replaces a universally quantified variable by an Expression.

Input Arguments		
Argument	Kind	Description
`original`	Theorem	Theorem with a universally quantified variable
`variable`	Variable	Universally quantified variable of `original`
`replacement`	Expression	Expression of same type as `variable`

Produced Theorem	
Sentence	`original[replacement/variable]`
Scope	Same as `original`

Replacement (classical)

Replace an Expression by another Expression when these Expressions are proved equal by an identity Theorem.

Input Arguments		
Argument	Kind	Description
`original`	Theorem	Any Theorem
`identity`	Theorem	Theorem of the form `a = b`
`replacementPart`	`Left` or `Right`	`Left`: replace `a` by `b`, `Right`: replace `b` by `a`
`replacementIndex`	integer	Which occurrence in `original` to replace

Produced Theorem	
Sentence	`original` after replacing `replacementIndex`-th occurrence of `a` by `b` (or `b` by `a` if `replacementPart` is `Right`)
Scope	Most specific scope between `original` and `identity`

Conjunction (classical)

Creates the conjunction of two theorems.

Input Arguments		
Argument	Kind	Description
a	Theorem	Any Theorem
b	Theorem	Any Theorem

Produced Theorem	
Sentence	a∧b
Scope	Most specific scope between a and b

Boolean to Logical (boolean handling)

Converts a fact regarding the truth value of a boolean expression to a corresponding relation membership or logical operation. This is achieved by converting comparison operators to relation symbols and boolean operators to logical operators.

Input Arguments		
Argument	Kind	Description
`booleanFact`	Fact	Identity of the form `E==True` or `E==False`

Produced Theorem	
Sentence	`E` if `E==True` or ¬`E` if `E==False`
Scope	Same as `booleanFact`

Fact to Boolean (boolean handling)

Converts a fact to the truthiness of a boolean expression by applying the reverse process of the Boolean to Logical scheme.

Input Arguments		
Argument	Kind	Description
`fact`	Theorem	Any Fact

Produced Theorem	
Sentence	`fact == True`
Scope	Same as `fact`

Negated Fact to Boolean (boolean handling)

Converts the negation of a fact to the falseness of a boolean expression by applying the reverse process of the Boolean to Logical scheme.

Input Arguments		
Argument	Kind	Description
`negatedFact`	Theorem	A negated Fact of the form ¬`fact`

Produced Theorem	
Sentence	`fact == False`
Scope	Same as `negatedFact`

Computation

Computes a specified expression which contains only Literals (and not Variables) and asserts it's equality to the computed value.

Input Arguments		
Argument	Kind	Description
`expression`	Expression	Can only contain literals and builtin operators

Produced Theorem	
Sentence	`expression == compute(expression)`
Scope	Global

C Full-length Listing of the Experimentally Validated Proof

We list the sample proof used for evaluation (chapter 5) in it’s full length.

The combined `correctness` program is presented bellow.

```
function correctness(int i$12@definition, int result$11@solver, int
    h$16@definition)

  # ...variables declaration omitted...

  result$11@definition = i$12@definition >= 0@definition # 1.0
  if result$11@definition # 1.1
    while ((result$11@solver + 1@solver) * (result$11@solver + 1@solver))
     <= n$12@solver # 2.0
      result$11@solver = result$11@solver + 1@solver # 6.0
    end # 6.1
    result$13@definition = ((s$15@definition * s$15@definition) >
   i$14@definition) or (((s$15@definition + 1@definition) * (
   s$15@definition + 1@definition)) <= i$14@definition) # 2.1
  end # 2.2
end # 1.2
```

Program C.1: Correctness program for solver 5.2 against definition 5.1

The target postcondition is:
$result\$13@definition == False$ @1.2

Note that variables and literals are appended with their code and their context (i.e. whether they come from the definition or the solver program) in order to avoid ambiguity.

The proof of correctness follows.

1. ∀boolean x ∀boolean y ((x == y))→((y == x))
Axiom

2. ∀int x ∀int y ((x == y))→((y == x))
Axiom

3. $\forall$int x $\forall$int y ((x + y) == (y + x))
Axiom

4. $\forall$int x ((x - x) == 0)
Axiom

5. $\forall$int x $\forall$int y ((x + -(y)) == (x - y))
Axiom

6. $\forall$int x $\forall$int y ((x == y))$\rightarrow$((x <= y))
Axiom

7. $\forall$int x $\forall$int y ((x == y))$\rightarrow$((x >= y))
Axiom

8. $\forall$int x $\forall$int y ((x < y))$\rightarrow$((x <= y))
Axiom

9. $\forall$int x $\forall$int y ((x > y))$\rightarrow$((x >= y))
Axiom

10. $\forall$int x $\forall$int y ((x < y))$\rightarrow$($\neg$((x >= y)))
Axiom

11. $\forall$int x $\forall$int y ((x <= y))$\rightarrow$($\neg$((x > y)))
Axiom

12. $\forall$int x $\forall$int y ((x >= y))$\rightarrow$((y <= x))
Axiom

13. $\forall$int x $\forall$int y ((x <= y))$\rightarrow$((y >= x))
Axiom

14. $\forall$int x $\forall$int y ((x > y))$\rightarrow$((y < x))
Axiom

15. $\forall$int x $\forall$int y ((x < y))$\rightarrow$((y > x))
Axiom

16. $\forall$int x $\forall$int y $\forall$int a ((x == y))$\rightarrow$(((x + a) == (y + a)))
Axiom

17. ∀int x ∀int y ∀int a ((x > y))→(((x + a) > (y + a)))
Axiom

18. ∀int x ∀int y ∀int a ((x >= y))→(((x + a) >= (y + a)))
Axiom

19. ∀int x ∀int y ∀int a ((x == y))→(((x - a) == (y - a)))
Axiom

20. ∀int x ∀int y ∀int a ((x > y))→(((x - a) > (y - a)))
Axiom

21. ∀int x ∀int y ∀int a ((x >= y))→(((x - a) >= (y - a)))
Axiom

22. ∀int x ∀int y ((x > 0))→((y < (y + x)))
Axiom

23. ∀int x ∀int y ((x > 0))→((y > (y - x)))
Axiom

24. ∀int x ∀int y ∀int a (((x <= y))∧((y <= a)))→((x <= a))
Axiom

25. ∀int x ∀int y ((x < y))→((-(x) > -(y)))
Axiom

26. ∀int x ∀int y ((x >= 0))→(((x <= y))→(((x * x) <= (y * y))))
Axiom

27. ∀int x ∀int y ((x >= 0))→(((x < y))→(((x * x) < (y * y))))
Axiom

28. (result$11@definition == False) @1.0
Precondition

29. (result$13@definition == False) @1.1
Precondition

30. (n$12@solver == i$12@definition) @2.0
Precondition

31. (result$11@solver == 0) @2.0
Precondition

32. (i$12@definition == i$14@definition) @2.1
Precondition

33. (s$15@definition == result$11@solver) @2.1
Precondition

34. (result$13@definition == False) @2.1
Precondition

35. ∀int y ((result$11@solver == y))→((result$11@solver >= y))
From 7 specifying x to result$ 11@solver

36. ((result$11@solver == 0))→((result$11@solver >= 0))
From 35 specifying y to 0

37. (result$11@solver >= 0) @2.0
Modus Ponens 31, 36

38. (result$11@solver >= 0) @6.0
Assert 37 as invariant

39. ((((result$11@solver + 1@solver) * (result$11@solver + 1@solver)) <= n$12@solver)
== True) @6.0
Assert condition to while block

40. (((result$11@solver + 1@solver) * (result$11@solver + 1@solver)) <= n$12@solver)
@6.0
From 39 by converting boolean expression equality to logical sentence

41. (t == (n$12@solver - (result$11@solver * result$11@solver))) @6.0
*Set t=n$ 12@solver - (result$ 11@solver * result$ 11@solver) as variant*

42. ∀int y ((result$11@solver >= 0))→(((result$11@solver < y))→(((result$11@solver *
result$11@solver) < (y * y))))
From 27 specifying x to result$ 11@solver

43. ((result$11@solver >= 0))→(((result$11@solver < (result$11@solver + 1@solver)))→(((result$11@solver
* result$11@solver) < ((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))))
From 42 specifying y to (result$ 11@solver + 1@solver)

44. ((result$11@solver < (result$11@solver + 1@solver)))→(((result$11@solver * result$11@solver) < ((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))) @6.0
Modus Ponens 38, 43

45. ∀int y ((1@solver > 0))→((y < (y + 1@solver)))
From 22 specifying x to 1@solver

46. ((1@solver > 0))→((result$11@solver < (result$11@solver + 1@solver)))
From 45 specifying y to result$ 11@solver

47. ((1@solver > 0) == True)
Compute (1@solver > 0)

48. (1@solver > 0)
From 47 by converting boolean expression equality to logical sentence

49. (result$11@solver < (result$11@solver + 1@solver))
Modus Ponens 48, 46

50. ((result$11@solver * result$11@solver) < ((result$11@solver + 1@solver) * (result$11@solver + 1@solver))) @6.0
Modus Ponens 49, 44

51. ∀int y (((result$11@solver * result$11@solver) < y))→((-((result$11@solver * result$11@solver)) > -(y)))
*From 25 specifying x to (result$ 11@solver * result$ 11@solver)*

52. (((result$11@solver * result$11@solver) < ((result$11@solver + 1@solver) * (result$11@solver + 1@solver))))→((-((result$11@solver * result$11@solver)) > -(((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))))
*From 51 specifying y to ((result$ 11@solver + 1@solver) * (result$ 11@solver + 1@solver))*

53. (-((result$11@solver * result$11@solver)) > -(((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))) @6.0
Modus Ponens 50, 52

54. ∀int y ∀int a ((-((result$11@solver * result$11@solver)) > y))→(((-((result$11@solver * result$11@solver)) + a) > (y + a)))
*From 17 specifying x to -((result$ 11@solver * result$ 11@solver))*

55. ∀int a ((-((result$11@solver * result$11@solver)) > -(((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))))→(((-((result$11@solver * result$11@solver)) + a) > (-(((result$11@solver + 1@solver) * (result$11@solver + 1@solver))) + a)))
*From 54 specifying y to -(((result$ 11@solver + 1@solver) * (result$ 11@solver + 1@solver)))*

56. ((-((result$11@solver * result$11@solver)) > -(((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))))→(((-((result$11@solver * result$11@solver)) + n$12@solver) > (-(((result$11@solver + 1@solver) * (result$11@solver + 1@solver))) + n$12@solver)))
From 55 specifying a to n$ 12@solver

57. ((-((result$11@solver * result$11@solver)) + n$12@solver) > (-(((result$11@solver + 1@solver) * (result$11@solver + 1@solver))) + n$12@solver)) @6.0
Modus Ponens 53, 56

58. ∀int y ((n$12@solver + y) == (y + n$12@solver))
From 3 specifying x to n$ 12@solver

59. ((n$12@solver + -((result$11@solver * result$11@solver))) == (-((result$11@solver * result$11@solver)) + n$12@solver))
*From 58 specifying y to -((result$ 11@solver * result$ 11@solver))*

60. ∀int y ((n$12@solver + -(y)) == (n$12@solver - y))
From 5 specifying x to n$ 12@solver

61. ((n$12@solver + -((result$11@solver * result$11@solver))) == (n$12@solver - (result$11@solver * result$11@solver)))
*From 60 specifying y to (result$ 11@solver * result$ 11@solver)*

62. ((-((result$11@solver * result$11@solver)) + n$12@solver) == (n$12@solver - (result$11@solver * result$11@solver)))
From 61 replacing by 59 at position 0

63. ((n$12@solver - (result$11@solver * result$11@solver)) > (-(((result$11@solver + 1@solver) * (result$11@solver + 1@solver))) + n$12@solver)) @6.0
From 57 replacing by 62 at position 0

64. ((n$12@solver + -(((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))) == (-(((result$11@solver + 1@solver) * (result$11@solver + 1@solver))) + n$12@solver))
*From 58 specifying y to -(((result$ 11@solver + 1@solver) * (result$ 11@solver + 1@solver)))*

65. ((n$12@solver + -(((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))) == (n$12@solver - ((result$11@solver + 1@solver) * (result$11@solver + 1@solver))))

*From 60 specifying y to ((result$ 11@solver + 1@solver) * (result$ 11@solver + 1@solver))*

66. ((-(((result$11@solver + 1@solver) * (result$11@solver + 1@solver))) + n$12@solver) == (n$12@solver - ((result$11@solver + 1@solver) * (result$11@solver + 1@solver))))
From 65 replacing by 64 at position 0

67. ((n$12@solver - (result$11@solver * result$11@solver)) > (n$12@solver - ((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))) @6.0
From 63 replacing by 66 at position 0

68. (t > (n$12@solver - ((result$11@solver + 1@solver) * (result$11@solver + 1@solver)))) @6.0
From 67 replacing by 41 at position 0

69. (t > (n$12@solver - (result$11@solver * result$11@solver))) @6.1
Assignment Scheme for precondition 68 at command 6.0

70. ((result$11@solver * result$11@solver) <= n$12@solver) @6.1
Assignment Scheme for precondition 40 at command 6.0

71. ∀int y (((result$11@solver * result$11@solver) <= y))→((y >= (result$11@solver * result$11@solver)))
*From 13 specifying x to (result$ 11@solver * result$ 11@solver)*

72. (((result$11@solver * result$11@solver) <= n$12@solver))→((n$12@solver >= (result$11@solver * result$11@solver)))
From 71 specifying y to n$ 12@solver

73. (n$12@solver >= (result$11@solver * result$11@solver)) @6.1
Modus Ponens 70, 72

74. ∀int y ∀int a ((n$12@solver >= y))→(((n$12@solver - a) >= (y - a)))
From 21 specifying x to n$ 12@solver

75. ∀int a ((n$12@solver >= (result$11@solver * result$11@solver)))→(((n$12@solver - a) >= ((result$11@solver * result$11@solver) - a)))
*From 74 specifying y to (result$ 11@solver * result$ 11@solver)*

76. ((n$12@solver >= (result$11@solver * result$11@solver)))→(((n$12@solver - (result$11@solver * result$11@solver)) >= ((result$11@solver * result$11@solver) - (result$11@solver * result$11@solver))))
*From 75 specifying a to (result$ 11@solver * result$ 11@solver)*

77. ((n$12@solver - (result$11@solver * result$11@solver)) >= ((result$11@solver * result$11@solver) - (result$11@solver * result$11@solver))) @6.1
Modus Ponens 73, 76

78. (((result$11@solver * result$11@solver) - (result$11@solver * result$11@solver)) == 0)
*From 4 specifying x to (result$ 11@solver * result$ 11@solver)*

79. ((n$12@solver - (result$11@solver * result$11@solver)) >= 0) @6.1
From 77 replacing by 78 at position 0

80. ((((result$11@solver + 1@solver) * (result$11@solver + 1@solver)) <= n$12@solver) == False) @2.1
While total by 79 and 69 for variant 41

81. ((((s$15@definition + 1@solver) * (result$11@solver + 1@solver)) <= n$12@solver) == False) @2.1
From 80 replacing by 33 at position 0

82. ((((s$15@definition + 1@solver) * (s$15@definition + 1@solver)) <= n$12@solver) == False) @2.1
From 81 replacing by 33 at position 0

83. (n$12@solver == i$12@definition) @6.0
Assert 30 as invariant

84. (n$12@solver == i$12@definition) @6.1
Irrelevant Variable Assignment Scheme for precondition 83 at command 6.0

85. (n$12@solver == i$12@definition) @2.1
Extract invariant 83/84 by 80

86. ((((s$15@definition + 1@solver) * (s$15@definition + 1@solver)) <= i$12@definition) == False) @2.1
From 82 replacing by 85 at position 0

87. ∀int y ((result$11@solver >= y))→((y <= result$11@solver))
From 12 specifying x to result$ 11@solver

88. ((result$11@solver >= 0))→((0 <= result$11@solver))
From 87 specifying y to 0

89. (0 <= result$11@solver) @6.0
Modus Ponens 38, 88

90. ∀int y ∀int a (((0 <= y))∧((y <= a)))→((0 <= a))
From 24 specifying x to 0

91. ∀int a (((0 <= result$11@solver))∧((result$11@solver <= a)))→((0 <= a))
From 90 specifying y to result$ 11@solver

92. (((0 <= result$11@solver))∧((result$11@solver <= (result$11@solver + 1@solver))))→((0 <= (result$11@solver + 1@solver)))
From 91 specifying a to (result$ 11@solver + 1@solver)

93. ∀int y ((result$11@solver < y))→((result$11@solver <= y))
From 8 specifying x to result$ 11@solver

94. ((result$11@solver < (result$11@solver + 1@solver)))→((result$11@solver <= (result$11@solver + 1@solver)))
From 93 specifying y to (result$ 11@solver + 1@solver)

95. (result$11@solver <= (result$11@solver + 1@solver))
Modus Ponens 49, 94

96. ((0 <= result$11@solver))∧((result$11@solver <= (result$11@solver + 1@solver))) @6.0
Conjunct 89 and 95

97. (0 <= (result$11@solver + 1@solver)) @6.0
Modus Ponens 96, 92

98. (0 <= result$11@solver) @6.1
Assignment Scheme for precondition 97 at command 6.0

99. ∀int y ((0 <= y))→((y >= 0))
From 13 specifying x to 0

100. ((0 <= result$11@solver))→((result$11@solver >= 0))
From 99 specifying y to result$ 11@solver

101. (result$11@solver >= 0) @6.1
Modus Ponens 98, 100

102. (result$11@solver >= 0) @2.1
Extract invariant 38/101 by 80

103. ((0 * 0) == 0)
*Compute (0 * 0)*

104. ((result$11@solver * 0) == 0) @2.0
From 103 replacing by 31 at position 0

105. ((result$11@solver * result$11@solver) == 0) @2.0
From 104 replacing by 31 at position 0

106. (result$11@definition == (i$12@definition >= 0@definition)) @1.1
Non Self Referring Assignment Scheme at command 1.0

107. (result$11@definition == (i$12@definition >= 0@definition)) @2.0
Pass 106 as precondition to if block

108. (result$11@definition == True) @2.0
Assert condition to if block

109. ((i$12@definition >= 0@definition) == True) @2.0
From 108 replacing by 107 at position 0

110. (i$12@definition >= 0@definition) @2.0
From 109 by converting boolean expression equality to logical sentence

111. (0@definition == 0)
Compute 0@definition

112. (i$12@definition >= 0) @2.0
From 110 replacing by 111 at position 0

113. (i$12@definition >= (result$11@solver * result$11@solver)) @2.0
From 112 replacing by 105 at position 0

114. (n$12@solver >= (result$11@solver * result$11@solver)) @2.0
From 113 replacing by 30 at position 0

115. (n$12@solver >= (result$11@solver * result$11@solver)) @6.0
Assert 114 as invariant

116. (n$12@solver >= (result$11@solver * result$11@solver)) @2.1
Extract invariant 115/73 by 80

117. (i$12@definition >= (result$11@solver * result$11@solver)) @2.1
From 116 replacing by 85 at position 0

118. (i$12@definition >= (s$15@definition * result$11@solver)) @2.1
From 117 replacing by 33 at position 0

119. (i$12@definition >= (s$15@definition * s$15@definition)) @2.1
From 118 replacing by 33 at position 0

120. (i$14@definition >= (s$15@definition * s$15@definition)) @2.1
From 119 replacing by 32 at position 0

121. ∀int y ((i$14@definition >= y))→((y <= i$14@definition))
From 12 specifying x to i$ 14@definition

122. ((i$14@definition >= (s$15@definition * s$15@definition)))→(((s$15@definition * s$15@definition) <= i$14@definition))
*From 121 specifying y to (s$ 15@definition * s$ 15@definition)*

123. ((s$15@definition * s$15@definition) <= i$14@definition) @2.1
Modus Ponens 120, 122

124. ∀int y (((s$15@definition * s$15@definition) <= y))→(¬(((s$15@definition * s$15@definition) > y)))
*From 11 specifying x to (s$ 15@definition * s$ 15@definition)*

125. (((s$15@definition * s$15@definition) <= i$14@definition))→(¬(((s$15@definition * s$15@definition) > i$14@definition)))
From 124 specifying y to i$ 14@definition

126. ¬(((s$15@definition * s$15@definition) > i$14@definition)) @2.1
Modus Ponens 123, 125

127. (((s$15@definition * s$15@definition) > i$14@definition) == False)
Negated Fact 126 to boolean

128. ((((s$15@definition + 1@solver) * (s$15@definition + 1@solver)) <= i$14@definition) == False) @2.1

From 86 replacing by 32 at position 0

129. (result$13@definition == (((s$15@definition * s$15@definition) > i$14@definition) or (((s$15@definition + 1@definition) * (s$15@definition + 1@definition)) <= i$14@definition))) @2.2
Non Self Referring Assignment Scheme at command 2.1

130. (((s$15@definition * s$15@definition) > i$14@definition) == False) @2.2
Irrelevant Variable Assignment Scheme for precondition 127 at command 2.1

131. ((((s$15@definition + 1@solver) * (s$15@definition + 1@solver)) <= i$14@definition) == False) @2.2
Irrelevant Variable Assignment Scheme for precondition 128 at command 2.1

132. (result$13@definition == (False or (((s$15@definition + 1@definition) * (s$15@definition + 1@definition)) <= i$14@definition))) @2.2
From 129 replacing by 130 at position 0

133. ((1@definition == 1@solver) == True)
Compute (1@definition == 1@solver)

134. (1@definition == 1@solver)
From 133 by converting boolean expression equality to logical sentence

135. ((((s$15@definition + 1@definition) * (s$15@definition + 1@solver)) <= i$14@definition) == False) @2.2
From 131 replacing by 134 at position 0

136. ((((s$15@definition + 1@definition) * (s$15@definition + 1@definition)) <= i$14@definition) == False) @2.2
From 135 replacing by 134 at position 0

137. (result$13@definition == (False or False)) @2.2
From 132 replacing by 136 at position 0

138. ((False or False) == False)
Compute (False or False)

139. (result$13@definition == False) @2.2
From 137 replacing by 138 at position 0

140. (result$13@definition == False) @1.2

IfScheme at 1.1 for common condition 139, 29