university of groningen

faculty of science and engineering

Bachelor Thesis in Computing Science

Hoare Logics for Skeletal Semantics

Bachelor Candidate

Laura-Andrea Schimbător

Student ID 4103580

Supervisor

Dr. Dan Frumin

University of Groningen

Co-supervisor

Prof. Dr. Jorge Pérez

University of Groningen

Academic Year

2021/2022

Abstract

Proving the correctness of programs is essential for programming languages. A widely used approach for proving the partial correctness of programs is Hoare logic, a concrete formal system using axioms and rules. It is sound and complete, and has since been adapted for numerous languages. A drawback of the system is its low modularity, meaning any alteration of the semantics influences the soundness of the Hoare logic. This implies that a separate Hoare logic must be derived and proved for each individual programming language.

A modular way for representing programming languages is skeletal semantics. It was introduced as a meta-language for describing languages specified from natural semantics and has been used to derive correct-by-construction interpreters. In this project we provide a comprehensive presentation of Hoare logic and skeletal semantics. We introduce an imperative language WHILE, and prove the soundness property of its skeletal semantics with respect to Hoare logic. Lastly, we make the first step towards a general way to describe correct-by-construction Hoare logic from skeletal semantics by providing a heuristic for mapping skeletons to Hoare logic rules.

Contents

List of Tables

1 Introduction

Programming languages are the building blocks of applied computer science. Although only part of them are widely used, hundreds of programming languages exist [2]. Naturally, as building blocks, we want to have the certainty they are reliable enough to build on top of them. Therefore, for every programming language, there is a need for formal verification as a means to prove the correctness of programs. In order to ensure the reliable behaviour of a program, we ought to define its specifications and develop a thorough method to prove they are satisfied [4].

One of the most used formal approaches for proving soundness with respect to a certain semantics is Hoare logic, an axiomatic formal system. However, these proofs have little modularity, meaning that even small changes in the semantics of a language affect the soundness of the Hoare logic. Thus, it is necessary to develop a separate Hoare logic for each programming language, which can become a laborious task.

Skeletal semantics is a concept that has been recently introduced in the field of programming languages semantics [3]. It represents a framework for defining languages specified in natural semantics. It uses a modular approach, meaning some common structures are shared by the languages formalized in skeletal semantics, and only the language-specific atoms must be defined for each language. Although not yet extensively researched, skeletal semantics has tremendous potential for automatic derivations. Currently, the framework allows correct-by-construction interpreters to be instantiated for programming languages described

using skeletal semantics [1].

The previously mentioned considerations represent the foundation of our motivation. We believe the process of describing Hoare-style logic can be automated to some extent and generalised, as long as the targeted programming languages are formalised consistently. More specifically, we propose to investigate the formal way to describe a Hoare-style logic for languages defined using skeletal semantics, and examine the possibility of deriving correct-by-construction Hoare logic for these languages. We investigate this possibility by first proving the soundness of a simple imperative language called WHILE. We then present a heuristic for mapping skeletal semantics constructs to Hoare-style logic rules. We expect our research will prompt the interest of the program verification community and will direct their attention to the possibilities offered by skeletal semantics with regard to the provable correctness of programs.

In this project, we first explore the related work and provide the necessary background information on Hoare logic and skeletal semantics in Chapter 2. In Chapter 3, we introduce the syntax and semantics of the imperative language WHILE in section 3.1. We then provide the formal definitions of filters in 3.2, before proceeding to proving the soundness of expressions in 3.3 and statements, in 3.4. In section 3.5, we extend the language with new rules, and in 3.6 we introduce a heuristic for mapping skeletal semantics to Hoare logic. Lastly, we analyze possible future work in Chapter 4.

2 Background

2.1 Hoare Logic

The objective of any program is to perform a particular function decided by the programmer. This function is called the behaviour of the program. There are two options to assess whether the program exhibits the expected behaviour. A program can possess the *partial correctness property*, which expresses that *if* the program terminates, a certain condition will be met in the final state. A program can also have the *total correctness property*, which adds to the notion of partial correctness the condition that the program *will* terminate [11].

Proving the correctness of programs is one of the most challenging and resource-consuming tasks in computing science [9]. A formal system for proving program correctness by means of rules and axioms was developed by Hoare [6]. This axiomatic system has revolutionised the discipline, prompting the development of numerous applications of Hoare logic in diverse fields: provable concurrency laws [7], correctness conditions for non-linearizable concurrent objects [13], verification-focused formal genetic programming systems [5], information flow control [10]. Since the interest in the topic grew, attempts at automating the process of proving program correctness have been made, but mostly separately for each language [8, 12], due to their lack of modularity.

The most important characteristic of the system is the Hoare triple. It has the form: $\{P\}\ S\ \{Q\}$, where P is called the precondition, Q - the postcondition, and

S is the command. P and Q are also called assertions or predicates. The triple expresses that if the precondition is met, the postcondition will be satisfied after executing the command. Hoare triples can be applied to prove only the partial correctness of programs.

The system contains rules and axioms. Both contain a precondition, command, and postcondition. The difference between them is that rules have premises, while axioms don't. Equation 2.1 shows the *skip* axiom, while 2.2 is the rule for *composition*. The two premises of the rule are above the line: $\{P\}\ S_1\ \{Q\}$ and $\{Q\}\ S_2\ \{R\}$. For each rule, if the premises are met, executing the command in an initial state in which the precondition holds results in a state in which the postcondition is met.

$$\{P\}\ \text{skip}\ \{P\} \tag{2.1}$$

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}} \tag{2.2}$$

2.2 Skeletal Semantics

Hoare logic is an example of a *semantics*. The semantics of a programming language is concerned with the meaning, or behaviour, of grammatically correct programs, as opposed to *syntax*, which represents the grammatical structure of programs [11]. The most important semantics for programming languages are *operational semantics*, *denotational semantics*, and *axiomatic semantics* - represented by Hoare logic.

In this project, we are particularly interested in operational semantics, which describes the meaning of programs as executions of sequences of steps. It can be further divided into *natural semantics*, describing how the overall results of computations are obtained; and *structural operational semantics*, describing how the individual steps of executions are evaluated [11]. If any partial correctness property that can be proved using a certain inference system also holds according to the semantics, the semantics is said to be *sound*. Using the Hoare logic system, we want to prove this property for *skeletal semantics*.

Skeletal semantics is a meta-language used for describing languages specified from natural semantics [3]. It has been introduced as a solution for systematically constructing both concrete and abstract semantics, while proving general consistency results. The semantics proposes the notion of *skeletons*, which describe the behaviour of reusable language constructs. Based on them, the generic *interpretations* derive semantic judgements. Skeletal semantics can be interpreted using non-deterministic and deterministic abstract machines [1]. In addition, for any language, a certified OCaml interpreter can be instantiated [1].

Recently, a language-independent framework has been proposed as a solution for automatically generating correct-by-construction program verifiers for operational semantics [14]. It has been tested on real-world languages (C, Java, and JavaScript) and proved their functional correctness. Both interpreters and verifiers are exceptionally useful, and the groundwork for automatically deriving them has already been laid. However, a similarly challenging task - automatically deriving a Hoare-style logic for languages formalised in skeletal semantics, has not yet been investigated.

Skeletal semantics uses *terms*, which can either base terms - corresponding to basic blocks of syntax, term variables, or constructors applied to terms. Each term has its own *sort*, representing its "type", which can be a *base sort*, for base terms, or a *program sort*, otherwise. We assume a set of term variables and a set of constructors, ranged over by x_t and c, respectively.

Term variables together with *flow variables* form the set of *skeletal variables*. The purpose of flow variables is to hold semantic values, such as states and intermediate values. Similarly to terms, flow variables have *flow sorts*. We write $t : s$ or $Sort(t) = s$ to denote that s is the base sort of the base term t, and $v : s$ or $Sort(v) = s$ to denote that value v has the flow sort s. For a program sort s, we write $in(s)$ for its input flow sort and $out(s)$ for its output flow sort.

The semantics of each constructor c is given by a *skeleton*. A skeleton has the form $\textsc{Name}(c(x_{t_1}..x_{t_n})) := S$, where Name is the name of the skeleton, $x_{t_1}..x_{t_n}$ are term variables and S is the *skeleton body*. Formula 2.3 is an example of a skeleton defining the If statement of an imperative language. Here $x_{t_1}, x_{t_2}, x_{t_3}$ are term variables, while all other variables are flow variables. Among these, we notice x_σ

- the input state, and x_o - the output state of the skeleton. The skeleton name is IF, and the skeleton body is contained in the square brackets.

$$\text{IF}(if\ x_{t_1}\ x_{t_2}\ x_{t_3}) := \left[H(x_\sigma, x_{t_1}, x_{f_1}); \text{isBool}(x_{f_1})? \triangleright x_{f_{1'}}; \begin{pmatrix} \text{isTrue}(x_{f_{1'}}); H(x_\sigma, x_{t_2}, x_o) \\ \text{isFalse}(x_{f_{1'}}); H(x_\sigma, x_{t_3}, x_o) \end{pmatrix}_{\{x_o\}} \right] \tag{2.3}$$

The body of a skeleton is composed of *bones*, which can be either *hooks*, *filters*, or *sets of branches*. *Hook judgements* represent required subcomputations, or recursion. They are built from an input flow variable, a term to be hooked, and an output flow variable. For example in 2.3, hook $H(x_\sigma, x_{t_1}, x_{f_1})$ has input x_σ - the state of the program, term x_{t_1}, and output x_{f_1}. During the interpretation of the hook, the constructor of term x_{t_1} will be applied with the input state bound to x_σ, and the output state will be bound to x_{f_1}.

The *filters* in skeletal semantics have the form $\text{F}(x_1..x_n)? \triangleright (y_1..y_m)$, where F is the filter name, x and y range over skeletal variables, with input variables $x_1..x_n$ and output variables $y_1..y_m$. Filters test whether the values of the input variables satisfy a certain condition, and bind the result to the output variables. In 2.3, the $isBool$ filter checks if the value of x_{f_1} is a boolean, and binds the result to $x_{f_{1'}}$. A filter with no output variables, such as $isTrue$ in 2.3, acts like a predicate and is written $\text{F}(x_1..x_n)$.

A *set of branches* represents the set of possible pathways in the derivation when the control flow splits. Each branch is itself a sequence of bones. A set of branches has a set V of shared skeletal variables that must be defined by each branch, written to the right of the branching. In example 2.3, there is one set of branches with two pathways, depending on the value of $x_{f_{1'}}$. Here the set V has a single element - x_o.

There are three interpretations defined for skeletal semantics. The *well-formedness interpretation* ensures that all bones in the skeletons use terms of the correct sorts. The *concrete interpretation* always picks one branch from a set of branches, and the variables in V take the values of the variables in the branch. The *abstract interpretation* evaluates all possible paths in a branching, and if one or more branches

succeed, the variables in V are set to $\top$. The three interpretations of skeletal semantics are proven to be consistent [3].

The interpretations are represented by triple sets of the form *(state, term, result)*. For the concrete interpretation, the input state is a pair (Σ, T). Here Σ is an environment, mapping term variables to closed terms and flow variables to values, and T is a set of triples of value, closed term, and value, representing already known subderivations. Just like the environment Σ, the result of the interpretation maps term variables to closed terms and flow variables to values.

The set T is used in the interpretation of hooks. T is the input for the *immediate consequence* $\mathcal{H}$, which is a function from triple sets to triple sets, deriving new judgements based on already known ones. For a triple (σ, t, v) to be in $\mathcal{H}(T)$, a few conditions must be met. First, the term t must be a constructor applied to terms, such that $t = c(t_1..t_n)$ and $Sort(t) = s$, where s is the input sort for σ. Next, if the skeleton $\textsc{Name}(c(x_{t_1}..x_{t_n})) := S$ is evaluated in the initial environment $\Sigma = x_\sigma \rightarrow \sigma + x_{t_1} \rightarrow t_1 + .. + x_{t_n} \rightarrow t_n$, the output environment is Σ', which binds x_o to v. In other words, $[\![S]\!](\Sigma, T) \Downarrow \Sigma'$, and $\Sigma'(x_o) = v$.

In this context, the symbol $\Downarrow$ represents concrete semantics of the language. The concrete semantics $\Downarrow$ is defined to be the smallest fixpoint of $\mathcal{H}$, formally written $\Downarrow = \bigcup_n \mathcal{H}^n(\emptyset)$. The definition is derived using the Kleene fixpoint theorem, from the facts that function $\mathcal{H}$ is monotone and continuous [3].

In the process of deriving $\mathcal{H}(T)$, all known evaluations $(\sigma, t, v) \in T$ will be combined using the applicable constructors of the language to obtain new judgements. For a short running example, we assume we know the results of evaluating two arithmetic expressions in $T = \mathcal{H}^k(\emptyset)$, namely $(\sigma, a_1, v_1) \in T$ and $(\sigma, a_2, v_2) \in T$. Then, we will be able to derive new judgements based on the constructors in our language that require two input variables. If the language used has the rule for arithmetic expression addition, $a_1 + a_2$, then from $(\sigma, a_1, v_1) \in \mathcal{H}^k(\emptyset)$ and $(\sigma, a_2, v_2) \in \mathcal{H}^k(\emptyset)$, we can derive a new judgement for evaluating the sum of the expressions. The new triple, $(\sigma, a_1 + a_2, v_3)$, will be part of $\mathcal{H}(T)$, otherwise written as $(\sigma, a_1 + a_2, v_3) \in \mathcal{H}^{k+1}(\emptyset)$. Here, the value of v_3 will be the sum of the values of v_1 and v_2.

Similarly, we assume the rule for checking if two expressions are equal, $a_1 = a_2$, is present in our language. Then, from $(\sigma, a_1, v_1) \in \mathcal{H}^k(\emptyset)$ and $(\sigma, a_2, v_2) \in \mathcal{H}^k(\emptyset)$, we will be able to derive a new triple $(\sigma, a_1 = a_2, v_4) \in \mathcal{H}^{k+1}(\emptyset)$, where v_4 contains the truth value of the equality $v_1 = v_2$.

3 Analysis

This chapter represents the main part of our report. We first describe the simple imperative language While in 3.1. We provide the syntax in 3.1.1 and introduce functions defining the semantics of expressions in 3.1.2. In subsection 3.1.3, we present the Hoare logic and skeletal semantics, and in 3.2 we provide the formal definitions of the filters in the language. We then proceed by proving the soundness of the language While, first for expressions, in section 3.3, then for statements, in section 3.4. We follow by extending the language with two new rules in section 3.5, for which we also prove the soundness property. Lastly, in section 3.6, we describe a heuristic for mapping skeletons to Hoare logic rules.

3.1 Language While

We want to prove the soundness property for a simple imperative language called While. For this, we first describe the language by presenting its syntax, semantics and the corresponding Hoare logic rules.

3.1.1 Syntax

The language While contains expressions and statements. Expressions evaluate to values, while statements modify the state of the program. We have the following categories:

n – ranging over integers, $\mathbb{Z}$

x – ranging over arithmetic variables

y – ranging over boolean variables

a – ranging over arithmetic expressions

b – ranging over boolean expressions

S – ranging over statements

The syntax of the language WHILE is then:

$$
\begin{aligned}
a &::== n \mid x \mid a_1 + a_2 \\
b &::== \text{true} \mid \text{false} \mid y \mid b_1 = b_2 \mid \neg b_1 \\
S &::== \text{skip} \mid x := a \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S
\end{aligned}
$$

3.1.2 SEMANTICS OF EXPRESSIONS

Now that we know the syntax of the language, we proceed by presenting its semantics. First, we want to provide the meaning of arithmetical and boolean expressions. These expressions will always be evaluated in a certain state of the program. This means that the state has to be an input parameter of the semantic functions of expressions.

It is also important to mention the sorts in the WHILE language. The base sorts are *ident* for identifiers (strings) and *lit* for literals (integers). The program sorts are *expr* for expressions and *stat* for statements. The flow sorts are *val* for values, *int* for integers, *bool* for booleans, and *store* for variable store, which is a function from strings to *val*. When evaluating expressions, the output flow sort is *val*, while the output flow sort of statements is *store* [3]. Here *val* is instantiated with the disjoint union *int+bool*. For example, if $n_1 : int$ and $b_1 : bool$, then the corresponding values $n_2 : val$ and $b_2 : val$ are $n_2 = int(n_1)$ and $b_2 = bool(b_2)$.

Function A

We introduce the semantic function $\mathcal{A}$: **(expr** $\times$ **state)** $\rightarrow$ **int**, defining the meaning of arithmetic expressions:

$$\begin{aligned} \mathcal{A}[\![n]\!](\sigma) &= n \quad \text{if } n \in Z \\ \mathcal{A}[\![x]\!](\sigma) &= n \quad \text{if } \sigma(x) = int(n),\ n \in Z \\ \mathcal{A}[\![a_1 + a_2]\!](\sigma) &= \mathcal{A}[\![a_1]\!](\sigma) + \mathcal{A}[\![a_2]\!](\sigma) \end{aligned}$$

Function $\mathcal{A}$ is not total, as it is not defined for boolean expressions. The function can take the arguments one at a time. If we only provide the first parameter - the expression, we obtain a function with one parameter, the state. The semantic function evaluates arithmetic expressions in a certain state, and returns its integer value.

Function B

We introduce the semantic function $\mathcal{B}$: **(expr** $\times$ **state)** $\rightarrow$ **bool**, defining the meaning of boolean expressions. The function evaluates a boolean expression in a particular state and returns its boolean value:

$$\mathcal{B}[\![x]\!](\sigma) = b \quad \text{if } \sigma(x) = bool(b),\ b \in \{\text{true}, \text{false}\}$$

$$\mathcal{B}[\![a_1 = a_2]\!](\sigma) = \begin{cases} \text{true} & \text{if } \mathcal{A}[\![a_1]\!](\sigma) = \mathcal{A}[\![a_2]\!](\sigma) \\ \text{false} & \text{if } \mathcal{A}[\![a_1]\!](\sigma) \neq \mathcal{A}[\![a_2]\!](\sigma) \end{cases}$$

$$\mathcal{B}[\![\neg b]\!](\sigma) = \begin{cases} \text{true} & \text{if } \mathcal{B}[\![b]\!](\sigma) = \text{false} \\ \text{false} & \text{if } \mathcal{B}[\![b]\!](\sigma) = \text{true} \end{cases}$$

Similarly to function $\mathcal{A}$, function $\mathcal{B}$ is not total, as it is not defined for arithmetic expressions. The function also doesn't require both parameters, and providing only the expression leads to function $\mathcal{B}[\![b]\!]$: **state** $\rightarrow$ **bool**.

Function E

Lastly, we introduce the semantic function $\mathcal{E} : (\textbf{expr} \times \textbf{state}) \to \textbf{val}$:

$$\mathcal{E}[\![x]\!](\sigma) = \begin{cases} int(\mathcal{A}[\![x]\!](\sigma)) & \text{if } \mathcal{A}[\![x]\!](\sigma) \text{ is defined} \\ bool(\mathcal{B}[\![x]\!](\sigma)) & \text{otherwise} \end{cases}$$

Unlike functions $\mathcal{A}$ and $\mathcal{B}$, function $\mathcal{E}$ is total on expressions, as it is defined for both arithmetic and boolean expressions. Similarly to $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{E}$ becomes a function on states if we only provide the first argument. Function $\mathcal{E}$ has the purpose of transforming expressions to val outputs, so it transforms the results of functions $\mathcal{A}$ and $\mathcal{B}$ using val wrappers.

3.1.3 Definitions

The functions we defined in 3.1.2 are an important prerequisite for the Hoare logic of the language. In this subsection we first introduce a notation for predicates, which we then use in the Hoare logic rules of While. Lastly, we provide the skeletal semantics for the language, for which we will prove the soundness property in future sections.

Predicates

We introduce the following notation for reasoning on predicates:

$P_1 \wedge P_2$	for P where $P(\sigma) = P_1(\sigma)$ and $P_2(\sigma)$
$\neg P$	for P' where $P'(\sigma) = \neg P(\sigma)$
$P[x \mapsto \mathcal{E}[\![a]\!]]$	for P' where $P'(\sigma) = P(\sigma[x \mapsto \mathcal{E}[\![a]\!](\sigma)])$

When the predicate is of the form $\mathcal{B}[\![b]\!]$, we say the predicate is true if function $\mathcal{B}$ is defined on expressions b, and the result of the evaluation is $true$.

Hoare Logic

The Hoare logic for the language While [11] is given in Table 3.1.

[asn]	$\{P[x \mapsto \mathcal{E}[\![a]\!]]\}\ x := a\ \{P\}$
[skip]	$\{P\}$ skip $\{P\}$
[comp]	$\dfrac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$
[if]	$\dfrac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S_1\ \{Q\} \qquad \{\neg\mathcal{B}[\![b]\!] \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{Q\}}$
[while]	$\dfrac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S\ \{P\}}{\{P\}\ \text{while } b \text{ do } S\ \{\neg\mathcal{B}[\![b]\!] \wedge P\}}$
[cons]	$\dfrac{\{P'\}\ S\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$

Table 3.1: Hoare Logics While

Skeletal Semantics

The Skeletal Semantics of the language [3] is presented in Table 3.2. Each language construct is represented by one skeleton.

3.2 Formal Definitions of Filters

Bodin et al. [3] describe the sorts of filters in the language While. They do not, however, provide their definitions. While hooks and branching sets do not depend on the programming language, the filters in the skeletons are language-specific atoms. Filters have the role of evaluating relations between variables and bind new results to variables, so in order to complete derivations in skeletal semantics, we require their definition. Thus, we will provide the formal definitions of filters in Table 3.3. Special attention will be drawn to predicates $isTrue$ and $isFalse$. They will return () if their input has the right value, i.e. the input of $isTrue$ is **true** and the input of $isFalse$ is **false**. Otherwise, they will not return, not allowing the execution to continue.

$$\textsc{Lit}(const(x_t)) := \left[\text{litInt}(x_t)? \triangleright x_{f_1}; \text{intVal}(x_{f_1})? \triangleright x_o\right]$$

$$\textsc{Var}(var(x_t)) := \left[\text{read}(x_t, x_\sigma)? \triangleright x_o\right]$$

$$\textsc{Add}(x_{t_1} + x_{t_2}) := \begin{bmatrix} H(x_\sigma, x_{t_1}, x_{f_1}); \text{isInt}(x_{f_1})? \triangleright x_{f_1'}; H(x_\sigma, x_{t_2}, x_{f_2}); \\ \text{isInt}(x_{f_2})? \triangleright x_{f_2'}; \text{add}(x_{f_1'}, x_{f_2'})? \triangleright x_{f_3}; \text{intVal}(x_{f_3})? \triangleright x_o \end{bmatrix}$$

$$\textsc{Eq}(x_{t_1} = x_{t_2}) := \begin{bmatrix} H(x_\sigma, x_{t_1}, x_{f_1}); \text{isInt}(x_{f_1})? \triangleright x_{f_1'}; H(x_\sigma, x_{t_2}, x_{f_2}); \\ \text{isInt}(x_{f_2})? \triangleright x_{f_2'}; \text{eq}(x_{f_1'}, x_{f_2'})? \triangleright x_{f_3}; \text{boolVal}(x_{f_3})? \triangleright x_o \end{bmatrix}$$

$$\textsc{Neg}(\neg x_t) := \left[H(x_\sigma, x_t, x_{f_1}); \text{isBool}(x_{f_1})? \triangleright x_{f_2}; \text{neg}(x_{f_2})? \triangleright x_{f_3}; \text{boolVal}(x_{f_3})? \triangleright x_o\right]$$

$$\textsc{Skip}(skip) := \left[\text{id}(x_\sigma)? \triangleright x_o\right]$$

$$\textsc{Asn}(x_{t_1} := x_{t_2}) := \left[H(x_\sigma, x_{t_2}, x_{f_1});\ \text{write}(x_{t_1}, x_\sigma, x_{f_1})? \triangleright x_o\right]$$

$$\textsc{Seq}(x_{t_1}; x_{t_2}) := \left[H(x_\sigma, x_{t_1}, x_{f_1}); H(x_{f_1}, x_{t_2}, x_o)\right]$$

$$\textsc{If}(if\ x_{t_1}\ x_{t_2}\ x_{t_3}) := \left[H(x_\sigma, x_{t_1}, x_{f_1}); \text{isBool}(x_{f_1})? \triangleright x_{f_{1'}}; \begin{pmatrix} \text{isTrue}(x_{f_{1'}}); H(x_\sigma, x_{t_2}, x_o) \\ \text{isFalse}(x_{f_{1'}}); H(x_\sigma, x_{t_3}, x_o) \end{pmatrix}_{\{x_o\}}\right]$$

$$\textsc{While}(while\ x_{t_1}\ x_{t_2}) := \left[\begin{array}{l} H(x_\sigma, x_{t_1}, x_{f_1}); \text{isBool}(x_{f_1})? \triangleright x_{f_{1'}}; \\ \begin{pmatrix} \text{isTrue}(x_{f_{1'}}); H(x_\sigma, x_{t_2}, x_{f_2}); H(x_{f_2}, while\ x_{t_1}\ x_{t_2}, x_o) \\ \text{isFalse}(x_{f_{1'}}); id(x_\sigma)? \triangleright x_o \end{pmatrix}_{\{x_o\}} \end{array}\right]$$

Table 3.2: Skeletal Semantics While

3.3 Soundness of Expressions

We now have all the necessary information for proving the soundness property for the skeletons that evaluate expressions. We do this by introducing two Lemmas, one for arithmetic and one for boolean expressions.

3.3.1 Soundness of Arithmetic Expressions

We will first prove the partial correctness of arithmetic expressions with respect to function $\mathcal{A}$. We will do this by induction.

Lemma 1: If $(\sigma, e, int(v)) \in \Downarrow$, where e is an expression, then $\mathcal{A}[\![e]\!](\sigma) = v$.

Proof:

Filter	Definition
litInt	$\{(lit(x) \rightarrow x) \mid x : int\}$
intVal	$\{(x \rightarrow int(x)) \mid x : int\}$
isInt	$\{(int(x) \rightarrow x) \mid x : int\}$
add	$\{((x, y) \rightarrow (x +_{int} y)) \mid x, y : int\}$
boolVal	$\{(true \rightarrow bool(true)),$ $(false \rightarrow bool(false))\}$
isBool	$\{(bool(x) \rightarrow x) \mid x : bool\}$
isTrue	$\{true \rightarrow ()\}$
isFalse	$\{false \rightarrow ()\}$
eq	$\{((x, y) \rightarrow true) \mid x, y : int, x =_{int} y\} \cup$ $\{((x, y) \rightarrow false) \mid x, y : int, x \neq_{int} y\}$
neg	$\{(true \rightarrow false), (false \rightarrow true)\}$
read	$\{(id, st) \rightarrow st(id) \mid st : store, id : ident\}$
write	$\{(id, st, v) \rightarrow st' \mid id : ident, st : store, v : val,$ $st' : store, st'(id) = v, \forall id'.id' \neq id \Rightarrow st'(id') = st(id')\}$
id	$\{st \rightarrow st \mid st : store\}$

Table 3.3: Formal Filter Definitions

1. The first base case is $e = const(n_1)$ for $n_1 = lit(n)$, $n \in \mathbb{Z}$: For skeletal semantics we have skeleton:

$$\textsc{Lit}(const(x_t)) := \left[\text{litInt}(x_t)? \triangleright x_{f_1}; \text{intVal}(x_{f_1})? \triangleright x_o\right]$$

We examine the skeleton, starting with initial environment $\Sigma = x_\sigma \rightarrow \sigma, x_t \rightarrow n_1$, with σ - the initial state. Assume $T = \mathcal{H}^n(\emptyset)$ for some n, such that $[\![\text{litInt}(x_t)? \triangleright x_{f_1}]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, with $\Sigma^1 = \Sigma + x_{f_1} \rightarrow v$, where $[\![\text{litInt}]\!](\Sigma(x_t)) \Downarrow v$, or $[\![\text{litInt}]\!](n_1) \Downarrow v$. Since $n_1 = lit(n)$, we can conclude $n = v$.

Next, we have filter: $[\![\text{intVal}(x_{f_1})? \triangleright x_o]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$, where $[\![\text{intVal}]\!](v) \Downarrow v'$ and $\Sigma^2 = \Sigma^1 + x_o \rightarrow v'$. Lastly, the empty skeleton returns its environment, Σ^2. By the definition of $intVal$, v' will be $int(v)$, the same as $int(n)$. Thus, we can apply the definition for $\mathcal{A}$ and conclude $\mathcal{A}[\![n]\!](\sigma) = n$.

2. The second base case is $e = x$, where x is an identifier: In this case for skeletal semantics we have skeleton:

$$\textsc{Var}(var(x_t)) := [\text{read}(x_t, x_\sigma)? \triangleright x_o]$$

We analyse the filter comprising the skeleton. Let the initial environment be $\Sigma = x_\sigma \rightarrow \sigma + x_t \rightarrow x$, where σ is the initial state. We assume $T = \mathcal{H}^n(\emptyset)$ for some n, such that $[\![\text{read}(x_t, x_\sigma)?\triangleright x_o]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, with $\Sigma^1 = \Sigma + x_o \rightarrow v$, where $[\![\text{read}]\!](\Sigma(x_t), \Sigma(x_\sigma)) \Downarrow v$, or $[\![\text{read}]\!](x, \sigma) \Downarrow v$, and $v = \sigma(x)$. Next, the empty skeleton returns its environment, Σ^1, ending the derivation.

$\sigma(x)$ returns val, or in other words, $int(n)$ or $bool(b)$, for $n : int$ and $b : bool$. We will examine the first case here, and the second one in the proof for **Lemma 2**. We assume $\sigma(x) = int(n)$. We can then directly apply the definition of $\mathcal{A}$, and conclude $\mathcal{A}[\![x]\!](\sigma) = n$.

3. $e = (a_1 + a_2)$: We have the corresponding skeleton:

$$\textsc{Add}(x_{t_1} + x_{t_2}) := \begin{bmatrix} H(x_\sigma, x_{t_1}, x_{f_1}); \text{isInt}(x_{f_1})? \triangleright x_{f_{1'}}; H(x_\sigma, x_{t_2}, x_{f_2}); \\ \text{isInt}(x_{f_2})? \triangleright x_{f_{2'}}; \text{add}(x_{f_{1'}}, x_{f_{2'}})? \triangleright x_{f_3}; \text{intVal}(x_{f_3})? \triangleright x_o \end{bmatrix}$$

If we use the definition of $\mathcal{A}$, we see that $\mathcal{A}[\![a_1 + a_2]\!](\sigma) = \mathcal{A}[\![a_1]\!](\sigma) + \mathcal{A}[\![a_2]\!](\sigma)$. We introduce our induction hypothesis. We assume

$$(\sigma, a_1, int(k_1)) \in \mathcal{H}^n(\emptyset) \implies \mathcal{A}[\![a_1]\!](\sigma) = k_1$$

and

$$(\sigma, a_2, int(k_2)) \in \mathcal{H}^n(\emptyset) \implies \mathcal{A}[\![a_1]\!](\sigma) = k_2$$

for some n. Now we want to show that

$$(\sigma, a_1 + a_2, int(k)) \in \mathcal{H}^{n+1}(\emptyset) \implies \mathcal{A}[\![a_1 + a_2]\!](\sigma) = k$$

We start to evaluate the skeleton. We take the initial environment $\Sigma = x_\sigma \rightarrow \sigma + x_{t_1} \rightarrow a_1 + x_{t_2} \rightarrow a_2$.

- First, we have a hook. Assume there exists $T = \mathcal{H}^n(\emptyset)$ such that $[\![H(x_\sigma, x_{t_1}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_{t_1}), v_1) = (\sigma, a_1, v_1) \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \rightarrow v_1$. This means that $\sigma(a_1) = v_1$.
- Next we have $[\![\text{isInt}(x_{f_1})?\triangleright x_{f_{1'}}]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$ with $\Sigma^2 = \Sigma^1 + x_{f_{1'}} \rightarrow v_2$, where $[\![\text{isInt}]\!]\Sigma^1(x_{f_1}) \Downarrow v_2$. Since $\Sigma^1(x_{f_1}) = v_1$, we have $[\![\text{isInt}]\!]v_1 \Downarrow v_2$. By the definition of $isInt$, this means that $v_1 = int(v_2)$. We can thus apply the induction hypothesis and conclude $\mathcal{A}[\![a_1]\!](\sigma) = v_2$.
- For the next hook, we have $[\![H(x_\sigma, x_{t_2}, x_{f_2})]\!](\Sigma^2, T) \Downarrow (\Sigma^3, T)$, where $(\Sigma^2(x_\sigma), \Sigma^2(x_{t_2}), v_3) = (\sigma, a_2, v_3) \in T$ and $\Sigma^3 = \Sigma^2 + x_{f_2} \rightarrow v_3$. Note that $\Sigma(x_\sigma) = \Sigma^2(x_\sigma)$, $\Sigma(x_{t_2}) = \Sigma^2(x_{t_2})$, and $\sigma(a_2) = v_3$.
- We then have again a filter: $[\![\text{isInt}(x_{f_2})? \triangleright x_{f_{2'}}]\!](\Sigma^3, T) \Downarrow (\Sigma^4, T)$ with $\Sigma^4 = \Sigma^3 + x_{f_{2'}} \rightarrow v_4$, where $[\![\text{isInt}]\!]\Sigma^3(x_{f_2}) \Downarrow v_4$. Since $\Sigma^3(x_{f_2}) = v_3$, we have $[\![\text{isInt}]\!]v_3 \Downarrow v_4$. By the definition of $isInt$, we see that $v_3 = int(v_4)$. We can then apply the induction hypothesis again, and say that $\mathcal{A}[\![a_2]\!](\sigma) = v_4$.

- The next filter we have, $[\![\text{add}(x_{f_1'}, x_{f_2'})? \triangleright x_{f_3}]\!](\Sigma^4, T) \Downarrow (\Sigma^5, T)$ will define $\Sigma^5 = \Sigma^4 + x_{f_3} \rightarrow v_5$, where $[\![\text{add}]\!](\Sigma^4(x_{f_1'}), \Sigma^4(x_{f_2'})) \Downarrow v_5$. We note that v_5 will be $\Sigma^4(x_{f_1'}) + \Sigma^4(x_{f_2'}) = v_2 + v_4$, or in other words, $\mathcal{A}[\![a_1]\!](\sigma) + \mathcal{A}[\![a_2]\!](\sigma)$.
- Our last bone is the $intVal$ filter. We evaluate $[\![\text{intVal}(x_{f_3})? \triangleright x_o]\!](\Sigma^5, T) \Downarrow (\Sigma^6, T)$, with $\Sigma^6 = \Sigma^5 + x_o \rightarrow v_6$, where $[\![\text{intVal}]\!]\Sigma^5(x_{f_3}) \Downarrow v_6$. Since $\Sigma^5(x_{f_3}) = v_5$, we get $[\![\text{intVal}]\!]v_5 \Downarrow v_6$. Looking at the definition of $intVal$, we see that $v_6 = int(v_5)$.
- Lastly, we have the empty skeleton, which simply returns its environment: $[\![]\!](\Sigma^6, \emptyset) \Downarrow \Sigma^6$, marking the end of the derivation. We conclude $(\sigma, x_{t_1} + x_{t_2}, v_6) \in \mathcal{H}^{n+1}(\emptyset)$. But since v_5 is $\mathcal{A}[\![a_1]\!](\sigma) + \mathcal{A}[\![a_2]\!](\sigma)$ and $v_6 = int(v_5)$, we conclude $\mathcal{A}[\![a_1 + a_2]\!](\sigma) = v_5$.

3.3.2 Soundness of Boolean Expressions

Here we prove by induction the partial correctness of boolean expressions with respect to function $\mathcal{B}$.

Lemma 2: If $(\sigma, e, bool(v)) \in \Downarrow$, where e is an expression, then $\mathcal{B}[\![e]\!](\sigma) = v$.

Proof:

1. The first base case is $e = x$, when x is an identifier. In this case for skeletal semantics we have skeleton:

$$\textsc{Var}(var(x_t)) := [\text{read}(x_t, x_\sigma)? \triangleright x_o]$$

We analyse the filter comprising the skeleton. Let the initial environment be $\Sigma = x_\sigma \rightarrow \sigma + x_t \rightarrow x$, where σ is the initial state. We assume $T = \mathcal{H}^n(\emptyset)$ for some n, such that $[\![\text{read}(x_t, x_\sigma)? \triangleright x_o]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, with $\Sigma^1 = \Sigma + x_o \rightarrow v$, where $[\![\text{read}]\!](\Sigma(x_t), \Sigma(x_\sigma)) \Downarrow v$, or $[\![\text{read}]\!](x, \sigma) \Downarrow v$, and $v = \sigma(x)$. Next, the empty skeleton returns its environment, Σ^1, ending the derivation.

$\sigma(x)$ returns val, or in other words, $int(n)$ or $bool(b)$, for $n : int$ and $b : bool$. We will examine the second case here, as we already addressed the first one in the proof for **Lemma 1**. We assume $\sigma(x) = bool(b)$. We can then directly apply the definition of $\mathcal{B}$, and conclude $\mathcal{B}[\![x]\!](\sigma) = b$.

2. The second base case is $e = (a_1 = a_2)$. We have the skeleton:

$$\textsc{Eq}(x_{t_1} = x_{t_2}) := \begin{bmatrix} H(x_\sigma, x_{t_1}, x_{f_1}); \text{isInt}(x_{f_1})? \triangleright x_{f_1'}; H(x_\sigma, x_{t_2}, x_{f_2}); \\ \text{isInt}(x_{f_2})? \triangleright x_{f_2'}; \text{eq}(x_{f_1'}, x_{f_2'})? \triangleright x_{f_3}; \text{boolVal}(x_{f_3})? \triangleright x_o \end{bmatrix}$$

If we use the definition of $\mathcal{B}$, we see that

$$\mathcal{B}[\![a_1 = a_2]\!](\sigma) = \begin{cases} \text{true} & \text{if } \mathcal{A}[\![a_1]\!](\sigma) = \mathcal{A}[\![a_2]\!](\sigma) \\ \text{false} & \text{if } \mathcal{A}[\![a_1]\!](\sigma) \neq \mathcal{A}[\![a_2]\!](\sigma) \end{cases}$$

So if the evaluation of $\mathcal{A}[\![a_1]\!](\sigma)$ is equal to that of $\mathcal{A}[\![a_2]\!](\sigma)$, $\mathcal{B}[\![a_1 = a_2]\!](\sigma)$ will be true, and false otherwise. By **Lemma 1**, we know

$$(\sigma, a_1, int(k_1)) \in \mathcal{H}^n(\emptyset) \implies \mathcal{A}[\![a_1]\!](\sigma) = k_1$$

and

$$(\sigma, a_2, int(k_2)) \in \mathcal{H}^n(\emptyset) \implies \mathcal{A}[\![a_1]\!](\sigma) = k_2$$

for some n. Now we want to show that

$$(\sigma, \textsc{Eq}(a_1 = a_2), bool(k)) \in \mathcal{H}^{n+1}(\emptyset) \implies \mathcal{B}[\![a_1 = a_2]\!](\sigma) = k$$

We start to evaluate the skeleton. We take the initial environment for the derivation $\Sigma = x_\sigma \to \sigma + x_{t_1} \to a_1 + x_{t_2} \to a_2$.

- First, we have a hook. There exists $T = \mathcal{H}^n(\emptyset)$ such that $[\![H(x_\sigma, x_{t_1}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_{t_1}), v_1) = (\sigma, a_1, v_1) \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \to v_1$. This means that $\sigma(a_1) = v_1$.
- Next we have $[\![\text{isInt}(x_{f_1})?\triangleright x_{f_{1'}}]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$ with $\Sigma^2 = \Sigma^1 + x_{f_{1'}} \to v_2$, where $[\![\text{isInt}]\!]\Sigma^1(x_{f_1}) \Downarrow v_2$. Since $\Sigma^1(x_{f_1}) = v_1$, we have $[\![\text{isInt}]\!]\ v_1 \Downarrow v_2$. By the definition of $isInt$, $v_1 = int(v_2)$. We can apply **Lemma 1** and conclude $\mathcal{A}[\![a_1]\!](\sigma) = v_2$.
- For the next hook, we have $[\![H(x_\sigma, x_{t_2}, x_{f_2})]\!](\Sigma^2, T) \Downarrow (\Sigma^3, T)$, where $(\Sigma^2(x_\sigma), \Sigma^2(x_{t_2}), v_3) = (\sigma, a_2, v_3) \in T$ and $\Sigma^3 = \Sigma^2 + x_{f_2} \to v_3$. Note that $\Sigma(x_\sigma) = \Sigma^2(x_\sigma)$, $\Sigma(x_{t_2}) = \Sigma^2(x_{t_2})$, and $\sigma(a_2) = v_3$.
- We then have again a filter: $[\![\text{isInt}(x_{f_2})? \triangleright x_{f_{2'}}]\!](\Sigma^3, T) \Downarrow (\Sigma^4, T)$ with $\Sigma^4 = \Sigma^3 + x_{f_{2'}} \to v_4$, where $[\![\text{isInt}]\!]\Sigma^3(x_{f_2}) \Downarrow v_4$. Since $\Sigma^3(x_{f_2}) = v_3$, we have $[\![\text{isInt}]\!]v_3 \Downarrow v_4$. By the definition of $isInt$, we see that $v_3 = int(v_4)$. We can then apply **Lemma 1** again, and say that $\mathcal{A}[\![a_2]\!](\sigma) = v_4$.
- The next filter we have, $[\![\text{eq}(x_{f_{1'}}, x_{f_{2'}})? \triangleright x_{f_3}]\!](\Sigma^4, T) \Downarrow (\Sigma^5, T)$ will define $\Sigma^5 = \Sigma^4 + x_{f_3} \to v_5$, where $[\![\text{eq}]\!](\Sigma^4(x_{f_{1'}}), \Sigma^4(x_{f_{2'}})) \Downarrow v_5$. v_5 will be $true$ if $\Sigma^4(x_{f_{1'}})$ and $\Sigma^4(x_{f_{2'}})$ are equal, or $v_2 = v_4$, and $false$ otherwise. In other words, v_5 is $true$ if $\mathcal{A}[\![a_1]\!](\sigma) = \mathcal{A}[\![a_2]\!](\sigma)$ and $false$ otherwise.
- Our last bone is the $boolVal$ filter. We evaluate $[\![\text{boolVal}(x_{f_3})?\triangleright x_o]\!](\Sigma^5, T) \Downarrow (\Sigma^6, T)$, with $\Sigma^6 = \Sigma^5 + x_o \to v_6$, where $[\![\text{boolVal}]\!]\Sigma^5(x_{f_3}) \Downarrow v_6$. Since $\Sigma^5(x_{f_3}) = v_5$, we get $[\![\text{boolVal}]\!]v_5 \Downarrow v_6$. Looking at the definition of $boolVal$, we see that $v_6 = bool(v_5)$.
- Lastly, we have the empty skeleton, which simply returns its environment: $[\![]\!](\Sigma^6, \emptyset) \Downarrow \Sigma^6$, marking the end of the derivation. We conclude $(\sigma, \textsc{Eq}(x_{t_1} = x_{t_2}), v_6) \in \mathcal{H}^{n+1}(\emptyset)$. But since v_5 is $true$ if $\mathcal{A}[\![a_1]\!](\sigma) = \mathcal{A}[\![a_2]\!](\sigma)$ and $false$ otherwise, and $v_6 = bool(v_5)$ we conclude $\mathcal{B}[\![a_1 = a_2]\!](\sigma) = v_5$.

3. $e = (\neg b)$: We have the skeleton

$$\textsc{Neg}(\neg x_t) := \left[H(x_\sigma, x_t, x_{f_1}); \text{isBool}(x_{f_1})? \triangleright x_{f_2}; \text{neg}(x_{f_2})? \triangleright x_{f_3}; \text{boolVal}(x_{f_3})? \triangleright x_o\right]$$

Using the definition for $\mathcal{B}$:

$$\mathcal{B}[\![\neg b]\!](\sigma) = \begin{cases} \text{true} & \text{if } \mathcal{B}[\![b]\!](\sigma) = \text{false} \\ \text{false} & \text{if } \mathcal{B}[\![b]\!](\sigma) = \text{true} \end{cases}$$

So if the evaluation of $\mathcal{B}[\![b]\!](\sigma)$ is $false$, $\mathcal{B}[\![\neg b]\!](\sigma)$ will be $true$, and $true$ otherwise. We introduce our induction hypothesis: we assume

$$(\sigma, b, bool(k_1)) \in \mathcal{H}^n(\emptyset) \implies \mathcal{B}[\![b]\!](\sigma) = k_1$$

for some n. Now we want to show that

$$(\sigma, \textsc{Neg}(\neg b), bool(k_2)) \in \mathcal{H}^{n+1}(\emptyset) \implies \mathcal{B}[\![\neg b]\!](\sigma) = k_2$$

We start to evaluate the skeleton. We take the initial environment $\Sigma = x_\sigma \to \sigma + x_t \to b$.

- First, we have a hook. Assume there exists $T = \mathcal{H}^n(\emptyset)$ such that $[\![H(x_\sigma, x_t, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_t), v_1) = (\sigma, b, v_1) \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \to v_1$. This means that $\sigma(b) = v_1$.
- Next we have $[\![\text{isBool}(x_{f_1})? \triangleright x_{f_2}]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$ with $\Sigma^2 = \Sigma^1 + x_{f_2} \to v_2$, where $[\![\text{isBool}]\!]\Sigma^1(x_{f_1}) \Downarrow v_2$. Since $\Sigma^1(x_{f_1}) = v_1$, we have $[\![\text{isBool}]\!]v_1 \Downarrow v_2$. By the definition of $isBool$, $v_1 = bool(v_2)$. We can thus apply the induction hypothesis and say that $\mathcal{B}[\![b]\!](\sigma) = v_2$.
- The next filter we have, $[\![\text{neg}(x_{f_2})? \triangleright x_{f_3}]\!](\Sigma^2, T) \Downarrow (\Sigma^3, T)$ will define $\Sigma^3 = \Sigma^2 + x_{f_3} \to v_3$, where $[\![\text{neg}]\!]\Sigma^2(x_{f_2}) \Downarrow v_3$. v_3 will be $true$ if $\Sigma^2(x_{f_2})$, which is v_2, is $false$, or in other words, $\mathcal{B}[\![b]\!](\sigma) = false$, and v_3 will be false $false$ if $\mathcal{B}[\![b]\!](\sigma) = true$.
- Our last bone is the $boolVal$ filter. We evaluate $[\![\text{boolVal}(x_{f_3})? \triangleright x_o]\!](\Sigma^3, T) \Downarrow (\Sigma^4, T)$, with $\Sigma^4 = \Sigma^3 + x_o \to v_3$, where $[\![\text{boolVal}]\!]\Sigma^3(x_{f_3}) \Downarrow v_4$. Since $\Sigma^3(x_{f_3}) = v_3$, we get $[\![\text{boolVal}]\!]v_3 \Downarrow v_4$. By the definition of $boolVal$, we know $v_4 = bool(v_3)$.
- Lastly, we have the empty skeleton, which simply returns its environment: $[\![]\!](\Sigma^4, \emptyset) \Downarrow \Sigma^4$, marking the end of the derivation. We conclude $(\sigma, \textsc{Neg}(\neg x_t), v_4) \in \mathcal{H}^{n+1}(\emptyset)$. But since v_3 is $true$ if $\mathcal{B}[\![x_t]\!](\sigma) = false$ and $false$ otherwise, and $v_4 = bool(v_3)$, we conclude $\mathcal{B}[\![\neg x_t]\!](\sigma) = v_3$.

3.4 Soundness of Statements

We have proven the soundness property for skeletons evaluating expressions. Since expressions are needed for evaluating statements, we can now proceed with proving the property for skeletons evaluating statements. We define a partial correctness assertion $\{P\}S\{Q\}$ to be valid if and only if for all input states σ, if

$P(\sigma) = true$ and $(\sigma, S, \sigma') \in \Downarrow$ for some σ' then $Q(\sigma') = true$. We will next show the assertion is valid for all rules and axioms in Table 3.1.

Asn Axiom

Asn axiom:

$$\{P[x \mapsto \mathcal{E}[\![a]\!]]\}\ x := a\ \{P\}$$

Asn skeleton:

$$\textsc{Asn}(x_{t_1} := x_{t_2}) := \left[H(x_\sigma, x_{t_2}, x_{f_1});\ \text{write}(x_{t_1}, x_\sigma, x_{f_1})? \triangleright x_o\right]$$

Assume initial state σ such that $P[x \mapsto \mathcal{E}[\![a]\!]] = true$. In other words, $P(\sigma[x \mapsto \mathcal{E}[\![a]\!](\sigma)]) = true$. Take environment $\Sigma = x_\sigma \to \sigma + x_{t_1} \to x + x_{t_2} \to a$.

First, we have a hook. Assume there exists $T = \mathcal{H}^n(\emptyset)$ for some n such that $[\![H(x_\sigma, x_{t_2}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_{t_2}), v) = (\sigma, a, v) \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \to v$. It is clear that v is a value, since $\sigma(a) = v$. If a is an arithmetical expression, then v will be $int(\mathcal{A}[\![a]\!](\sigma))$, using **Lemma 1**. If a is a boolean expression, using **Lemma 2**, we know $v = bool(\mathcal{B}[\![a]\!](\sigma))$. Now applying the definition of $\mathcal{E}$, we see that $v = \mathcal{E}[\![a]\!](\sigma)$.

Next we have $[\![\text{write}(x_{t_1}, x_\sigma, x_{f_1})? \triangleright x_o]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$ with $\Sigma^2 = \Sigma^1 + x_o \to \sigma'$, where $[\![\text{write}]\!](\Sigma^1(x_{t_1}), \Sigma^1(x_\sigma), \Sigma^1(x_{f_1})) \Downarrow \sigma'$, or $[\![\text{write}]\!](x, \sigma, v) \Downarrow \sigma'$. From this, we know that $\sigma' = \sigma[x \mapsto v]$, or $\sigma[x \mapsto \mathcal{E}[\![a]\!](\sigma)]$. Lastly, the empty skeleton simply returns its environment, Σ^2. Now, we want to show that $P(\sigma') = true$. Since $P(\sigma') = P(\sigma[x \mapsto \mathcal{E}[\![a]\!](\sigma)])$, and our assumption was $P(\sigma[x \mapsto \mathcal{E}[\![a]\!](\sigma)]) = true$, we immediately see that $P(\sigma') = true$.

Skip Axiom

Skip axiom:

$$\{P\}\ \text{skip}\ \{P\}$$

Skip skeleton:

$$\textsc{Skip}(skip) := [\text{id}(x_\sigma)? \triangleright x_o]$$

Assume that $P(\sigma) = true$. Then there exists σ' such that $(\sigma, skip, \sigma') \in \mathcal{H}^n(\emptyset)$, for some n. Since there is no recursion in the rule (there are no hooks), we can take $n = 0$, such that $\mathcal{H}^n(\emptyset) = \emptyset$.

Take σ such that $P(\sigma) = true$ and $\Sigma = x_\sigma \to \sigma$. According to the concrete interpretation of filters, we have $[\![\mathrm{id}(x_\sigma)?\triangleright x_o]\!](\Sigma, \emptyset) \Downarrow (\Sigma^1, \emptyset)$, for $\Sigma^1 = \Sigma + x_o \to \sigma'$, where $[\![\mathrm{id}]\!]\Sigma(x_\sigma) \Downarrow \sigma'$. Now we have the empty skeleton, which simply returns its environment: $[\![]\!](\Sigma^1, \emptyset) \Downarrow \Sigma^1$, marking the end of the derivation. Using the definition for id, we see that $[\![\mathrm{id}]\!]\Sigma(x_\sigma) \Downarrow \Sigma(x_\sigma)$. But this means that $\Sigma^1(x_o) = \Sigma(x_\sigma) = \sigma$, so $\sigma = \sigma'$ and $\Sigma^1 = \Sigma + x_o \to \sigma$. Since we know $P(\sigma) = true$, and $\sigma = \sigma'$, we can conclude $P(\sigma') = true$ and $(\sigma, skip, \sigma) \in \mathcal{H}^1(\emptyset)$.

Comp Rule

Comp rule:

$$\frac{\{P\}S_1\{Q\} \qquad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

Seq skeleton:

$$\textsc{Seq}(x_{t_1}; x_{t_2}) := \left[H(x_\sigma, x_{t_1}, x_{f_1}); H(x_{f_1}, x_{t_2}, x_o)\right]$$

Assume initial state σ such that $P(\sigma) = true$. Take environment $\Sigma = x_\sigma \to \sigma + x_{t_1} \to S_1 + x_{t_2} \to S_2$. We start with the first hook. We assume $T = \mathcal{H}^n(\emptyset)$ exists for some n, such that $[\![H(x_\sigma, x_{t_1}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_{t_1}), \sigma') \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \to \sigma'$. By the first premise, it is true that $Q(\sigma') = true$.

Moving on to the second hook, we have $[\![H(x_{f_1}, x_{t_2}, x_o)]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$, where $(\Sigma^1(x_{f_1}), \Sigma^1(x_{t_2}), \sigma'') \in T$ and $\Sigma^2 = \Sigma^1 + x_{f_2} \to \sigma''$. Since $\Sigma^1(x_{f_1}) = \sigma'$ and $Q(\sigma') = true$, we use the second premise to conclude $R(\sigma'') = true$. Lastly, the empty skeleton returns its environment: $[\![]\!](\Sigma^2, T) \Downarrow \Sigma^2$. Since $P(\sigma) = true$, $R(\sigma'') = true$, and σ'' is the output state of the derivation, we can conclude that, given the premises hold, $\{P\}\ S_1; S_2\ \{R\}$ holds.

If Rule

If rule:

$$\frac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S_1\ \{Q\} \qquad \{\neg\mathcal{B}[\![b]\!] \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

If skeleton:

$$\text{IF}(if\ x_{t_1}\ x_{t_2}\ x_{t_3}) := \left[H(x_\sigma, x_{t_1}, x_{f_1}); \text{isBool}(x_{f_1})? \triangleright x_{f_{1'}}; \begin{pmatrix} \text{isTrue}(x_{f_{1'}}); H(x_\sigma, x_{t_2}, x_o) \\ \text{isFalse}(x_{f_{1'}}); H(x_\sigma, x_{t_3}, x_o) \end{pmatrix}_{\{x_o\}} \right]$$

Assume initial state σ such that $P(\sigma) = true$. Take environment $\Sigma = x_\sigma \to \sigma + x_{t_1} \to b + x_{t_2} \to S_1 + x_{t_3} \to S_2$. First, we have a hook. There exists $T = \mathcal{H}^n(\emptyset)$ for some n such that $[\![H(x_\sigma, x_{t_1}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_{t_1}), v_1) \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \to v_1$.

Next we have $[\![\text{isBool}(x_{f_1})? \triangleright x_{f_{1'}}]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$ with $\Sigma^2 = \Sigma^1 + x_{f_{1'}} \to v_2$, where $[\![\text{isBool}]\!]\Sigma^1(x_{f_1}) \Downarrow v_2$. Now, according to the definition of $isBool$, $x_{f_{1'}}$ will either be $true$ or $false$. This is the evaluation of x_{t_1} in skeletal semantics, so we choose to regard $\Sigma(x_{t_1})$ as b and $\Sigma^1(x_{f_{1'}})$ as $\mathcal{B}[\![b]\!]$, using **Lemma 2**.

Assuming $x_{f_{1'}}$ evaluates to $true$, meaning $isTrue$ should not block, we conclude $(\mathcal{B}[\![b]\!] \wedge P)(\sigma) = true$. According to the premises of the rule, this means that $\forall \sigma'$ such that $(\sigma, \Sigma^2(x_{t_2}), \sigma') \in T$, $Q(\sigma') = true$. We continue with the first branch, and evaluate $[\![\text{isTrue}(x_{f_{1'}})]\!](\Sigma^2, T) \Downarrow (\Sigma^2, T)$. The environment doesn't change, since $isTrue$ returns (), meaning no fresh flow variable is added. Now we perform $[\![H(x_\sigma, x_{t_2}, x_o)]\!](\Sigma^2, T) \Downarrow (\Sigma^3, T)$, where $(\Sigma^2(x_\sigma), \Sigma^2(x_{t_2}), \sigma') \in T$ and $\Sigma^3 = \Sigma^2 + x_o \to \sigma'$. From the premise, $Q(\sigma') = true$.

Assuming $x_{f_{1'}}$ evaluates to $false$, meaning $isFalse$ should not block, we conclude $(\neg\mathcal{B}[\![b]\!] \wedge P)(\sigma) = true$. According to the premises of the rule, this means that if $(\sigma, \Sigma^2(x_{t_3}), \sigma'') \in T$, then $Q(\sigma'') = true$. We choose the second branch, and evaluate $[\![\text{isFalse}(x_{f_{1'}})]\!](\Sigma^2, T) \Downarrow (\Sigma^2, T)$. The environment doesn't change, since $isFalse$ returns (), meaning no fresh flow variable is added. Now we perform $[\![H(x_\sigma, x_{t_3}, x_o)]\!](\Sigma^2, T) \Downarrow (\Sigma^4, T)$, where $(\Sigma^2(x_\sigma), \Sigma^2(x_{t_3}), \sigma'') \in T$ and $\Sigma^4 = \Sigma^2 + x_o \to \sigma''$. From the premise, $Q(\sigma'') = true$.

Using the concrete interpretation for merging branches, we evaluate the entire branching part:

$$\left[\!\left[\begin{pmatrix} \text{isTrue}(x_{f_{1'}}); H(x_\sigma, x_{t_2}, x_o) \\ \text{isFalse}(x_{f_{1'}}); H(x_\sigma, x_{t_3}, x_o) \end{pmatrix} \right]\!\right] (\Sigma^2, T) \Downarrow (\Sigma^5, T)$$

where $\Sigma^5 = \Sigma^2 + \Sigma^3_{|\{x_o\}}$ or $\Sigma^5 = \Sigma^2 + \Sigma^4_{|\{x_o\}}$. From this, we conclude $\Sigma^5 = \Sigma^2 + x_o \to \sigma'$ or $\Sigma^5 = \Sigma^2 + x_o \to \sigma''$. Lastly, we have the empty skeleton $[\![\,]\!](\Sigma^5, T) \Downarrow \Sigma^5$. Since $Q(\sigma') = true, Q(\sigma'') = true$ and either σ' or σ'' is the output state of the derivation, we can conclude that, given that the premises hold, $\{P\}$ if b then S_1 else S_2 $\{Q\}$ also holds.

While Rule

While rule:

$$\frac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S\ \{P\}}{\{P\}\ \text{while}\ b\ \text{do}\ S\ \{\neg\mathcal{B}[\![b]\!] \wedge P\}}$$

While skeleton:

$$\textsc{While}(while\ x_{t_1}\ x_{t_2}) := \left[\begin{array}{l} H(x_\sigma, x_{t_1}, x_{f_1}); \text{isBool}(x_{f_1})? \triangleright x_{f_{1'}}; \\ \left(\begin{array}{l} \text{isTrue}(x_{f_{1'}}); H(x_\sigma, x_{t_2}, x_{f_2}); H(x_{f_2}, while\ x_{t_1}\ x_{t_2}, x_o) \\ \text{isFalse}(x_{f_{1'}}); \text{id}(x_\sigma)? \triangleright x_o \end{array}\right)_{\{x_o\}} \end{array}\right]$$

For this proof, we will proceed by strong induction on the depth of recursion n, from $\mathcal{H}^n(\emptyset)$. The $isFalse$ branch will serve as the base case, while the $isTrue$ branch will represent the induction step. The induction hypothesis:

$$\forall \sigma, \sigma' : (P(\sigma) \wedge (\sigma, \text{while}\ b\ \text{do}\ S,\ \sigma') \in \mathcal{H}^n(\emptyset)) \implies (\neg\mathcal{B}[\![b]\!](\sigma') \wedge P(\sigma'))$$

We will now start to analyse the skeleton. We take the initial environment: $\Sigma = x_\sigma \to \sigma + x_{t_1} \to b + x_{t_2} \to S$. We assume $P(\sigma) = true$.

First, we have a hook. Assume there exists $T = \mathcal{H}^n(\emptyset)$ for some n such that $[\![H(x_\sigma, x_{t_1}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_{t_1}), v_1) = (\sigma, b, v_1) \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \to v_1$. Next we have $[\![\text{isBool}(x_{f_1})? \triangleright x_{f_{1'}}]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$ with $\Sigma^2 = \Sigma^1 + x_{f_{1'}} \to v_2$, where $[\![\text{isBool}]\!]\Sigma^1(x_{f_1}) \Downarrow v_2$. According to the definition of $isBool$, $v_1 = bool(v_2)$. Using **Lemma 2**, we view $\Sigma^2(x_{f_{1'}})$ as $\mathcal{B}[\![b]\!]$, and conclude that $\mathcal{B}[\![b]\!](\sigma) = v_2$.

We start with the base case, and assume $v_2 = false$. We choose the second branch, and evaluate $[\![\text{isFalse}(x_{f_{1'}})]\!](\Sigma^2, T) \Downarrow (\Sigma^2, T)$. The environment doesn't change, since $isFalse$ returns (), meaning no fresh flow variable is added. Now we perform $[\![\text{id}(x_\sigma)]\!](\Sigma^2, T) \Downarrow (\Sigma^3, T)$, where $\Sigma^3 = \Sigma^2 + x_o \to \sigma'$. Using the definition

for id, we see that $[\![\text{id}]\!]\Sigma^2(x_\sigma) \Downarrow \Sigma^2(x_\sigma)$. But this means that $\Sigma^3(x_o) = \Sigma^2(x_\sigma) = \sigma$, so $\sigma = \sigma'$ and $\Sigma^3 = \Sigma^2 + x_o \to \sigma$. We assumed $\mathcal{B}[\![b]\!](\sigma) = false$ and $P(\sigma) = true$. We concluded that $\sigma = \sigma'$, meaning $P(\sigma') = true$. From this, we can conclude $\{\neg\mathcal{B}[\![b]\!](\sigma') \wedge P(\sigma')\}$.

Now, for the inductive step, we assume $v_2 = \mathcal{B}[\![b]\!](\sigma) = true$, and evaluate the first branch. We take $[\![\text{isTrue}(x_{f_1'})]\!](\Sigma^2, T) \Downarrow (\Sigma^2, T)$. The filter returns (), so the environment doesn't change. Next we have hook $[\![H(x_\sigma, x_{t_2}, x_{f_2})]\!](\Sigma^2, T) \Downarrow (\Sigma^4, T)$, where $(\Sigma^2(x_\sigma), \Sigma^2(x_{t_2}), \sigma'') = (\sigma, S, \sigma'') \in T$ and $\Sigma^4 = \Sigma^2 + x_{f_2} \to \sigma''$. Since $v_2 = B[\![b]\!](\sigma) = true$ and $P(\sigma) = true$, we can use the premise to conclude $P(\sigma'') = true$.

Following is another hook evaluation: $[\![H(x_{f_2}, while\ x_{t_1}\ x_{t_2},\ x_o)]\!](\Sigma^4, T) \Downarrow (\Sigma^5, T)$, where $(\Sigma^4(x_{f_2}), \Sigma^4(while\ x_{t_1}\ x_{t_2}), \sigma^3) = (\sigma'', \text{while } b \text{ do } S, \sigma^3) \in T$ and $\Sigma^5 = \Sigma^4 + x_o \to \sigma^3$. Since we already know $P(\sigma'') = true$, using our induction hypothesis, we can conclude that $P(\sigma^3) = true$ and $\mathcal{B}[\![b]\!](\sigma^3) = false$, or $\{\neg\mathcal{B}[\![b]\!](\sigma^3) \wedge P(\sigma^3)\}$.

Using the concrete interpretation for merging branches, we evaluate the entire branching part:

$$\left[\!\!\left[\begin{pmatrix}\text{isTrue}(x_{f_1'}); H(x_\sigma, x_{t_2}, x_{f_2}); H(x_{f_2}, while\ x_{t_1}\ x_{t_2}, x_o) \\ \text{isFalse}(x_{f_1'}); \text{id}(x_\sigma)? \triangleright x_o\end{pmatrix}\right]\!\!\right](\Sigma^2, T) \Downarrow (\Sigma^6, T)$$

where $\Sigma^6 = \Sigma^2 + \Sigma^3_{|\{x_o\}}$ or $\Sigma^6 = \Sigma^2 + \Sigma^5_{|\{x_o\}}$. From this, we conclude $\Sigma^6 = \Sigma^2 + x_o \to \sigma'$ or $\Sigma^5 = \Sigma^2 + x_o \to \sigma^3$. Lastly, we have the empty skeleton $[\![\,]\!](\Sigma^6, T) \Downarrow \Sigma^6$.

Since $\{\neg\mathcal{B}[\![b]\!](\sigma') \wedge P(\sigma')\}$, $\{\neg\mathcal{B}[\![b]\!](\sigma^3) \wedge P(\sigma^3)\}$ and either σ' or σ^3 is the output state of the derivation, we can conclude that, given that the premise holds, $\{P\}$ while b do S $\{\neg\mathcal{B}[\![b]\!] \wedge P\}$ also holds.

Cons Rule

Consequence rule:

$$\frac{\{P'\}\ S\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

We want to show that if $(\sigma, S, \sigma') \in \mathcal{H}^n(\emptyset)$ and $P(\sigma) = true$, then $Q(\sigma') = true$. We start our proof with the assumption $P(\sigma) = true$, where σ is our initial state. Using the second precondition, we see that $(P(\sigma) = true) \Rightarrow (P'(\sigma) = true)$. This means that we can now apply the first precondition and obtain $Q'(\sigma') = true$. Lastly, we apply the third precondition and conclude $Q(\sigma') = true$, completing our proof.

3.5 Additional Rules

The soundness of all rules and axioms in the language While has been proven. In this section, we go beyong the language While, and extend it with two additional rules by providing their skeletons. We prove their soundness with respect to the Hoare logic rules.

Repeat Rule

Repeat rule:

$$\frac{\{P\}\ S\ \{P\}}{\{P\}\ \text{repeat } S \text{ until } b\ \{\mathcal{B}[\![b]\!] \wedge P\}}$$

Repeat skeleton:

$$\textsc{Repeat}(repeat\ x_{t_1}\ x_{t_2}) := \left[\begin{array}{l} H(x_\sigma, x_{t_1}, x_{f_1}); H(x_{f_1}, x_{t_2}, x_{f_2}); \text{isBool}(x_{f_2})? \triangleright x_{f_{2'}}; \\ \left(\begin{array}{l} \text{isTrue}(x_{f_{2'}});\ \text{id}(x_{f_1})? \triangleright x_o \\ \text{isFalse}(x_{f_{2'}});\ H(x_{f_1},\ repeat\ x_{t_1}\ x_{t_2},\ x_o) \end{array}\right)_{\{x_o\}} \end{array}\right]$$

Similarly to the While rule, the Repeat rule represents a loop construct. Here statement S is executed once, and then expression b is evaluated. If the evaluation returns $false$, the entire process is repeated, while if b evaluates to $true$, the execution ends. We will use again strong induction on the depth of recursion n, in $\mathcal{H}^n(\emptyset)$. The $isTrue$ branch will serve as the base case, while the $isFalse$ branch will represent the induction step. The induction hypothesis:

$$\forall \sigma, \sigma' : (P(\sigma) \wedge (\sigma, \text{repeat } S \text{ until } b,\ \sigma') \in \mathcal{H}^n(\emptyset)) \implies (\mathcal{B}[\![b]\!](\sigma') \wedge P(\sigma'))$$

We will now start to analyse the skeleton. We take the initial environment: $\Sigma = x_\sigma \rightarrow \sigma + x_{t_1} \rightarrow S + x_{t_2} \rightarrow b$. We assume $P(\sigma) = true$. First, we have a hook.

Assume there exists $T = \mathcal{H}^n(\emptyset)$ for some n such that $[\![H(x_\sigma, x_{t_1}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$, where $(\Sigma(x_\sigma), \Sigma(x_{t_1}), \sigma') = (\sigma, S, \sigma') \in T$ and $\Sigma^1 = \Sigma + x_{f_1} \to \sigma'$. Since $P(\sigma) = true$, by using the premise we conclude $P(\sigma') = true$.

For the next hook, we have $[\![H(x_{f_1}, x_{t_2}, x_{f_2})]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$, where $(\Sigma(x_{f_1}), \Sigma(x_{t_2}), v_1) = (\sigma', b, v_1) \in T$ and $\Sigma^2 = \Sigma + x_{f_2} \to v_1$. Next we have $[\![\text{isBool}(x_{f_2})?\triangleright x_{f_{2'}}]\!](\Sigma^2, T) \Downarrow (\Sigma^3, T)$ with $\Sigma^3 = \Sigma^2 + x_{f_{2'}} \to v_2$, where $[\![\text{isBool}]\!]\Sigma^2(x_{f_2}) \Downarrow v_2$. According to the definition of $isBool$, $v_1 = bool(v_2)$. Using **Lemma 2**, we view $\Sigma^3(x_{f_{2'}})$ as $\mathcal{B}[\![b]\!]$, and conclude that $\mathcal{B}[\![b]\!](\sigma') = v_2$.

We start with the base case, and assume $v_2 = true$. We choose the first branch, and evaluate $[\![\text{isTrue}(x_{f_{2'}})]\!](\Sigma^3, T) \Downarrow (\Sigma^3, T)$. The environment doesn't change, since $isFalse$ returns (), meaning no fresh flow variable is added. Now we perform $[\![\text{id}(x_{f_1})]\!](\Sigma^3, T) \Downarrow (\Sigma^4, T)$, where $\Sigma^4 = \Sigma^3 + x_o \to \sigma''$.

Using the definition for id, we see that $[\![\text{id}]\!]\Sigma^3(x_{f_1}) \Downarrow \Sigma^3(x_{f_1})$. But this means that $\Sigma^4(x_o) = \Sigma^3(x_{f_1}) = \sigma'$, so $\sigma' = \sigma''$ and $\Sigma^4 = \Sigma^3 + x_o \to \sigma'$. We assumed $\mathcal{B}[\![b]\!](\sigma') = true$ and $P(\sigma') = true$. We concluded that $\sigma' = \sigma''$, meaning $P(\sigma'') = true$. From this, we can conclude $\{\mathcal{B}[\![b]\!](\sigma'') \wedge P(\sigma'')\}$.

Now, for the inductive step, we assume $v_2 = \mathcal{B}[\![b]\!](\sigma') = false$, and evaluate the second branch. We take $[\![\text{isFalse}(x_{f_{2'}})]\!](\Sigma^3, T) \Downarrow (\Sigma^3, T)$. The filter returns (), so the environment doesn't change. Following is a hook evaluation: $[\![H(x_{f_2}, repeat\ x_{t_1}\ x_{t_2},\ x_o)]\!](\Sigma^3, T) \Downarrow (\Sigma^5, T)$, where $(\Sigma^3(x_{f_1}), \Sigma^3(repeat\ x_{t_1}\ x_{t_2}), \sigma^3) = (\sigma', \text{repeat } S \text{ until } b, \sigma^3) \in T$ and $\Sigma^5 = \Sigma^3 + x_o \to \sigma^3$. Since we already know $P(\sigma') = true$, using our induction hypothesis, we can conclude that $P(\sigma^3) = true$ and $\mathcal{B}[\![b]\!](\sigma^3) = true$, or $\{\mathcal{B}[\![b]\!](\sigma^3) \wedge P(\sigma^3)\}$.

Using the concrete interpretation for merging branches, we evaluate the entire branching part:

$$\left[\!\!\left[\begin{pmatrix} \text{isTrue}(x_{f_{2'}});\ \text{id}(x_{f_1})?\triangleright x_o \\ \text{isFalse}(x_{f_{2'}});\ H(x_{f_1},\ repeat\ x_{t_1}\ x_{t_2},\ x_o) \end{pmatrix} \right]\!\!\right](\Sigma^3, T) \Downarrow (\Sigma^6, T)$$

where $\Sigma^6 = \Sigma^3 + \Sigma^4_{|\{x_o\}}$ or $\Sigma^6 = \Sigma^3 + \Sigma^5_{|\{x_o\}}$. From this, we conclude $\Sigma^6 = \Sigma^3 + x_o \to \sigma''$ or $\Sigma^5 = \Sigma^2 + x_o \to \sigma^3$. Lastly, we have the empty skeleton $[\![\,]\!](\Sigma^6, T) \Downarrow \Sigma^6$.

Since $\{\mathcal{B}[\![b]\!](\sigma'') \wedge P(\sigma'')\}$, $\{\mathcal{B}[\![b]\!](\sigma^3) \wedge P(\sigma^3)\}$ and either σ'' or σ^3 is the output state of the derivation, we can conclude that, given that the premise holds, $\{P\}$ repeat S until b $\{\mathcal{B}[\![b]\!] \wedge P\}$ also holds.

For Rule

We want to construct a proof for the for-rule. In other words, we want to repeat the execution of a statement S for all possible values of an iterator x, in an interval $[a_1, a_2)$. We define the rule for values up to and not including the upper value. So in: for $x := a_1$ to a_2 do S, the values x will take are $[a_1, a_1 + 1, ..., a_2 - 1]$. We also assume x doesn't occur in S. In this case it is easy to see that the for-rule is equivalent to the statement: $x := a_1$; while $x \neq a_2$ do $(S;\ x := x + 1)$. Next we have the following derivation tree:

$$\dfrac{\{R[x \to \mathcal{E}[\![a_1]\!]]\}\ x := a\ \{R\} \qquad \dfrac{\dfrac{\{R \wedge \mathcal{B}[\![x \neq a_2]\!]\}\ S\ \{R[x \to \mathcal{E}[\![x+1]\!]]\} \qquad \{R[x \to \mathcal{E}[\![x+1]\!]]\}\ x := x+1\ \{R\}}{\{R \wedge \mathcal{B}[\![x \neq a_2]\!]\}\ S;\ x := x+1\ \{R\}}}{\{R\}\ \text{while}\ (x \neq a_2)\ \text{do}\ (S; x := x+1)\ \{R \wedge \mathcal{B}[\![x = a_2]\!]\}}}{\{R[x \to \mathcal{E}[\![a_1]\!]]\}\ x := a_1;\ \text{while}\ (x \neq a_2)\ \text{do}\ (S;\ x := x+1)\ \{R \wedge \mathcal{B}[\![x = a_2]\!]\}}$$

From the tree, we see that the only derivation step which is not using a rule, and therefore has to be an assumption is $\{R \wedge \mathcal{B}[\![x \neq a_2]\!]\}\ S\ \{R[x \to \mathcal{E}[\![x+1]\!]]\}$. To be consistent with notation, we write $\neg\mathcal{B}[\![x = a_2]\!]$ instead of $\mathcal{B}[\![x \neq a_2]\!]$. From this, we can derive the for-rule:

$$\dfrac{\{R \wedge \neg\mathcal{B}[\![x = a_2]\!]\}\ S\ \{R[x \to \mathcal{E}[\![x+1]\!]]\}}{\{R[x \to \mathcal{E}[\![a_1]\!]]\}\ \text{for}\ x := a_1\ \text{to}\ a_2\ \text{do}\ S\ \{R \wedge \mathcal{B}[\![x = a_2]\!]\}}$$

We introduce the skeleton for the For rule:

$$\textsc{For}(for\ x_{t_1}\ x_{t_2}\ x_{t_3}\ x_{t_4}) := \left[\begin{array}{l} H(x_\sigma, x_{t_2}, x_{f_1}); \text{isInt}(x_{f_1})? \triangleright x_{f_{1'}}; \text{write}(x_{t_1}, x_\sigma, x_{f_1})? \triangleright x_{f_2}; \\ H(x_{f_2}, x_{t_3}, x_{f_3}); \text{isInt}(x_{f_3})? \triangleright x_{f_{3'}}; \text{eq}(x_{f_{1'}}, x_{f_{3'}})? \triangleright x_{f_4}; \\ \left(\begin{array}{l} \text{isTrue}(x_{f_4});\ \text{id}(x_{f_2})? \triangleright x_o \\ \text{isFalse}(x_{f_4});\ H(x_{f_2}, x_{t_4}, x_{f_5}); \\ \quad \text{add}(x_{f_{1'}}, 1)? \triangleright x_{f_6};\ \text{intVal}(x_{f_6})? \triangleright x_{f_7}; \\ \quad\quad H(x_{f_5},\ for\ x_{t_1}\ x_{f_7}\ x_{t_3}\ x_{t_4},\ x_o) \end{array}\right)_{\{x_o\}} \end{array}\right]$$

We proceed with a proof by strong induction on the depth of recursion n. The isTrue branch will correspond to the base case, while the isFalse branch will represent the inductive step. The induction hypothesis is:

$$\forall \sigma, \sigma' : (R(\sigma[x \to \mathcal{E}[\![a_1]\!](\sigma)]) \wedge (\sigma, \text{for } x := a_1 \text{ to } a_2 \text{ do } S,\ \sigma') \in \mathcal{H}^n(\emptyset)) \implies (\mathcal{B}[\![x = a_2]\!](\sigma') \wedge R(\sigma'))$$

We start evaluating the skeleton in state σ, assuming $R(\sigma[x \to \mathcal{E}[\![a_1]\!](\sigma)]) = true$. The initial environment is $\Sigma = x_\sigma \to \sigma + x_{t_1} \to x + x_{t_2} \to a_1 + x_{t_3} \to a_2 + x_{t_4} \to S$. We start with the hook $[\![H(x_\sigma, x_{t_2}, x_{f_1})]\!](\Sigma, T) \Downarrow (\Sigma^1, T)$. We assume $T = \mathcal{H}^n(\emptyset)$ exists for some n such that $(\Sigma(x_\sigma), \Sigma(x_{t_2}), v_1) = (\sigma, a_1, v_1) \in T$. Here $\Sigma^1 = \Sigma + x_{f_1} \to v_1$. As a_1 is an expression, v_1 will be a value.

We continue with filter $[\![\text{isInt}(x_{f_1})?\triangleright x_{f_{1'}}]\!](\Sigma^1, T) \Downarrow (\Sigma^2, T)$. We have $[\![\text{isInt}]\!]\Sigma^1(x_{f_1}) \Downarrow v_2$, or $[\![\text{isInt}]\!]v_1 \Downarrow v_2$. Here $\Sigma^2 = \Sigma^1 + x_{f_{1'}} \to v_2$. By the definition of isInt, we know $v_1 = int(v_2)$. We can thus apply **Lemma 1** and deduce that $v_2 = \mathcal{A}[\![a_1]\!](\sigma)$. From this, we also know $v_1 = \mathcal{E}[\![a_1]\!](\sigma)$.

Moving on to the next filter, $[\![\text{write}(x_{t_1}, x_\sigma, x_{f_1})? \triangleright x_{f_2}]\!](\Sigma^2, T) \Downarrow (\Sigma^3, T)$. We evaluate $[\![\text{write}]\!](\Sigma^2(x_{t_1}), \Sigma^2(x_\sigma), \Sigma^2(x_{f_1})) \Downarrow \sigma'$. In other words, $[\![\text{write}]\!](x, \sigma, v_1) \Downarrow \sigma'$. The new environment is $\Sigma^3 = \Sigma^2 + x_{f_2} \to \sigma'$. Following the definition of write, we know $\sigma' = \sigma[x \to v_1] = \sigma[x \to \mathcal{E}[\![a_1]\!](\sigma)]$. From our initial assumption, we can deduce $R(\sigma') = true$. Now since the value of x is v_1, we can also conclude $\mathcal{A}[\![x]\!](\sigma') = v_2$.

We arrive at another hook: $[\![H(x_{f_2}, x_{t_3}, x_{f_3})]\!](\Sigma^3, T) \Downarrow (\Sigma^4, T)$. Assuming $(\Sigma^3(x_{f_2}), \Sigma^3(x_{t_3}), v_3) = (\sigma', a_2, v_3) \in T$, we have $\Sigma^4 = \Sigma^3 + x_{f_3} \to v_3$. Next we have $[\![\text{isInt}(x_{f_3})? \triangleright x_{f_{3'}}]\!](\Sigma^4, T) \Downarrow (\Sigma^5, T)$. Let $[\![\text{isInt}]\!]\Sigma^4(x_{f_3}) \Downarrow v_4$, or $[\![\text{isInt}]\!]v_3 \Downarrow v_4$, $\Sigma^5 = \Sigma^4 + x_{f_{3'}} \to v_4$. By the definition of isInt, we know $v_3 = int(v_4)$. Applying **Lemma 1**, we conclude $\mathcal{A}[\![a_2]\!](\sigma') = v_4$.

Now we take filter $[\![\text{eq}(x_{f_{1'}}, x_{f_{3'}})? \triangleright x_{f_4}]\!](\Sigma^5, T) \Downarrow (\Sigma^6, T)$. We get $[\![\text{eq}]\!](\Sigma^5(x_{f_{1'}}), \Sigma^5(x_{f_{3'}})) \Downarrow v_5$, or $[\![\text{eq}]\!](v_2, v_4) \Downarrow v_5$, where $\Sigma^6 = \Sigma^5 + x_{f_4} \to v_5$. Since $\mathcal{A}[\![x]\!](\sigma) = \mathcal{A}[\![x]\!](\sigma') = v_2$ and $\mathcal{A}[\![a_2]\!](\sigma') = v_4$, using the definition for $\mathcal{B}$, $v_5 = \mathcal{B}[\![x = a_2]\!](\sigma')$.

We start evaluating the branches. We first choose the $isTrue$ branch, corresponding to the base case for our proof. We assume $v_5 = true$, so $\mathcal{B}[\![x = a_2]\!](\sigma') = true$. We evaluate the filter: $[\![\text{isTrue}(x_{f_4})]\!](\Sigma^6, T) \Downarrow (\Sigma^6, T)$. The filter doesn't introduce new flow varibles, so the environment stays the same. Moving on, we have the filter $[\![\text{id}(x_{f_2})? \triangleright x_o]\!](\Sigma^6, T) \Downarrow (\Sigma^7, T)$. Since $\Sigma^6(x_{f_2}) = \sigma'$, using the definition of id, we get $[\![\text{id}]\!]\sigma' \Downarrow \sigma'$, thus $\Sigma^7 = \Sigma^6 + x_o \rightarrow \sigma'$. Since we know $R(\sigma') = true$ and $\mathcal{B}[\![x = a_2]\!](\sigma') = true$, the base case holds.

The second branch represents our inductive step. We assume $v_5 = false$. This means $\mathcal{B}[\![x = a_2]\!](\sigma') = false$, which is the same as $\neg\mathcal{B}[\![x = a_2]\!](\sigma') = true$. We evaluate the filter $[\![\text{isFalse}(x_{f_4})]\!](\Sigma^6, T) \Downarrow (\Sigma^6, T)$. Again, the environment doesn't change.

We continue with hook $[\![H(x_{f_2}, x_{t_4}, x_{f_5})]\!](\Sigma^6, T) \Downarrow (\Sigma^8, T)$. Assuming $(\Sigma^6(x_{f_2}), \Sigma^6(x_{t_4}), \sigma'') = (\sigma', S, \sigma'') \in T$, we have $\Sigma^8 = \Sigma^6 + x_{f_5} \rightarrow \sigma''$. Since $R(\sigma') = true$ and $\neg\mathcal{B}[\![x = a_2]\!](\sigma') = true$, we use the premise to conclude $R(\sigma''[x \rightarrow \mathcal{E}[\![x + 1]\!](\sigma'')]) = true$.

We now analyse filter $[\![\text{add}(x_{f_{1'}}, 1)? \triangleright x_{f_6}]\!](\Sigma^8, T) \Downarrow (\Sigma^9, T)$. We have $[\![\text{add}]\!](\Sigma^9(x_{f_{1'}}), \Sigma^9(1)) \Downarrow v_6$, or $[\![\text{add}]\!](v_2, 1) \Downarrow v_6$, where $\Sigma^9 = \Sigma^8 + x_{f_6} \rightarrow v_6$. We know $v_2 = \mathcal{A}[\![x]\!](\sigma') = \mathcal{A}[\![x]\!](\sigma'')$. From the definitions of function $\mathcal{A}$ and filter add, we have $v_6 = v_2 + 1 = \mathcal{A}[\![x]\!](\sigma'') + \mathcal{A}[\![1]\!](\sigma'') = \mathcal{A}[\![x + 1]\!](\sigma'')$.

The next filter is $[\![\text{intVal}(x_{f_6})? \triangleright x_{f_7}]\!](\Sigma^9, T) \Downarrow (\Sigma^{10}, T)$. We have $[\![\text{intVal}]\!]\Sigma^9(x_{f_6}) \Downarrow v_7$, or $[\![\text{intVal}]\!]v_6 \Downarrow v_7$, and $\Sigma^{10} = \Sigma^9 + x_{f_7} \rightarrow v_7$. Since $v_6 = \mathcal{A}[\![x + 1]\!](\sigma'')$ and $v_7 = int(v_6)$, using the definition of $\mathcal{E}$, we find $v_7 = \mathcal{E}[\![x + 1]\!](\sigma'')$. We know $R(\sigma''[x \rightarrow \mathcal{E}[\![x + 1]\!](\sigma'')]) = true$, so $R(\sigma''[x \rightarrow v_7]) = true$.

Now we have hook $[\![H(x_{f_5},\ for\ x_{t_1}\ x_{f_7}\ x_{t_3}\ x_{t_4},\ x_o)]\!](\Sigma^{10}, T) \Downarrow (\Sigma^{11}, T)$. Here $\Sigma^{11} = \Sigma^{10} + x_o \rightarrow \sigma^3$. By the induction hypothesis, since $R(\sigma''[x \rightarrow v_7]) = true$, and $\Sigma^{10}(x_{f_7}) = v_7$, and $(\sigma'',\ \text{for}\ x := v_7\ \text{to}\ a_2\ \text{do}\ S, \sigma^3) \in T$, we can conclude $\{R(\sigma^3) \wedge \mathcal{B}[\![x = a_2]\!](\sigma^3)\}$.

By the concrete interpretation for merging branches, we evaluate the branching:

$$\left[\!\!\left[\begin{pmatrix} \text{isTrue}(x_{f_4});\ \text{id}(x_{f_2})? \triangleright x_o \\ \text{isFalse}(x_{f_4});\ H(x_{f_2}, x_{t_4}, x_{f_5}); \\ \quad \text{add}(x_{f_1},, 1)? \triangleright x_{f_6};\ \text{intVal}(x_{f_6})? \triangleright x_{f_7}; \\ \quad\quad H(x_{f_5},\ for\ x_{t_1}\ x_{f_7}\ x_{t_3}\ x_{t_4},\ x_o) \end{pmatrix}\right]\!\!\right] (\Sigma^6, T) \Downarrow (\Sigma^{12}, T)$$

where $\Sigma^{12} = \Sigma^6 + \Sigma^7_{|\{x_o\}} = \Sigma^6 + x_o \to \sigma'$ or $\Sigma^{12} = \Sigma^6 + \Sigma^{11}_{|\{x_o\}} = \Sigma^6 + x_o \to \sigma^3$. Lastly, we have the empty skeleton $[\![\,]\!](\Sigma^{12}, T) \Downarrow \Sigma^{12}$.

Since $\{\mathcal{B}[\![x = a_2]\!](\sigma') \wedge R(\sigma')\}$, $\{\mathcal{B}[\![x = a_2]\!](\sigma^3) \wedge R(\sigma^3)\}$ and either σ' or σ^3 is the output state of the derivation, we can conclude that, given that the premise holds, $\{R[x \to \mathcal{E}[\![a_1]\!]]\}$ for $x := a_1$ to a_2 do S $\{R \wedge \mathcal{B}[\![x = a_2]\!]\}$ also holds.

3.6 Suggestions for Interpreting Skeletons

In the process of proving the soundness property for the language While, we developed an intuition about the mapping of skeletons to Hoare logic rules. These suggestions are intended to act as a heuristic, and do not guarantee that strictly following them will produce a correct mapping. The observations are:

1. A hook on a term x_t representing an expression: $H[\![(\sigma, x_t, x_f)]\!]$, followed by an application of the $isBool$ filter on x_f and a $isTrue/isFalse$ branching on the $bool$ result suggests the presence of a predicate of the form $\mathcal{B}[\![b]\!]$ or $\neg\mathcal{B}[\![b]\!]$, respectively.
2. The presence of a branching on the truth value of a (flow) variable suggests the presence of an if-statement, or a statement that can be rewritten as an if-statement, such as $while$, $repeat$, for.
3. The presence of the bone in its own skeleton means the construct is recursive. This happens in a branching, one of the other branches representing the base case of the recursion. As an example, in the skeleton for the $while$ rule, we have hook $H(x_{f_2}, while\ x_{t_1}\ x_{t_2}, x_o)$.
4. The id filter is usually translated to a $skip$ axiom, or it is not translated and represents an unwritten branch, as we have in the $while$ rule.
5. A branching with more than two branches would suggest the use of a $switch/case$ statement, in languages that contain one, or nested if-statements otherwise.
6. Two consecutive hooks suggest the use of statement composition, represented by the seq rule.

4

Conclusions and Future Works

The goal of this project was to examine the relation between Hoare logic and skeletal semantics. We wanted to explore the possibility of building correct-by-construction Hoare logics for languages formalised using skeletal semantics.

In the paper, we described the relation between Hoare logic and the skeletal semantics of a simple imperative language called WHILE. We formally introduced the syntax and semantics of the language and proved the soundness property with respect to Hoare logic. We did this by proving the partial correctness for all arithmetic and boolean expressions, and also for all statements in the language - part which turned out to be the most difficult. We added two additional rules to the language, deriving their skeletons, and proving their partial correctness.

Lastly, we introduced a short heuristic for mapping skeletons to Hoare logic rules. The heuristic is incomplete and does not guarantee a correct mapping every time, but it can be used as an intuition for making the task more straightforward.

Although studying the relation between Hoare logic and a specific language described using skeletal semantics is the first step, our goal of finding a general approach still requires more research. Suggested next steps would be enhancing the language with new, commonly used constructs and proving their soundness. Examples would be supporting arrays, lists, input/output, selection statements with more than two branches, exceptions and control transfer statements, such as try-catch, break, return, etc. Another option would be proving soundness for more languages, imperative and not only, with the prospect of observing

more similarities and ultimately formalising a language-independent theorem for deriving Hoare logic from skeletal semantics.

References

[1] G. Ambal, S. Lenglet, and A. Schmitt. "Certified Abstract Machines for Skeletal Semantics". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2022. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 55–67. ISBN: 9781450391825. DOI: 10.1145/3497775.3503676. URL: https://doi-org.proxy-ub.rug.nl/10.1145/3497775.3503676.

[2] T. F. Bissyandé et al. "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects". In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. 2013, pp. 303–312. DOI: 10.1109/COMPSAC.2013.55.

[3] M. Bodin et al. "Skeletal Semantics and Their Interpretations". In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290357. URL: https://doi-org.proxy-ub.rug.nl/10.1145/3290357.

[4] S. L. Hantler and J. C. King. "An Introduction to Proving the Correctness of Programs". In: *ACM Comput. Surv.* 8.3 (1976), pp. 331–353. ISSN: 0360-0300. DOI: 10.1145/356674.356677. URL: https://doi-org.proxy-ub.rug.nl/10.1145/356674.356677.

[5] P. He et al. "Hoare logic-based genetic programming". In: *Science China Information Sciences* 54.3 (2011). 623, pp. 623–637. ISSN: 1674-733X. DOI: 10.1007/s11432-011-4200-4. URL: https://doi.org/10.1007/s11432-011-4200-4.

[6] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi-org.proxy-ub.rug.nl/10.1145/363235.363259.

[7] T. Hoare and S. van Staden. "In Praise of Algebra". In: *Form. Asp. Comput.* 24.46 (2012), pp. 423–431. ISSN: 0934-5043. DOI: 10.1007/s00165-012-0249-0. URL: https://doi-org.proxy-ub.rug.nl/10.1007/s00165-012-0249-0.

[8] B. Jacobs. "Weakest pre-condition reasoning for Java programs with JML annotations". In: *The Journal of Logic and Algebraic Programming* 58.1 (2004). Formal Methods for Smart Cards, pp. 61–88. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2003.07.005. URL: https://www.sciencedirect.com/science/article/pii/S1567832603000766.

[9] T. A. Linden. "A Summary of Progress toward Proving Program Correctness". In: *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*. AFIPS '72 (Fall, part I). Anaheim, California: Association for Computing Machinery, 1972, pp. 201–211. ISBN: 9781450379120. DOI: 10.1145/1479992.1480019. URL: https://doi-org.proxy-ub.rug.nl/10.1145/1479992.1480019.

[10] H. R. Nielson and F. Nielson. "Content dependent information flow control". In: *Journal of Logical and Algebraic Methods in Programming* 87 (2017), pp. 6–32. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2016.09.005. URL: https://www.sciencedirect.com/science/article/pii/S2352220816301134.

[11] H. R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 1846286913.

[12] T. Nipkow. "Winskel is (Almost) Right: Towards a Mechanized Semantics Textbook". In: *Form. Asp. Comput.* 10.2 (1998), pp. 171–186. ISSN: 0934-5043. DOI: 10.1007/s001650050009. URL: https://doi-org.proxy-ub.rug.nl/10.1007/s001650050009.

[13] I. Sergey et al. "Hoare-Style Specifications as Correctness Conditions for Non-Linearizable Concurrent Objects". In: *SIGPLAN Not.* 51.10 (2016), pp. 92–110. ISSN: 0362-1340. DOI: 10.1145/3022671.2983999. URL: https://doi-org.proxy-ub.rug.nl/10.1145/3022671.2983999.

[14] A. Stefnescu et al. "Semantics-Based Program Verifiers for All Languages". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016.

Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 74–91. ISBN: 9781450344449. DOI: 10.1145/2983990.2984027. URL: https://doi-org.proxy-ub.rug.nl/10.1145/2983990.2984027.

Acknowledgments

I would like to thank the incredibly skilled and competent Dr. Dan Frumin, without whom the completion of this project would not have been possible. Thank you for the constant guidance and support. Special thanks to my best friend Evghenii - thank you for reviewing my drafts, for your patience and kindness, for inspiring me and making this process significantly more fun and productive.