Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

March 1, 2016

DRAFT

Springer

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing[1]

DRAFT

[1] The purpose of ordinal logics (from Systems of Logic Based on Ordinals), Proceedings of the London Mathematical Society, series 2, vol. 45, 1939.

Contents

Part I Preliminaries

1 Introduction 3
- 1.1 Structure and Meaning 3
  - 1.1.1 Syntax and Types 4
  - 1.1.2 Semantics 4
  - 1.1.3 Mathematical Models of Computation 6
  - 1.1.4 Operational Semantics 7
  - 1.1.5 Denotational Semantics 8
  - 1.1.6 Axiomatic Semantics 8
- 1.2 A Taste of Semantics Methods: Numerical Expressions 9
  - 1.2.1 An Informal Semantics 10
  - 1.2.2 A Small-Step Operational Semantics 11
  - 1.2.3 A Big-Step Operational Semantics (or Natural Semantics) . . 13
  - 1.2.4 A Denotational Semantics 15
  - 1.2.5 Semantic Equivalence 16
  - 1.2.6 Expressions with Variables 17
- 1.3 Applications of Semantics 18
  - 1.3.1 Language Design 18
  - 1.3.2 Implementation 18
  - 1.3.3 Analysis and Verification 19
  - 1.3.4 Synergy Between Different Semantics Approaches 19
- 1.4 Content Overview 20
  - 1.4.1 Induction and Recursion 22
  - 1.4.2 Semantic Domains 24
  - 1.4.3 Bisimulation 26
  - 1.4.4 Temporal and Modal Logics 26
  - 1.4.5 Probabilistic Systems 27
- 1.5 Chapters Contents and Reading Guide 27

DRAFT

2 Preliminaries . . . 31
2.1 Notation . . . 31
2.1.1 Basic Notation . . . 31
2.1.2 Signatures and Terms . . . 32
2.1.3 Substitutions . . . 33
2.1.4 Unification Problem . . . 33
2.2 Inference Rules and Logical Systems . . . 35
2.3 Logic Programming . . . 43
Problems . . . 45

Part II IMP: a simple imperative language

3 Operational Semantics of IMP . . . 51
3.1 Syntax of IMP . . . 51
3.1.1 Arithmetic Expressions . . . 52
3.1.2 Boolean Expressions . . . 52
3.1.3 Commands . . . 52
3.1.4 Abstract Syntax . . . 53
3.2 Operational Semantics of IMP . . . 54
3.2.1 Memory State . . . 54
3.2.2 Inference Rules . . . 55
3.2.3 Examples . . . 59
3.3 Abstract Semantics: Equivalence of Expressions and Commands . . . 64
3.3.1 Examples: Simple Equivalence Proofs . . . 65
3.3.2 Examples: Parametric Equivalence Proofs . . . 66
3.3.3 Examples: Inequality Proofs . . . 68
3.3.4 Examples: Diverging Computations . . . 70
Problems . . . 73

4 Induction and Recursion . . . 75
4.1 Noether Principle of Well-founded Induction . . . 75
4.1.1 Well-founded Relations . . . 75
4.1.2 Noether Induction . . . 81
4.1.3 Weak Mathematical Induction . . . 82
4.1.4 Strong Mathematical Induction . . . 83
4.1.5 Structural Induction . . . 83
4.1.6 Induction on Derivations . . . 86
4.1.7 Rule Induction . . . 87
4.2 Well-founded Recursion . . . 90
Problems . . . 95

5 Partial Orders and Fixpoints . . . 99
5.1 Orders and Continuous Functions . . . 99
5.1.1 Orders . . . 100
5.1.2 Hasse Diagrams . . . 101
5.1.3 Chains . . . 105

DRAFT

5.1.4 Complete Partial Orders 105
5.2 Continuity and Fixpoints 108
5.2.1 Monotone and Continuous Functions 108
5.2.2 Fixpoints 110
5.3 Immediate Consequence Operator 113
5.3.1 The Operator $\widehat{R}$ 113
5.3.2 Fixpoint of $\widehat{R}$ 114
Problems 117

6 Denotational Semantics of IMP 121
6.1 λ-Notation 121
6.1.1 λ-Notation: Main Ideas 122
6.1.2 Alpha-Conversion, Beta-Rule and Capture-Avoiding Substitution 125
6.2 Denotational Semantics of IMP 127
6.2.1 Denotational Semantics of Arithmetic Expressions: The Function $\mathscr{A}$ 128
6.2.2 Denotational Semantics of Boolean Expressions: The Function $\mathscr{B}$ 129
6.2.3 Denotational Semantics of Commands: The Function $\mathscr{C}$ 130
6.3 Equivalence Between Operational and Denotational Semantics 135
6.3.1 Equivalence Proofs For Expressions 135
6.3.2 Equivalence Proof for Commands 136
6.4 Computational Induction 143
Problems 145

Part III HOFL: a higher-order functional language

7 Operational Semantics of HOFL 151
7.1 Syntax of HOFL 151
7.1.1 Typed Terms 152
7.1.2 Typability and Typechecking 156
7.2 Operational Semantics of HOFL 159
Problems 164

8 Domain Theory 167
8.1 The Flat Domain of Integer Numbers $\mathbb{Z}_\perp$ 167
8.2 Cartesian Product of Two Domains 167
8.3 Functional Domains 169
8.4 Lifting 172
8.5 Function's Continuity Theorems 174
8.6 Useful Functions 177
Problems 181

DRAFT

9 HOFL Denotational Semantics 183
9.1 HOFL Semantic Domains 183
9.2 HOFL Evaluation Function 184
9.2.1 Constants 184
9.2.2 Variables 184
9.2.3 Binary Operators 185
9.2.4 Conditional 185
9.2.5 Pairing 186
9.2.6 Projections 186
9.2.7 Lambda Abstraction 187
9.2.8 Function Application 187
9.2.9 Recursion 187
9.3 Continuity of Meta-language's Functions 189
9.4 Substitution Lemma 191
Problems 192

10 Equivalence between HOFL denotational and operational semantics . 195
10.1 Completeness 196
10.2 Equivalence (on Convergence) 199
10.3 Operational and Denotational Equivalences of Terms 201
10.4 A Simpler Denotational Semantics 202
Problems 203

Part IV Concurrent Systems

11 CCS, the Calculus for Communicating Systems 209
11.1 Syntax of CCS 214
11.2 Operational Semantics of CCS 215
11.2.1 Action Prefix 216
11.2.2 Restriction 216
11.2.3 Relabelling 216
11.2.4 Choice 217
11.2.5 Parallel Composition 217
11.2.6 Recursion 218
11.2.7 CCS with Value Passing 221
11.2.8 Recursive Declarations and the Recursion Operator 222
11.3 Abstract Semantics of CCS 224
11.3.1 Graph Isomorphism 224
11.3.2 Trace Equivalence 226
11.3.3 Bisimilarity 227
11.4 Compositionality 233
11.4.1 Bisimilarity is Preserved by Choice 234
11.5 A Logical View to Bisimilarity: Hennessy-Milner Logic 235
11.6 Axioms for Strong Bisimilarity 238
11.7 Weak Semantics of CCS 240

DRAFT

11.7.1 Weak Bisimilarity . . . 240
11.7.2 Weak Observational Congruence . . . 242
11.7.3 Dynamic Bisimilarity . . . 243
Problems . . . 244

12 Temporal Logic and μ-Calculus . . . 249
12.1 Temporal Logic . . . 249
12.1.1 Linear Temporal Logic . . . 250
12.1.2 Computation Tree Logic . . . 252
12.2 μ-Calculus . . . 254
12.3 Model Checking . . . 257
Problems . . . 258

13 π-Calculus . . . 261
13.1 Name Mobility . . . 261
13.2 Syntax of the π-calculus . . . 264
13.3 Operational Semantics of the π-calculus . . . 266
13.3.1 Action Prefix . . . 267
13.3.2 Choice . . . 268
13.3.3 Name Matching . . . 268
13.3.4 Parallel Composition . . . 268
13.3.5 Restriction . . . 269
13.3.6 Scope Extrusion . . . 269
13.3.7 Replication . . . 269
13.3.8 A Sample Derivation . . . 270
13.4 Structural Equivalence of π-calculus . . . 271
13.4.1 Reduction semantics . . . 271
13.5 Abstract Semantics of the π-calculus . . . 272
13.5.1 Strong Early Ground Bisimulations . . . 273
13.5.2 Strong Late Ground Bisimulations . . . 274
13.5.3 Strong Full Bisimilarities . . . 275
13.5.4 Weak Early and Late Ground Bisimulations . . . 276
Problems . . . 277

Part V Probabilistic Systems

14 Measure Theory and Markov Chains . . . 281
14.1 Probabilistic and Stochastic Systems . . . 281
14.2 Measure Theory . . . 282
14.2.1 σ-field . . . 282
14.2.2 Constructing a σ-field . . . 283
14.2.3 Continuous Random Variables . . . 285
14.2.4 Stochastic Processes . . . 289
14.3 Markov Chains . . . 289
14.3.1 Discrete and Continuous Time Markov Chain . . . 290
14.3.2 DTMC as LTS . . . 291

DRAFT

14.3.3 DTMC Steady State Distribution 293
14.3.4 CTMC as LTS 295
14.3.5 Embedded DTMC of a CTMC 296
14.3.6 CTMC Bisimilarity 296
14.3.7 DTMC Bisimilarity 298
Problems 299

15 Markov Chains with Actions and Non-determinism 303
15.1 Discrete Markov Chains With Actions 303
15.1.1 Reactive DTMC 304
15.1.2 DTMC With Non-determinism 306
Problems 309

16 PEPA - Performance Evaluation Process Algebra 311
16.1 From Qualitative to Quantitative Analysis 311
16.2 CSP 312
16.2.1 Syntax of CSP 312
16.2.2 Operational Semantics of CSP 313
16.3 PEPA 314
16.3.1 Syntax of PEPA 314
16.3.2 Operational Semantics of PEPA 316
Problems 321

Glossary 325

Solutions 327

DRAFT

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\cong$	CCS weak observational congruence (see Section 11.7.2)
$\approx_d$	CCS dynamic bisimilarity (see Definition 11.17)
$\overset{\circ}{\sim}_E$	π-calculus early bisimilarity (see Definition 13.3)
$\overset{\circ}{\sim}_L$	π-calculus late bisimilarity (see Definition 13.4)
$\sim_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\sim_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\overset{\bullet}{\approx}_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\overset{\bullet}{\approx}_L$	π-calculus weak late bisimilarity (see Section 13.5.4)
$\mathscr{A}$	interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1)
ack	Ackermann function (see Example 4.18)
Aexp	set of IMP arithmetic expressions (see Chapter 3)
$\mathscr{B}$	interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2)
Bexp	set of IMP boolean expressions (see Chapter 3)
$\mathbb{B}$	set of booleans
$\mathscr{C}$	interpretation function for the denotational semantics of IMP commands (see Section 6.2.3)
CCS	Calculus of Communicating Systems (see Chapter 11)
Com	set of IMP commands (see Chapter 3)
CPO	Complete Partial Order (see Definition 5.11)
$\text{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.1.2)
CTMC	Continuous Time Markov Chain (see Definition 14.15)

DRAFT

DTMC	Discrete Time Markov Chain (see Definition **??**)
Env	set of HOFL environments (see Chapter 9)
fix	(least) fixpoint (see Definition 5.2.2)
FIX	(greatest) fixpoint
gcd	greatest common divisor
HML	Hennessy-Milner modal Logic (see Section 11.5)
HM-Logic	Hennessy-Milner modal Logic (see Section 11.5)
HOFL	A Higher-Order Functional Language (see Chapter 7)
IMP	A simple IMPerative language (see Chapter 3)
int	integer type in HOFL (see Definition 7.2)
Loc	set of locations (see Chapter 3)
LTL	Linear Temporal Logic (see Section 12.1.1)
LTS	Labelled Transition System (see Definition 11.2)
lub	least upper bound (see Definition 5.7)
$\mathbb{N}$	set of natural numbers
$\mathscr{P}$	set of closed CCS processes (see Definition 11.1)
PEPA	Performance Evaluation Process Algebra (see Chapter 16)
Pf	set of partial functions on natural numbers (see Example 5.10)
PI	set of partial injective functions on natural numbers (see Problem 5.11)
PO	Partial Order (see Definition 5.1)
PTS	Probabilistic Transition System (see Section 14.3.2)
$\mathbb{R}$	set of real numbers
$\mathscr{T}$	set of HOFL types (see Definition 7.2)
Tf	set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.11)
Var	set of HOFL variables (see Chapter 7)
$\mathbb{Z}$	set of integers

DRAFT

Part II
IMP: a simple imperative language

DRAFT

This part focuses on models for sequential computations that are associated to IMP, a simple imperative language. The syntax and natural semantics of IMP are studied in Chapter 3, while its denotational semantics is presented in Chapter 6, where it is also reconciled with the operational semantics. Chapter 4 explains several induction principles exploited to prove properties of programs and semantics. Chapter 5 fixes the mathematical basis of denotational semantics. The concepts in Chapters 4 and 5 are extensively used in Chapter 6 and in the rest of the monograph.

DRAFT

Chapter 3
Operational Semantics of IMP

Programs must be written for people to read, and only incidentally for machines to execute. (H. Abelson and G. Sussman)

Abstract This chapter introduces the formal syntax and operational semantics of a simple, structured imperative language called IMP, with static variable allocation and no sophisticated declaration constructs for data types, functions, classes, methods and the like. The operational semantics is defined in the natural style and it assumes an abstract machine with a vary basic form of memory to associate integer value to variables. The operational semantics is used to derive a notion of program equivalence and several example of (in)equivalence proofs are shown.

3.1 Syntax of IMP

The IMP programming language is a simple imperative language (e.g., it can be seen as a bare bone version of the C language) with only three data types:

int: the set of integer numbers, ranged over by metavariables $m, n, \ldots$

$$\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$$

bool: the set of boolean values, ranged over by metavariables $u, v, \ldots$

$$\mathbb{B} = \{\ \mathbf{true},\ \mathbf{false}\ \}$$

locations: the (denumerable) set of memory locations (we consider programs that use a finite number of locations and we assume there are enough locations available for any program), ranged over by metavariables $x, y, \ldots$

Loc *locations*

The grammar for IMP comprises three syntactic categories:

Aexp: Arithmetic expressions, ranged over by $a, a', \ldots$
Bexp: Boolean expressions, ranged over by $b, b', \ldots$

DRAFT

Com : Commands, ranged over by $c, c', \ldots$

Definition 3.1 (IMP: syntax). The following productions define the syntax of IMP:

$$\begin{array}{rcl} a \in Aexp & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ b \in Bexp & ::= & v \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1 \\ c \in Com & ::= & \mathbf{skip} \mid x := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c \end{array}$$

where we recall that n is an integer number, v a boolean value and x a location.

IMP is a very simple imperative language and there are several constructs we deliberately omit. For example we omit other common conditional statements, like *switch*, and other cyclic constructs like *repeat*. Moreover IMP commands imposes a structured flow of control, i.e., IMP has no labels, no goto statements, no break statements, no continue statements. Other things which are missing and are difficult to model are those concerned with *modular programming*. In particular, we have no *procedures*, no *modules*, no *classes*, no *types*. Since IMP does not include variable declarations, procedures and blocks, memory allocation is essentially static and finite. Of course, IMP has no *concurrent programming* construct.

3.1.1 Arithmetic Expressions

An arithmetic expression can be an integer number, or a location, a sum, a difference or a product. We notice that we do not have division, because it can be undefined (e.g., $7/0$) or give different values (e.g., $0/0$) so that its use would introduce unnecessary complexity.

3.1.2 Boolean Expressions

A boolean expression can be a logical value v, or the equality of an arithmetic expression with another, an arithmetic expression less or equal than another one, a negation, a logical conjunction or disjunction.

To keep the notation compact, in the following we will take the liberty of writing boolean expressions such as $x \neq 0$, in place of $\neg(x = 0)$ and $x > 0$ in place of $1 \leq x$ or $(0 \leq x) \wedge \neg(x = 0)$.

3.1.3 Commands

A command can be **skip**, i.e. a command which is not doing anything, or an assignment where we have that an arithmetic expression is evaluated and the value is

DRAFT

assigned to a location; we can also have the sequential execution of two commands (one after the other); an **if-then-else** with the obvious meaning: we evaluate a boolean expression b, if it is true we execute c_0 and if it is false we execute c_1. Finally we have a **while** statement, which is a command that keeps executing c until b becomes false.

3.1.4 Abstract Syntax

The notation above gives the so-called *abstract syntax* in that it simply says how to build up new expressions and commands but it is ambiguous for parsing a string. It is the job of the *concrete syntax* to provide enough information through parentheses or orders of precedence between operation symbols for a string to parse uniquely. It is helpful to think of a term in the abstract syntax as a specific parse tree of the language.

Example 3.1 (Valid expressions).

$(\textbf{while}\ b\ \textbf{do}\ c_1)\ ;\ c_2$	is a valid command;
$\textbf{while}\ b\ \textbf{do}\ (c_1\ ;\ c_2)$	is a valid command;
$\textbf{while}\ b\ \textbf{do}\ c_1\ ;\ c_2$	is not a valid command, because it is ambiguous.

In the following we will assume that enough parentheses have been added to resolve any ambiguity in the syntax. Then, given any formula of the form $a \in Aexp$, $b \in Bexp$, or $c \in Com$, the process to check if such formula is a “theorem” is deterministic (no backtracking is needed).

Example 3.2 (Validity check). Let us consider the formula:

$$\textbf{if}\ (x=0)\ \textbf{then}\ (\textbf{skip})\ \textbf{else}\ (x := (x-1)) \in Com$$

We can prove its validity by the following (deterministic) derivation, where we write $\nwarrow^*$ to mean that several derivation steps are grouped into one for brevity:

$$\begin{aligned}
\textbf{if}\,(x=0)\,\textbf{then}\,(\textbf{skip})\,\textbf{else}\,(x:=(x-1)) \in Com \ &\nwarrow\ x=0 \in Bexp,\ \textbf{skip} \in Com,\\
&\ \ x:=(x-1) \in Com\\
&\nwarrow\ x \in Aexp,\ 0 \in Aexp,\ \textbf{skip} \in Com,\\
&\ \ x:=(x-1) \in Com\\
&\nwarrow^*\ x-1 \in Aexp\\
&\nwarrow\ x \in Aexp, 1 \in Aexp\\
&\nwarrow^*\ \Box
\end{aligned}$$

DRAFT

3.2 Operational Semantics of IMP

3.2.1 Memory State

In order to define the evaluation of an expression or the execution of a command, we need to handle the state of the machine which is going to execute the IMP statements. Beside expressions to be evaluated and commands to be executed, we also need to record in the state some additional elements like values and stores. To this aim, we introduce the notion of *memory*:

$$\sigma \in \Sigma = (\mathbf{Loc} \to \mathbb{Z})$$

A memory σ is an element of the set Σ which contains all the functions from locations to integer numbers. A particular σ is just a function from locations to integer numbers so it is a function which associates to each location x the value $\sigma(x)$ that x stores.

Since **Loc** is an infinite set, things can be complicated: handling functions from an infinite set is not a good idea for a model of computation. Although **Loc** is large enough to store all the values that are manipulated by expressions and commands, the functions we are interested in are functions which are almost everywhere 0, except for a finite subset of memory locations.

If, for instance, we want to represent a memory such that the location x contains the value 5 and the location y the value 10 and elsewhere is stored 0, we write:

$$\sigma = (5/x,\ 10/y)$$

In this way we can represent any interesting memory by a finite set of pairs.

We let $()$ denote the memory such that all locations are assigned the value 0.

Definition 3.2 (Memory update). Given a memory σ, we denote by $\sigma[^n/_x]$ the memory where the value of x is updated to n, i.e. such that

$$\sigma[^n/_x](y) = \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{if } y \neq x \end{cases}$$

Note that $\sigma[^n/_x][^m/_x] = \sigma[^m/_x]$. In fact:

$$\sigma[^n/_x][^m/_x](y) = \begin{cases} m & \text{if } y = x \\ \sigma[^n/_x](y) = \sigma(y) & \text{if } y \neq x \end{cases}$$

Moreover, when $x \neq y$, then the order of updates is not important, i.e., $\sigma[^n/_x][^m/_y] = \sigma[^m/_y][^n/_x]$. For this reason, we often use the more compact notation $\sigma[^n/_x, ^m/_y]$.

DRAFT

3.2.2 *Inference Rules*

Now we are going to give the *operational semantics* to IMP using a logical system. It is called "big-step" semantics (see Section 1.2.3) because it leads to the result in one single proof.

We are interested in three kinds of *well formed formulas*:

Arithmetic expressions: The evaluation of an element $a \in Aexp$ in a given memory σ results in an integer number.

$$\langle a, \sigma \rangle \rightarrow n$$

Boolean expressions: The evaluation of an element $b \in Bexp$ in a given memory σ results in either **true** or **false**.

$$\langle b, \sigma \rangle \rightarrow v$$

Commands: The evaluation of an element $c \in Com$ in a given memory σ leads to an updated final state σ'.

$$\langle c, \sigma \rangle \rightarrow \sigma'$$

Next we show each inference rule and comment on it.

3.2.2.1 Inference Rules for Arithmetic Expressions

We start with the rules about arithmetic expressions.

$$\frac{}{\langle n, \sigma \rangle \rightarrow n} \text{ (num)} \tag{3.1}$$

The axiom 3.1 (num) is trivial: the evaluation of any numerical constant n (seen as syntax) results in the corresponding integer value n (read as an element of the semantic domain) no matter which σ.

$$\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)} \text{ (ide)} \tag{3.2}$$

The axiom 3.2 (ide) is also quite intuitive: the evaluation of an identifier x in the memory σ results in the value stored in x.

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n} \; n = n_0 + n_1 \text{ (sum)} \tag{3.3}$$

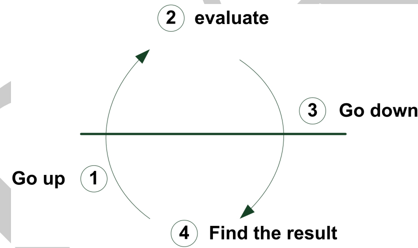
DRAFT

The rule 3.3 (sum) has several premises: the evaluation of the syntactic expression $a_0 + a_1$ in σ returns a value n that corresponds to the arithmetic sum of the values n_0 and n_1 obtained after evaluating, respectively, a_0 and a_1 in σ. Note that we exploit the side condition $n = n_0 + n_1$ to indicate the relation between the target n of the conclusion and the targets of the premises. We present an equivalent, but more compact, version of the rule (sum), where the target of the conclusion is obtained as the sum of the targets of the premises. In the following we shall adopt the second format (3.4).

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_0 + n_1} \text{ (sum)} \tag{3.4}$$

We remark the difference between the two occurrences of the symbol $+$ in the rule: in the source of the conclusion (i.e., $a_0 + a_1$) it denotes a piece of syntax, in the target of the conclusion (i.e., $n_0 + n_1$) it denotes a semantic operation. To avoid any ambiguity we could have introduced different symbols in the two cases, but we have preferred to overload the symbol and keep the notation simpler. We hope the reader is expert enough to assign the right meaning to each occurrence of overloaded symbols by looking at the context in which they appear.

The way we read this rule is very interesting because, in general, if we want to evaluate the lower part we have to go up, evaluate the uppermost part and then compose the results and finally go down again to draw the conclusion:



In this case we suppose we want to evaluate, in the memory σ, the arithmetic expression $a_0 + a_1$. We have to evaluate a_0 in the same memory σ and get n_0, then we have to evaluate a_1 within the same memory σ to get n_1 and then the final result will be $n_0 + n_1$. Note that the same memory σ is duplicated and distributed to the two evaluations of a_0 and a_1, which may occur independently in any order.

This kind of mechanism is very powerful because we deal with more proofs at once. First, we evaluate a_0. Second, we evaluate a_1. Then, we put all together. If we need to evaluate several expressions on a sequential machine we have to deal with the issue of fixing the order in which to proceed. On the other hand, in this case, using a logical language we just model the fact that we want to evaluate a tree (an expression) which is a tree of proofs in a very simple way and make explicit that the order is not important.

DRAFT

The rules for the remaining arithmetic expressions are similar to the one for the sum. We report them for completeness, but do not comment on them.

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n_0 - n_1} \text{ (dif)} \tag{3.5}$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n_0 \times n_1} \text{ (prod)} \tag{3.6}$$

3.2.2.2 Inference Rules for Boolean Expressions

The rules for boolean expressions are also similar to the previous ones and need no particular comment, except for noting that the premises of rules (equ) and (leq) refer the judgements of arithmetic expressions.

$$\frac{}{\langle v, \sigma \rangle \to v} \text{ (bool)} \tag{3.7}$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 = a_1, \sigma \rangle \to (n_0 = n_1)} \text{ (equ)} \tag{3.8}$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \leq a_1, \sigma \rangle \to (n_0 \leq n_1)} \text{ (leq)} \tag{3.9}$$

$$\frac{\langle b, \sigma \rangle \to v}{\langle \neg b, \sigma \rangle \to \neg v} \text{ (not)} \tag{3.10}$$

$$\frac{\langle b_0, \sigma \rangle \to v_0 \quad \langle b_1, \sigma \rangle \to v_1}{\langle b_0 \vee b_1, \sigma \rangle \to (v_0 \vee v_1)} \text{ (or)} \tag{3.11}$$

$$\frac{\langle b_0, \sigma \rangle \to v_0 \quad \langle b_1, \sigma \rangle \to v_1}{\langle b_0 \wedge b_1, \sigma \rangle \to (v_0 \wedge v_1)} \text{ (and)} \tag{3.12}$$

DRAFT

3.2.2.3 Inference Rules for Commands

Next, we move to the inference rules for commands.

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma} \text{ (skip)} \tag{3.13}$$

The rule 3.13 (skip) is very simple: it leaves the memory σ unchanged.

$$\frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma[^m/_x]} \text{ (assign)} \tag{3.14}$$

The rule 3.14 (assign) exploits the assignment operation to update σ: we remind that $\sigma[^m/_x]$ is the same memory as σ except for the value assigned to x (m instead of $\sigma(x)$). Note that the premise refers to the judgements of arithmetic expressions.

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'} \text{ (seq)} \tag{3.15}$$

The rule 3.15 (seq) for the sequential composition (concatenation) of commands is quite interesting. We start by evaluating the first command c_0 in the memory σ. As a result we get an updated memory σ'' which we use for evaluating the second command c_1. In fact the order of evaluation of the two command is important and it would not make sense to evaluate c_1 in the original memory σ, because the effects of executing c_0 would be lost. Finally, the memory σ' obtained by evaluating c_1 in σ'' is returned as the result of evaluating $c_0; c_1$ in σ.

The conditional statement requires two different rules, that depend on the evaluation of the condition b (they are mutually exclusive).

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'} \text{ (iftt)} \tag{3.16}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'} \text{ (ifff)} \tag{3.17}$$

The rule 3.16 (iftt) checks that b evaluated to true and then returns as result the memory σ' obtained by evaluating the command c_0 in σ. On the contrary, the rule 3.17 (ifff) checks that b evaluated to false and then returns as result the memory σ' obtained by evaluating the command c_1 in σ.

Also the while statement requires two different rules, that depends on the evaluation of the guard b; they are mutually exclusive.

DRAFT

$$\frac{\langle b,\sigma\rangle \to \mathbf{true} \quad \langle c,\sigma\rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c,\sigma''\rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c,\sigma\rangle \to \sigma'}\ (\text{whtt}) \tag{3.18}$$

$$\frac{\langle b,\sigma\rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c,\sigma\rangle \to \sigma}\ (\text{whff}) \tag{3.19}$$

The rule 3.18 (whtt) applies to the case where the guard evaluates to true: we need to compute the memory σ'' obtained by the evaluation of the body c in σ and then to iterate the evaluation of the cycle over σ''.

The rule 3.19 (whff) applies to the case where the guard evaluates to false: then the cycle terminates and the memory σ is returned unchanged.

Remark 3.1. There is an important difference between the rule 3.18 and all the other inference rules we have encountered so far. All the other rules take as premises formulas that are "smaller in size" than their conclusions. This fact allows to decrease the complexity of the atomic goals to be proved as the derivation proceeds further, until having basic formulas to which axioms can be applied. The rule 3.18 is different because it recursively uses as a premise a formula as complex as its conclusion. This justifies the fact that a while command can cycle indefinitely, without terminating.

The set of all inference rules above defines the operational semantics of IMP. Formally, they induce a relation that contains all the pairs input-result, where the input is the expression / command together with the initial memory and the result is the corresponding evaluation:

$$\to \subseteq (Aexp \times \Sigma \times \mathbb{Z}) \cup (Bexp \times \Sigma \times \mathbb{B}) \cup (Com \times \Sigma \times \Sigma)$$

We will see later that the computation is deterministic, in the sense that given any expression / commands and any memory as input there is at most one result (exactly one in case of arithmetic and boolean expressions).

3.2.3 Examples

Example 3.3 (Semantic evaluation of a command). Let us consider the (extra-bracketed) command

$$c \stackrel{\text{def}}{=} (x := 0)\ ;\ (\ \mathbf{while}\ (0 \le y)\ \mathbf{do}\ (\ (x := ((x + (2 \times y)) + 1))\ ;\ (y := (y - 1))\ )\ )$$

To improve readability and without introducing too much ambiguity, we can write it as follows:

$$c \stackrel{\text{def}}{=} x := 0\ ;\ \mathbf{while}\ 0 \le y\ \mathbf{do}\ (\ x := x + (2 \times y) + 1\ ;\ y := y - 1\ )$$

DRAFT

or exploiting the usual convention for indented code, as:

$$
\begin{array}{l}
c \stackrel{\text{def}}{=} x := 0\ ; \\
\quad \textbf{while}\ 0 \leq y\ \textbf{do}\ (\\
\qquad x := x + (2 \times y) + 1\ ; \\
\qquad y := y - 1 \\
\quad)
\end{array}
$$

Without too much difficulties, the experienced reader can guess the relation between the value of y at the beginning of the execution and that of x at the end of the execution: The program computes the square of (the value initially stored in) y plus 1 (when $y \geq 0$) and stores it in x. In fact, by exploiting the well-known equalities $0^2 = 0$ and $(n+1)^2 = n^2 + 2n + 1$, the value of $(y+1)^2$ is computed as the sum of the first $y+1$ odd numbers $\sum_{i=0}^{y}(2i+1)$. For example, for $y = 3$ we have $4^2 = 1+3+5+7 = 16$.

We report below the proof of well-formedness of the command, as a witness that c respects the syntax of IMP. (Of course the inference rules used in the derivation are those associated to the productions of the grammar of IMP.)

$$
\frac{
\dfrac{\overline{x} \quad \overline{0}}{x := 0}
\quad
\dfrac{
\dfrac{\overline{0} \quad \overline{y}}{0 \leq y}
\quad
\dfrac{
\dfrac{
\overline{x}
\quad
\dfrac{
\dfrac{\overline{x} \quad \dfrac{\overline{2} \quad \overline{y}}{(2 \times y)}}{a_1 \stackrel{\text{def}}{=} (x + (2 \times y))}
\quad \overline{1}
}{a \stackrel{\text{def}}{=} ((x + (2 \times y)) + 1)}
}{c_3 \stackrel{\text{def}}{=} (x := ((x + (2 \times y)) + 1))}
\quad
\dfrac{\overline{y} \quad \dfrac{\overline{y} \quad \overline{1}}{(y-1)}}{c_4 \stackrel{\text{def}}{=} (y := (y-1))}
}{c_2 \stackrel{\text{def}}{=} ((x := ((x + (2 \times y)) + 1)); (y := y - 1))}
}{c_1 \stackrel{\text{def}}{=} (\textbf{while}(0 \leq y)\,\textbf{do}((x := ((x + (2 \times y)) + 1)); (y := (y-1))))}
}{c \stackrel{\text{def}}{=} ((x := 0); (\textbf{while}(0 \leq y)\,\textbf{do}((x := ((x + (2 \times y)) + 1)); (y := (y+1)))))}
$$

We can summarize the above proof as follows, introducing several shorthands for referring to some subterms of c that will be useful later.

$$
\underbrace{x := 0; \underbrace{\textbf{while}\, 0 \leq y\, \textbf{do}\, \underbrace{(\underbrace{x := \overbrace{\overbrace{x + (2 \times y)}^{a_1} + 1}^{a}}_{c_3}; \underbrace{y := y - 1}_{c_4})}_{c_2}}_{c_1}}_{c}
$$

To find the semantics of c in a given memory we proceed in the goal-oriented fashion. For instance, we take the well-formed formula $\langle c, (^{27}/_x, ^2/_y)\rangle \to \sigma$, with σ unknown, and check if there exists a memory σ such that the formula becomes a theorem. This is equivalent to find an answer to the following question: "given the initial memory $(^{27}/_x, ^2/_y)$ and the command c to be executed, can we find a derivation that leads to some memory σ?" By answering in the affirmative, we would have a proof of termination for c and would establish the content of the memory at the end of the computation.

To convince the reader that the notation for goal-oriented derivations introduced in Section 2.3 is more effective than the tree-like notation, we first show the proof in the tree-like notation: the goal to prove is the root (situated at the bottom) and the "pieces" of derivation are added on top. As the tree grows rapidly large, we split the derivation in smaller pieces that are proved separately. We use "?" to mark the missing parts of the derivations.

$$\dfrac{\dfrac{\dfrac{}{\langle 0, (^{27}/_x, ^2/_y)\rangle \to 0}\ \text{num}}{\langle x := 0, (^{27}/_x, ^2/_y)\rangle \to (^{27}/_x, ^2/_y)\,[^0/_x] = \sigma_1}\ \text{assign} \qquad \dfrac{?}{\langle c_1, \sigma_1\rangle \to \sigma}}{\langle c, (^{27}/_x, ^2/_y)\rangle \to \sigma}\ \text{seq}$$

Note that c_1 is a cycle, therefore we have two possible rules that can be applied, depending on the evaluation of the guard. We only show the successful derivation, recalling that $\sigma_1 = (^{27}/_x, ^2/_y)\,[^0/_x] = (^0/_x, ^2/_y)$.

$$\dfrac{\dfrac{\dfrac{}{\langle 0, \sigma_1\rangle \to 0}\ \text{num} \quad \dfrac{}{\langle y, \sigma_1\rangle \to \sigma_1(y) = 2}\ \text{ide}}{\langle 0 \le y, \sigma_1\rangle \to (0 \le 2) = \mathbf{true}}\ \text{leq} \qquad \dfrac{?}{\langle c_2, \sigma_1\rangle \to \sigma_2} \qquad \dfrac{?}{\langle c_1, \sigma_2\rangle \to \sigma}}{\langle c_1, \sigma_1\rangle \to \sigma}\ \text{whtt}$$

Next we need to prove the goals $\langle c_2, (^0/_x, ^2/_y)\rangle \to \sigma_2$ and $\langle c_1, \sigma_2\rangle \to \sigma$. Let us focus on $\langle c_2, \sigma_1\rangle \to \sigma_2$ first:

$$\dfrac{\dfrac{\dfrac{\dfrac{?}{\langle a_1, (^0/_x, ^2/_y)\rangle \to m'} \quad \dfrac{}{\langle 1, (^0/_x, ^2/_y)\rangle \to 1}\ \text{num}}{\langle a, (^0/_x, ^2/_y)\rangle \to m = m' + 1}\ \text{sum}}{\langle c_3, (^0/_x, ^2/_y)\rangle \to (^0/_x, ^2/_y)\,[^m/_x] = \sigma_3}\ \text{assign} \qquad \dfrac{\dfrac{?}{\langle y - 1, \sigma_3\rangle \to m''}}{\langle c_4, \sigma_3\rangle \to \sigma_3\left[^{m''}/_y\right] = \sigma_2}\ \text{assign}}{\langle c_2, (^0/_x, ^2/_y)\rangle \to \sigma_2}\ \text{seq}$$

We show separately the details for the pending derivations of $\langle a_1, (^0/_x, ^2/_y)\rangle \to m'$ and $\langle y - 1, \sigma_3\rangle \to m''$:

DRAFT

$$\dfrac{\dfrac{}{\langle x, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 0}\ \text{ide} \qquad \dfrac{\dfrac{}{\langle 2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\ \text{num} \qquad \dfrac{}{\langle y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\ \text{ide}}{\langle 2 \times y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m''' = 2 \times 2 = 4}\ \text{prod}}{\langle a_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \to m' = 0 + 4 = 4}\ \text{sum}$$

Since $m' = 4$, then it means that $m = m' + 1 = 5$ and hence $\sigma_3 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{5}/_{x}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)$.

$$\dfrac{\dfrac{}{\langle y, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to 2}\ \text{ide} \qquad \dfrac{}{\langle 1, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to 1}\ \text{num}}{\langle y - 1, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \to m'' = 2 - 1 = 1}\ \text{dif}$$

Since $m'' = 1$ we know that $\sigma_2 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{m''}/_{y}\right] = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{1}/_{y}\right] = \left({}^{5}/_{x}, {}^{1}/_{y}\right)$.

Next we prove $\langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma$, this time omitting some details (the derivation is analogous to the one just seen).

$$\dfrac{\dfrac{\vdots}{\langle 0 \leq y, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \mathbf{true}}\ \text{leq} \qquad \dfrac{\vdots}{\langle c_2, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \left({}^{8}/_{x}, {}^{0}/_{y}\right) = \sigma_4}\ \text{seq} \qquad \dfrac{?}{\langle c_1, \sigma_4\rangle \to \sigma}}{\langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \to \sigma}\ \text{whtt}$$

Hence $\sigma_4 = \left({}^{8}/_{x}, {}^{0}/_{y}\right)$ and next we prove $\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma$.

$$\dfrac{\dfrac{\vdots}{\langle 0 \leq y, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \mathbf{true}}\ \text{leq} \qquad \dfrac{\vdots}{\langle c_2, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right) = \sigma_5}\ \text{seq} \qquad \dfrac{?}{\langle c_1, \sigma_5\rangle \to \sigma}}{\langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \to \sigma}\ \text{whtt}$$

Hence $\sigma_5 = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)$. Finally:

$$\dfrac{\dfrac{\vdots}{\langle 0 \leq y, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \mathbf{false}}\ \text{leq}}{\langle c_1, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right) = \sigma}\ \text{whff}$$

Summing up all the above, we have proved the theorem:

$$\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \to \left({}^{9}/_{x}, {}^{-1}/_{y}\right).$$

DRAFT

It is evident that as the proof tree grows larger it gets harder to paste the different pieces of the proof together. We now show the same proof as a goal-oriented derivation, which should be easier to follow. To this aim, we group several derivation steps into a single one omitting trivial steps.

$$
\begin{array}{rl}
\langle c, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \sigma \quad \nwarrow & \langle x := 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \sigma_1, \quad \langle c_1, \sigma_1\rangle \rightarrow \sigma \\
\nwarrow_{\sigma_1 = \left({}^{27}/_{x}, {}^{2}/_{y}\right)\left[{}^{n}/_{x}\right]} & \langle 0, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow n, \\
& \langle c_1, \left({}^{27}/_{x}, {}^{2}/_{y}\right)\left[{}^{n}/_{x}\right]\rangle \rightarrow \sigma \\
\nwarrow_{n=0\ \sigma_1 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)} & \langle c_1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \sigma \\
\nwarrow & \langle 0 \leq y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \mathbf{true}, \\
& \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \sigma_2, \ \langle c_1, \sigma_2\rangle \rightarrow \sigma \\
\nwarrow & \langle 0, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow n_1, \quad \langle y, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow n_2, \\
& n_1 \leq n_2, \ \langle c_2, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \sigma_2, \ \langle c_1, \sigma_2\rangle \rightarrow \sigma \\
\nwarrow^{*}_{n_1=0\ n_2=2} & \langle c_3, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \sigma_3, \quad \langle c_4, \sigma_3\rangle \rightarrow \sigma_2, \\
& \langle c_1, \sigma_2\rangle \rightarrow \sigma \\
\nwarrow_{\sigma_3 = \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{m}/_{x}\right]} & \langle x + (2 \times y) + 1, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow m, \\
& \langle c_4, \left({}^{0}/_{x}, {}^{2}/_{y}\right)\left[{}^{m}/_{x}\right]\rangle \rightarrow \sigma_2, \ \langle c_1, \sigma_2\rangle \rightarrow \sigma \\
\nwarrow^{*}_{m=0+(2\times 2)+1=5\ \sigma_3 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)} & \langle c_4, \left({}^{5}/_{x}, {}^{2}/_{y}\right)\rangle \rightarrow \sigma_2, \quad \langle c_1, \sigma_2\rangle \rightarrow \sigma \\
\nwarrow^{*}_{\sigma_2 = \left({}^{5}/_{x}, {}^{2}/_{y}\right)\left[{}^{1}/_{y}\right] = \left({}^{5}/_{x}, {}^{1}/_{y}\right)} & \langle c_1, \left({}^{5}/_{x}, {}^{1}/_{y}\right)\rangle \rightarrow \sigma \\
\nwarrow^{*}_{\sigma_4 = \left({}^{5}/_{x}, {}^{1}/_{y}\right)\left[{}^{8}/_{x}\right]\left[{}^{0}/_{y}\right] = \left({}^{8}/_{x}, {}^{0}/_{y}\right)} & \langle c_1, \left({}^{8}/_{x}, {}^{0}/_{y}\right)\rangle \rightarrow \sigma \\
\nwarrow^{*}_{\sigma_5 = \left({}^{8}/_{x}, {}^{0}/_{y}\right)\left[{}^{9}/_{x}\right]\left[{}^{-1}/_{y}\right] = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)} & \langle c_1, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \rightarrow \sigma \\
\nwarrow_{\sigma = \left({}^{9}/_{x}, {}^{-1}/_{y}\right)} & \langle 0 \leq y, \left({}^{9}/_{x}, {}^{-1}/_{y}\right)\rangle \rightarrow \mathbf{false} \\
\nwarrow^{*} & \square
\end{array}
$$

There are commands c and memories σ such that there is no σ' for which we can find a proof of $\langle c, \sigma\rangle \rightarrow \sigma'$. We use the notation below to denote such cases:

$$\langle c, \sigma\rangle \not\rightarrow \quad \text{iff} \quad \neg\exists \sigma'. \langle c, \sigma\rangle \rightarrow \sigma'$$

The condition $\neg\exists \sigma'. \langle c, \sigma\rangle \rightarrow \sigma'$ can be written equivalently as $\forall \sigma'. \langle c, \sigma\rangle \not\rightarrow \sigma'$.

Example 3.4 (Non termination). Let us consider the command

$$c \stackrel{\text{def}}{=} \mathbf{while\ true\ do\ skip}$$

Given σ, the only possible derivation goes as follows:

DRAFT

$$
\begin{array}{rl}
\langle c,\sigma\rangle \to \sigma' \quad \nwarrow & \langle \mathbf{true},\sigma\rangle \to \mathbf{true},\ \langle \mathbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\
\nwarrow & \langle \mathbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\
\nwarrow_{\sigma_1=\sigma} & \langle c,\sigma\rangle \to \sigma'
\end{array}
$$

After a few steps of derivation we reach the same goal from which we started and there are no alternatives to try!

In this case, we can prove that $\langle c,\sigma\rangle \not\to$. We proceed by contradiction, assuming there exists σ' for which we can find a (finite) derivation d for $\langle c,\sigma\rangle \to \sigma'$. Let d be the derivation sketched below:

$$
\begin{array}{rl}
\langle c,\sigma\rangle \to \sigma' \nwarrow & \langle \mathbf{true},\sigma\rangle \to \mathbf{true},\ \langle \mathbf{skip},\sigma\rangle \to \sigma_1,\ \langle c,\sigma_1\rangle \to \sigma' \\
\vdots & \\
(*) \nwarrow & \langle c,\sigma\rangle \to \sigma' \\
\vdots & \\
\nwarrow & \square
\end{array}
$$

We have marked by $(*)$ the last occurrence of the goal $\langle c,\sigma\rangle \to \sigma'$. But this leads to a contradiction, because the next step of the derivation can only be obtained by applying rule (whtt) and therefore it should lead to another instance of the original goal.

3.3 Abstract Semantics: Equivalence of Expressions and Commands

The same way as we can write different expressions denoting the same value, we can write different programs for solving the same problem. For example we are used not to distinguish between say $2+2$ and 2×2 because both evaluate to 4. Similarly, would you distinguish between, say, $x := 1; y := 0$ and $y := 0; x := y+1$? So a natural question arise: when are two programs "equivalent"? The equivalence between two commands is an important issue because it allows, e.g., to replace a program with an equivalent but more efficient one. Informally, two programs are equivalent if they *behave in the same way*. But can we make this idea more precise?

Since the evaluation of a command depends on the memory, two equivalent programs must behave the same *w.r.t. any initial memory*. For example the two commands $x := 1$ and $x := y+1$ assign the same value to x only when evaluated in a memory σ such that $\sigma(y) = 0$, so that it wouldn't be safe to replace one for the other in any program. Moreover, we must take into account that commands may diverge when evaluated with a certain memory, like **while** $x \geq 0$ **do** $x := x-1$ when evaluated in a store σ such that $\sigma(x) < 0$. We will call *abstract semantics* the notion of behaviour w.r.t. we will compare programs for equivalence.

DRAFT

The operational semantics offers a straightforward abstract semantics: two programs are equivalent if they result in the same memory when evaluated over the same initial memory.

Definition 3.3 (Equivalence of expressions and commands). We say that the arithmetic expressions a_1 and a_2 are *equivalent*, written $a_1 \sim a_2$ if and only if for any memory σ they evaluate in the same way. Formally:

$$a_1 \sim a_2 \quad \text{iff} \quad \forall \sigma, n. (\ \langle a_1, \sigma \rangle \to n \ \Leftrightarrow\ \langle a_2, \sigma \rangle \to n\ )$$

We say that the boolean expressions b_1 and b_2 are *equivalent*, written $b_1 \sim b_2$ if and only if for any memory σ they evaluate in the same way. Formally:

$$b_1 \sim b_2 \quad \text{iff} \quad \forall \sigma, v. (\ \langle b_1, \sigma \rangle \to v \ \Leftrightarrow\ \langle b_2, \sigma \rangle \to v\ )$$

We say that the commands c_1 and c_2 are *equivalent*, written $c_1 \sim c_2$ if and only if for any memory σ they evaluate in the same way. Formally:

$$c_1 \sim c_2 \quad \text{iff} \quad \forall \sigma, \sigma'. (\ \langle c_1, \sigma \rangle \to \sigma' \ \Leftrightarrow\ \langle c_2, \sigma \rangle \to \sigma'\ )$$

Note that if the evaluation of $\langle c_1, \sigma \rangle$ diverges there is no σ' such that $\langle c_1, \sigma \rangle \to \sigma'$. Then, when $c_1 \sim c_2$, the double implication prevents $\langle c_2, \sigma \rangle$ to converge. As an easy consequence, any two programs that diverge for any σ are equivalent.

3.3.1 Examples: Simple Equivalence Proofs

The first example we show is concerned with fully specified programs that operate on unspecified memories.

Example 3.5 (Equivalent commands). Let us try to prove that the following two commands are equivalent:

$$c_1 \stackrel{\text{def}}{=} \textbf{while}\, x \neq 0\, \textbf{do}\, x := 0$$
$$c_2 \stackrel{\text{def}}{=} x := 0$$

It is immediate to prove that

$$\forall \sigma. \langle c_2, \sigma \rangle \to \sigma' = \sigma[0/x]$$

Hence σ and σ' can differ only for the value stored in x. In particular, if $\sigma(x) = 0$ then $\sigma' = \sigma$.

The evaluation of c_1 in σ depends on $\sigma(x)$: if $\sigma(x) = 0$ we must apply the rule 3.19 (whff), otherwise the rule 3.18 (whtt) must be applied. Since we do not know the value of $\sigma(x)$, we consider the two cases separately. The corresponding hypotheses are called *path conditions* and outline a very important technique for the symbolic analysis of programs.

DRAFT

Case $\sigma(x) \neq 0$) Let us inspect a possible derivation for $\langle c_1, \sigma\rangle \to \sigma'$. Since $\sigma(x) \neq 0$ we select the rule (iftt) at the first step:

$$\begin{array}{rll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow & \langle x \neq 0, \sigma\rangle \to \mathbf{true}, \quad \langle x := 0, \sigma\rangle \to \sigma_1, \\
& & \langle c_1, \sigma_1\rangle \to \sigma' \\
& \nwarrow^{*}_{\sigma_1 = \sigma\left[{}^{0}/{}_{x}\right]} & \langle c_1, \sigma\left[{}^{0}/{}_{x}\right]\rangle \to \sigma' \\
& \nwarrow_{\sigma' = \sigma\left[{}^{0}/{}_{x}\right]} & \langle x \neq 0, \sigma\left[{}^{0}/{}_{x}\right]\rangle \to \mathbf{false} \\
& \nwarrow^{*} & \sigma\left[{}^{0}/{}_{x}\right](x) = 0 \\
& \nwarrow & \square
\end{array}$$

Case $\sigma(x) = 0$) Let us inspect a derivation for $\langle c_1, \sigma\rangle \to \sigma'$. Since $\sigma(x) \neq 0$ we select the rule (ifff) at the first step:

$$\begin{array}{rll}
\langle c_1, \sigma\rangle \to \sigma' & \nwarrow_{\sigma' = \sigma} & \langle x \neq 0, \sigma\rangle \to \mathbf{false} \\
& \nwarrow^{*} & \sigma(x) = 0 \\
& \nwarrow & \square
\end{array}$$

Finally, we observe the following:

- If $\sigma(x) = 0$, then $\begin{cases} \langle c_1, \sigma\rangle \to \sigma \\ \langle c_2, \sigma\rangle \to \sigma[{}^{0}/{}_{x}] = \sigma \end{cases}$
- Otherwise, if $\sigma(x) \neq 0$, then $\begin{cases} \langle c_1, \sigma\rangle \to \sigma[{}^{0}/{}_{x}] \\ \langle c_2, \sigma\rangle \to \sigma[{}^{0}/{}_{x}] \end{cases}$

Therefore $c_1 \sim c_2$ because for any σ they result in the same memory.

The general methodology should be clear by now: in case the computation terminates we need just to develop the derivation and compare the results.

3.3.2 Examples: Parametric Equivalence Proofs

The programs considered so far were entirely spelled out: all the commands and expressions were given and the only unknown parameter was the initial memory σ. In this section we address equivalence proofs for programs that contain symbolic expressions a and b and symbolic commands c: we will need to prove that the equality holds for any such a, b and c.

This is not necessarily more complicated than what we have done already: the idea is that we can just carry the derivation with symbolic parameters.

Example 3.6 (Parametric proofs (1)). Let us consider the commands:

DRAFT

$$c_1 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$$
$$c_2 \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip} = \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}$$

Is it true that $\forall b \in Bexp, c \in Com.\ (c_1 \sim c_2)$?

We start by considering the derivation for c_1 in a generic initial memory σ. The command c_1 is a cycle and there are two rules we can apply: either the rule 3.19 (whff), or the rule 3.18 (whtt). Which rule to use depends on the evaluation of b. Since we do not know what b is, we must take into account both possibilities and consider the two cases separately.

$\langle b, \sigma \rangle \to \textbf{false}$) For c_1 we have:

$$\begin{aligned} \langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma' \quad & \nwarrow_{\sigma'=\sigma} \ \langle b, \sigma \rangle \to \textbf{false} \\ & \nwarrow \ \square \end{aligned}$$

For c_2 we have:

$$\begin{aligned} \langle \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}, \sigma \rangle \to \sigma' \quad & \nwarrow \ \langle b, \sigma \rangle \to \textbf{false}, \\ & \qquad \langle \textbf{skip}, \sigma \rangle \to \sigma' \\ & \nwarrow^{*}_{\sigma'=\sigma} \ \square \end{aligned}$$

It is evident that if $\langle b, \sigma \rangle \to \textbf{false}$ then the two derivations for c_1 and c_2 lead to the same result.

$\langle b, \sigma \rangle \to \textbf{true}$) For c_1 we have:

$$\begin{aligned} \langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma' \ & \nwarrow \ \langle b, \sigma \rangle \to \textbf{true}, \quad \langle c, \sigma \rangle \to \sigma_1, \\ & \qquad \langle c_1, \sigma_1 \rangle \to \sigma' \\ & \nwarrow \ \langle c, \sigma \rangle \to \sigma_1, \quad \langle c_1, \sigma_1 \rangle \to \sigma' \end{aligned}$$

We find it convenient to stop here the derivation, because otherwise we should add further hypotheses on the evaluation of c and of the guard b after the execution of c. Instead, let us look at the derivation of c_2:

$$\begin{aligned} \langle \textbf{if } b \textbf{ then } (c; c_1) \textbf{ else skip}, \sigma \rangle \to \sigma' \ & \nwarrow \ \langle b, \sigma \rangle \to \textbf{true}, \\ & \qquad \langle c; c_1, \sigma \rangle \to \sigma' \\ & \nwarrow \ \langle c; c_1, \sigma \rangle \to \sigma' \\ & \nwarrow \ \langle c, \sigma \rangle \to \sigma_1, \\ & \qquad \langle c_1, \sigma_1 \rangle \to \sigma' \end{aligned}$$

DRAFT

Now we can stop again, because we have reached exactly the same subgoals that we have obtained by evaluating c_1! It is then obvious that if $\langle b,\sigma\rangle \rightarrow$ **true** then the two derivations for c_1 and c_2 will necessarily lead to the same result whenever they terminate, and if one diverges the other diverges too.

Summing up the two cases, and since there are no more alternatives to try, we can conclude that $c_1 \sim c_2$.

Note that the equivalence proof technique that exploits reduction to the same subgoals is one of the most convenient methods for proving the equivalence of **while** commands, whose evaluation may diverge.

Example 3.7 (Parametric proofs (2)). Let us consider the commands:

$$c_1 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$$
$$c_2 \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } c_1 \textbf{ else skip}$$

Is it true that $\forall b \in Bexp, c \in Com.\ c_1 \sim c_2$?

We have already examined the different derivations for c_1 in the previous example. Moreover, the evaluation of c_2 when $\langle b,\sigma\rangle \rightarrow$ **false** is also analogous to that of the command c_2 in Example 3.6. Therefore we focus on the analysis of c_2 for the case $\langle b,\sigma\rangle \rightarrow$ **true**. Trivially:

$$\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else skip}, \sigma\rangle \rightarrow \sigma' \nwarrow \langle b,\sigma\rangle \rightarrow \textbf{true}, \quad \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightarrow \sigma'$$
$$\nwarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightarrow \sigma'$$

So we reduce to the subgoal identical to the evaluation of c_1, and we can conclude that $c_1 \sim c_2$.

3.3.3 Examples: Inequality Proofs

The next example deals with programs that can behave the same or exhibit different behaviours depending on the initial memory.

Example 3.8 (Inequality proof). Let us consider the commands:

$$c_1 \stackrel{\text{def}}{=} (\textbf{while } x > 0 \textbf{ do } x := 1); x := 0$$
$$c_2 \stackrel{\text{def}}{=} x := 0$$

Let us prove that $c_1 \not\sim c_2$.

For c_2 we have

DRAFT

$$\begin{array}{rl} \langle x := 0, \sigma \rangle \to \sigma' \nwarrow_{\sigma' = \sigma[n/x]} & \langle 0, \sigma \rangle \to n \\ \nwarrow_{n=0} & \square \end{array}$$

That is: $\forall \sigma.\ \langle x := 0, \sigma \rangle \to \sigma[0/x]$.
Next, we focus on the first part of c_1

$$w \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := 1$$

If $\sigma(x) \leq 0$ it is immediate to check that

$$\langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma \rangle \to \sigma$$

The derivation is sketched below:

$$\begin{array}{rl} \langle w, \sigma \rangle \to \sigma' \nwarrow_{\sigma' = \sigma} & \langle x > 0, \sigma \rangle \to \textbf{false} \\ \nwarrow & \langle x, \sigma \rangle \to n,\ \langle 0, \sigma \rangle \to m,\ n \leq m \\ \nwarrow_{n = \sigma(x)} & \langle 0, \sigma \rangle \to m,\ \sigma(x) \leq m \\ \nwarrow_{m=0} & \sigma(x) \leq 0 \\ \nwarrow & \square \end{array}$$

Instead, if we assume $\sigma(x) > 0$, then:

$$\begin{array}{rl} \langle w, \sigma \rangle \to \sigma' \quad \nwarrow & \langle x > 0, \sigma \rangle \to \textbf{true},\ \langle x := 1, \sigma \rangle \to \sigma'',\ \langle w, \sigma'' \rangle \to \sigma' \\ \nwarrow^* & \langle x := 1, \sigma \rangle \to \sigma'',\ \langle w, \sigma'' \rangle \to \sigma' \\ \nwarrow^*_{\sigma'' = \sigma[1/x]} & \langle w, \sigma[1/x] \rangle \to \sigma' \end{array}$$

Let us continue the derivation for $\langle w, \sigma[1/x] \rangle \to \sigma'$:

$$\begin{array}{rl} \langle w, \sigma[1/x] \rangle \to \sigma' \nwarrow & \langle x > 0, \sigma[1/x] \rangle \to \textbf{true},\ \langle x := 1, \sigma[1/x] \rangle \to \sigma''',\ \langle w, \sigma''' \rangle \to \sigma' \\ \nwarrow^* & \langle x := 1, \sigma[1/x] \rangle \to \sigma''',\ \langle w, \sigma''' \rangle \to \sigma' \\ \nwarrow_{\sigma''' = \sigma[1/x]} & \langle w, \sigma[1/x] \rangle \to \sigma' \end{array}$$

Now, note that we got the same subgoal $\langle w, \sigma[1/x] \rangle \to \sigma'$ already inspected: hence it is not possible to conclude the derivation, which will loop.
Summing up all the above we conclude that:

$$\forall \sigma, \sigma'.\quad \langle \textbf{while } x > 0 \textbf{ do } x := 1, \sigma \rangle \to \sigma' \quad \Rightarrow \quad \sigma(x) \leq 0 \wedge \sigma' = \sigma$$

We can now complete the reduction for the whole c_1 when $\sigma(x) \leq 0$ (the case $\sigma(x) > 0$ is discharged, because we know that there is no derivation).

DRAFT

$$
\begin{array}{rl}
\langle w; x := 0, \sigma\rangle \to \sigma' \nwarrow & \langle w, \sigma\rangle \to \sigma'',\ \langle x := 0, \sigma''\rangle \to \sigma' \\
\nwarrow^{*}_{\sigma''=\sigma} & \langle x := 0, \sigma\rangle \to \sigma' \\
\nwarrow^{*}_{\sigma'=\sigma[0/x]} & \Box
\end{array}
$$

Therefore the evaluation ends with $\sigma' = \sigma[0/x]$.
By comparing c_1 and c_2 we have that:

- there are memories for which the two commands behave the same (i.e., when $\sigma(x) \leq 0$)

$$\exists \sigma, \sigma'. \begin{cases} \langle (\textbf{while}\, x > 0\, \textbf{do}\, x := 1); x := 0, \sigma\rangle \to \sigma' \\ \langle x := 0, \sigma\rangle \to \sigma' \end{cases}$$

- there are also cases for which the two commands exhibit different behaviours:

$$\exists \sigma, \sigma'. \begin{cases} \langle (\textbf{while}\, x > 0\, \textbf{do}\, x := 1); x := 0, \sigma\rangle \not\to \\ \langle x := 0, \sigma\rangle \to \sigma' \end{cases}$$

As an example, take any σ with $\sigma(x) = 1$ and $\sigma' = \sigma[0/x]$.

Since we can find pairs (σ, σ') such that c_1 loops and c_2 terminates we have that $c_1 \not\sim c_2$.

Note that in disproving the equivalence we have exploited a standard technique in logic: to show that a universally quantified formula is not valid we can exhibit one counterexample. Formally:

$$\neg \forall x. (P(x) \Leftrightarrow Q(x)) = \exists x. (P(x) \wedge \neg Q(x)) \vee (\neg P(x) \wedge Q(x))$$

3.3.4 Examples: Diverging Computations

What does it happen if the program has infinite different looping situations? How should we handle the memories for which this happens?

Let us rephrase the definition of equivalence between commands:

$$\forall \sigma, \sigma' \quad \begin{cases} \langle c_1, \sigma\rangle \to \sigma' \Leftrightarrow \langle c_2, \sigma\rangle \to \sigma' \\ \langle c_1, \sigma\rangle \not\to \quad \Leftrightarrow \langle c_2, \sigma\rangle \not\to \end{cases}$$

Next we see an example where this situation emerges.

Example 3.9 (Proofs of non-termination). Let us consider the commands:

$$
\begin{array}{l}
c_1 \stackrel{\text{def}}{=} \textbf{while}\ x > 0\ \textbf{do}\ x := 1 \\
c_2 \stackrel{\text{def}}{=} \textbf{while}\ x > 0\ \textbf{do}\ x := x + 1
\end{array}
$$

DRAFT

Is it true that $c_1 \sim c_2$? On the one hand, note that c_1 can only store 1 in x, whereas c_2 can keep incrementing the value stored in x, so one may be lead to suspect that the two commands are not equivalent. On the other hand, we know that when the commands diverge, the values stored in the memory locations are inessential.

As already done in previous examples, let us focus on the possible derivation of c_1 by considering two separate cases that depends of the evaluation of the guard $x > 0$:

Case $\sigma(x) \leq 0$) If $\sigma(x) \leq 0$, we know already from Example 3.8 that $\langle c_1, \sigma \rangle \to \sigma$:

$$\begin{array}{lll} \langle c_1, \sigma \rangle \to \sigma' & \nwarrow_{\sigma' = \sigma} & \langle x > 0, \sigma \rangle \to \textbf{false} \\ & \nwarrow^{*} & \square \end{array}$$

In this case, the body of the **while** is not executed and the resulting memory is left unchanged. We leave to the reader to fill the details for the analogous derivation of c_2, which behaves the same.

Case $\sigma(x) > 0$) If $\sigma(x) > 0$, we know already from Example 3.8 that $\langle c_1, \sigma \rangle \not\to$ Now we must check if c_2 diverges too when $\sigma(x) > 0$:

$$\begin{array}{lll} \langle c_2, \sigma \rangle \to \sigma' & \nwarrow & \langle x > 0, \sigma \rangle \to \textbf{true}, \\ & & \langle x := x+1, \sigma \rangle \to \sigma_1, \quad \langle c_2, \sigma_1 \rangle \to \sigma' \\ & \nwarrow^{*} & \langle x := x+1, \sigma \rangle \to \sigma_1, \quad \langle c_2, \sigma_1 \rangle \to \sigma' \\ & \nwarrow^{*}_{\sigma_1 = \sigma\left[^{\sigma(x)+1}/_x\right]} & \langle c_2, \sigma \left[^{\sigma(x)+1}/_x\right] \rangle \to \sigma' \\ & \nwarrow & \langle x > 0, \sigma \left[^{\sigma(x)+1}/_x\right] \rangle \to \textbf{true}, \\ & & \langle x := x+1, \sigma \left[^{\sigma(x)+1}/_x\right] \rangle \to \sigma_2, \\ & & \langle c_2, \sigma_2 \rangle \to \sigma' \\ & \nwarrow^{*} & \langle x := x+1, \sigma \left[^{\sigma(x)+1}/_x\right] \rangle \to \sigma_2, \\ & & \langle c_2, \sigma_2 \rangle \to \sigma' \\ & \nwarrow^{*}_{\sigma_2 = \sigma_1\left[^{\sigma_1(x)+1}/_x\right] = \sigma\left[^{\sigma(x)+2}/_x\right]} & \langle c_2, \sigma \left[^{\sigma(x)+2}/_x\right] \rangle \to \sigma' \\ & & \dots \end{array}$$

Now the situation is more subtle: we keep looping, but without crossing the same subgoal twice, because the memory is updated with different values for x at each iteration. However, using induction, that will be the subject of Section 4.1.3, we can prove that the derivation will not terminate. Roughly, the idea is the following:

- at step 0, i.e., at the first iteration, the cycle does not terminate;
- if at the ith step the cycle has not terminated yet, then it will not terminate at the $(i+1)$th step, because $x > 0 \Rightarrow x+1 > 0$.

DRAFT

The formal proof would require to show that at the ith iteration the values stored in the memory at location x will be $\sigma(x)+i$, from which we can conclude that the expression $x>0$ will hold true (since by assumption $\sigma(x)>0$ and thus $\sigma(x)+i>0$). Once the proof is completed, we can conclude that c_2 diverges and therefore $c_1 \sim c_2$.

Let us consider the command $w \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$. As we have seen in the last example, to prove the non-termination of w we can exploit the induction hypotheses over memory states to define the inference rule below: the idea is that if we can find a set S of memories such that, for any $\sigma' \in S$, the guard b is evaluated to **true** and the execution of c leads to a memory σ'' which is also in S, then we can conclude that w diverges when evaluated in any of the memories $\sigma \in S$.

$$\frac{\sigma \in S \quad \forall \sigma' \in S. \langle b, \sigma' \rangle \to \textbf{true} \quad \forall \sigma' \in S, \forall \sigma''. (\langle c, \sigma' \rangle \to \sigma'' \Rightarrow \sigma'' \in S)}{\langle w, \sigma \rangle \not\to} \tag{3.20}$$

Note that the property

$$\forall \sigma''. (\langle c, \sigma' \rangle \to \sigma'' \Rightarrow \sigma'' \in S)$$

is satisfied even when $\langle c, \sigma' \rangle \not\to$, because there is no σ'' such that the left-hand side of the implication holds.

Remind that, in general, program termination is semi-decidable (and non-termination possibly non semi-decidable), so we cannot have a proof technique for demonstrating the convergence or divergence of any program.

Example 3.10 (Collatz's algorithm). Consider the algorithm below, which is known as *Collatz's algorithm*, or also as *Half Or Triple Plus One*

$$d \stackrel{\text{def}}{=} x := y \,;\, k := 0 \,;\, \textbf{while } x > 0 \textbf{ do } (x := x - 2 \,;\, k := k + 1)$$
$$c \stackrel{\text{def}}{=} \textbf{while } y \neq 1 \textbf{ do } (d \,;\, \textbf{if } x = 0 \textbf{ then } y := k \textbf{ else } y := (3 \times y) + 1)$$

The command d, when executed in a memory σ with $\sigma(y) > 0$, terminates by producing either a memory σ' with $\sigma'(x) = 0$ and $\sigma(y) = 2 \times \sigma'(k)$ (when $\sigma(y)$ is even), or a memory σ'' with $\sigma''(x) = -1$ (when $\sigma(y)$ is odd). The command c exploits d to update at each iteration the value of y to either the half of y (when $\sigma(y)$ is even) or three times y plus one (when $\sigma(y)$ is odd). It is an open mathematical conjecture to prove that the command c terminates when executed in any memory σ with $\sigma(y) > 0$. The conjecture has been checked by computers and proved true[1] for all starting values of y up to 5×2^{60}.

[1] Source `http://en.wikipedia.org/wiki/Collatz_conjecture`, last visited July 2015.

DRAFT

Problems

3.1. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y > 0 \textbf{ do } (r := r \times x \;;\; y := y - 1)$$

Let $c \stackrel{\text{def}}{=} (r := 1 \;;\; w)$ and $\sigma \stackrel{\text{def}}{=} [9/x, 2/y]$. Use goal-oriented derivation, according to the operational semantics of IMP, to find the memory σ' such that $\langle c, \sigma \rangle \to \sigma'$, if it exists.

3.2. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y > 0 \textbf{ do if } y = 0 \textbf{ then } y := y + 1 \textbf{ else skip}$$

For which memories σ, σ' do we have $\langle w, \sigma \rangle \to \sigma'$?

3.3. Prove that for any $b \in Bexp, c \in Com$ we have $c \sim \textbf{if } b \textbf{ then } c \textbf{ else } c$.

3.4. Prove that for any $b \in Bexp, c \in Com$ we have $c_1 \sim c_2$, where:

$$c_1 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do if } b \textbf{ then } c \textbf{ else skip}$$

3.5. Prove that for any $b \in Bexp, c \in Com$ we have $c_1 \sim c_2$, where:

$$c_1 \stackrel{\text{def}}{=} c \;;\; \textbf{while } b \textbf{ do } c$$
$$c_2 \stackrel{\text{def}}{=} (\textbf{while } b \textbf{ do } c) \;;\; c$$

3.6. Prove that $c_1 \not\sim c_2$, where:

$$c_1 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := 0$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while } x \geq 0 \textbf{ do } x := 0$$

3.7. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \leq y \textbf{ do } (x := x + 1 \;;\; y := y + 2)$$

Find the largest set S of memories such that the command w diverges. Use the inference rule for divergence to prove non-termination.

3.8. Prove that $c_1 \not\sim c_2$, where:

$$c_1 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } x := x + 1$$
$$c_2 \stackrel{\text{def}}{=} \textbf{while } x \geq 0 \textbf{ do } x := x + 2$$

DRAFT

3.9. Suppose we extend IMP with the arithmetic expression a_0/a_1 for integer division, whose operational semantics is:

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0/a_1, \sigma\rangle \to n}\ n_0 = n_1 \times n\ (\text{div}) \tag{3.21}$$

1. Prove that the semantics of extended arithmetic expressions is not deterministic. In other words, give a counterexample to the property below:

$$\forall a \in Aexp, \forall \sigma \in \Sigma, \forall n, m \in \mathbb{Z}.\ (\langle a, \sigma\rangle \to n \wedge \langle a, \sigma\rangle \to m \quad \Rightarrow \quad n = m)$$

2. Prove that the semantics of extended arithmetic expressions is not always defined. In other words, give a counterexample to the property below:

$$\forall a \in Aexp, \forall \sigma \in \Sigma, \exists n \in \mathbb{Z}.\ \langle a, \sigma\rangle \to n$$

3.10. Define a small-step operational semantics for IMP. To this aim, introduce a special symbol $\star$ as a termination marker and consider judgements of either the form $\langle c, \sigma\rangle \to \langle c', \sigma'\rangle$ or $\langle c, \sigma\rangle \to \langle \star, \sigma'\rangle$. Define the semantics in such a way that the evaluation is deterministic and that $\langle c, \sigma\rangle \to^* \langle \star, \sigma'\rangle$ if and only if $\langle c, \sigma\rangle \to \sigma'$ in the usual big-step semantics seen for IMP.

DRAFT