# ASSIGNMENT CALCULUS:
# A PURE IMPERATIVE REASONING LANGUAGE

# Assignment Calculus: A Pure Imperative Reasoning Language

By
MARC BENDER, B.ENG.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Department of Computing and Software
McMaster University

DOCTOR OF PHILOSOPHY (2010)
(Computer Science)

McMaster University
Hamilton, Ontario

TITLE: Assignment Calculus: A Pure Imperative Reasoning Language

AUTHOR: Marc Bender, B.Eng. (McMaster University)

SUPERVISOR: Dr. Jeffery Zucker

NUMBER OF PAGES: v, 124

# Abstract

In this thesis, we undertake a study of *imperative reasoning*. Beginning with a philosophical analysis of the distinction between imperative and functional language features, we define a (pure) *imperative language* as one whose constructs are inherently *referentially opaque*. We then give a definition of a *reasoning language* by identifying desirable properties such a language should have.

The rest of the thesis presents a new pure imperative reasoning language, *Assignment Calculus* ***AC***. The main idea behind ***AC*** is that R. Montague's modal operators of *intension* and *extension* are useful tools for modeling *procedures* in programming languages. This line of thought builds on T. Janssen's demonstration that Montague's intensional logic is well suited to dealing with assignment statements, pointers, and other difficult features of imperative languages.

***AC*** consists of only four basic constructs, *assignment* '$\boldsymbol{X}$ **:=** **t**', *sequence* '**t ; u**', *procedure formation* '**ỉt**' and *procedure invocation* '**!t**'. Three interpretations are given for ***AC***: an operational semantics, a denotational semantics, and a term-rewriting system. The three are shown to be equivalent. Running examples are used to illustrate each of the interpretations.

Five variants of ***AC*** are then studied. By removing restrictions from ***AC***'s syntactic and denotational definitions, we can incorporate *L-values*, *lazy evaluation*, *state backtracking*, and *procedure composition* into ***AC***. By incorporating procedure composition, we show that ***AC*** becomes a self-contained Turing complete language

in the same way as the untyped $\lambda$-calculus: by encoding numerals, Booleans, and control structures as $\boldsymbol{AC}$ terms. Finally we look at the combination of $\boldsymbol{AC}$ with a typed $\lambda$-calculus.

# Acknowledgements

I am forever indebted to my supervisor and mentor, Dr. Jeff Zucker. Had I not attended his course on computability theory, in the last semester of my undergraduate program and entirely by chance, I would likely not have considered graduate school at all; without his support and encouragement, this thesis would have neither been undertaken nor been completed.

I would also like to thank the members of my supervisory committee. In particular, my numerous and lengthy conversations with Dr. William Farmer have always been enlightening and helpful, and in particular have made the mathematical foundations of Chapter 3 possible. From discussions with Dr. Jacques Carette I developed a much deeper understanding of the concept of intension, among other things. More importantly, without his insightful observations, I might never have arrived at my proof of Lemma 3.36.

To my external examiner, Dr. Jan Willem Klop, I am tremendously grateful for the very positive review of my thesis, and for the helpful indications of future work. Thanks also to Dr. Wolfram Kahl for help with referential transparency and rewriting issues, among other things, and to Dr. Tom Maibaum for his encouragement, particularly in letting me present my work in his departmental seminar series.

For their support, encouragement, patience and love, I thank my wife Rebecca, our daughter Robyn, our family and friends.

# Contents

**Abstract** iii

**Acknowledgements** v

**Introduction** 1

0.1 Assignment Calculus . . . . . 1

0.2 Structure of the Thesis . . . . . 3

0.3 Notation . . . . . 3

**1 Imperative Reasoning** 5

1.1 What is Imperative Reasoning? . . . . . 7

1.2 Referential Opacity . . . . . 10

1.3 Intension . . . . . 13

1.4 Intentions . . . . . 16

**2 Assignment Calculus** 18

2.1 Introduction to ***AC*** . . . . . 20

2.2 Types and Syntax of ***AC*** . . . . . 22

2.3 Operational Semantics of ***AC*** . . . . . 25

2.4 Rigidity, Modal Closedness, and Canonicity . . . . . 28

2.5 State Backtracking . . . . . 33

**3 Semantics of *AC*** 36

3.1 Domains . . . . . 37

3.1.1 Reflexive Domains . . . . . 40

3.2 Semantics . . . . . 43
3.2.1 States and semantic domains . . . . . 43
3.2.2 Denotational Semantics . . . . . 45
3.3 Equivalence of Interpretations . . . . . 49

**4 Term Rewriting** **63**
4.1 Rewrite rules and properties . . . . . 64
4.2 Equivalence of Interpretations . . . . . 71

**5 Extensions and Variants of *AC*** **77**
5.1 L-values . . . . . 78
5.2 Lazy Evaluation . . . . . 81
5.2.1 Lazy assignment . . . . . 82
5.2.2 Lazy sequence . . . . . 84
5.3 Labelled State Backtracking . . . . . 85
5.4 ***AC*** with Procedure Composition . . . . . 86
5.5 Combining ***AC*** with Typed $\lambda$-Calculus . . . . . 90

**6 Conclusion** **96**
6.1 Goals and Contributions . . . . . 96
6.2 Related Work . . . . . 98
6.3 Future Work . . . . . 100

**A Implementation** **102**
A.1 Introduction and Usage Instructions . . . . . 102
A.2 Parsing and Printing . . . . . 104
A.3 Properties and Manipulation . . . . . 106
A.4 Interaction . . . . . 113
A.5 Examples . . . . . 114

# Introduction

What is a pure imperative language? A careful attempt to answer this question leads to many interesting questions about programming languages. This dissertation presents and pursues one possible definition: a pure imperative language is one whose operators are fundamentally *referentially opaque*; in simple terms, they make substitution problematic.

## 0.1 Assignment Calculus

We begin the thesis with a primarily philosophical, but example-driven, discussion of the above definition. We also give a definition of a *reasoning language*, which is somewhat more subjective but can qualitatively be defined by identifying several desirable properties such a language should have; the $\lambda$-calculus is taken as the paradigm.

Referential opacity, the principle of substitutivity, and intensionality are then discussed. Starting with some natural-language examples, we motivate the use of intension for handling certain types of problems, and then demonstrate that these problems are also present in imperative programming languages. After a presentation of its historical and technical background, the main subject of this dissertation is introduced: our imperative reasoning language Assignment Calculus, $\boldsymbol{AC}$.

The formal syntax and operational semantics of $\boldsymbol{AC}$ is given. We define and derive some of its important properties. The running examples of the thesis are

presented, with their intuitive meanings and also their operational interpretations. These three examples will be used throughout the thesis to motivate and compare each of the interpretations of ***AC***. We also examine an interesting component of ***AC***: state backtracking. This is an unusual feature of ***AC*** that we argue is a highly useful part of imperative reasoning.

After going over some mathematical preliminaries of domain theory, the semantic domains in which ***AC*** will be interpreted are defined. Of note here is the interpretation of the domain of *possible worlds* or *states* as a *reflexive domain*; this allows for the storage of intensions (procedures) in the state. We then proceed to give the compositional denotational semantics of ***AC*** terms, and identify some properties and results. The semantic definitions are demonstrated by applying them to the running examples. A proof of the equivalence of the operational and denotational semantics of ***AC*** follows. This requires some new tools to be developed, most notably *bounded recursion semantics* that allows us roughly to limit (denotationally) the number of procedure calls in the evaluation of an ***AC*** term.

A term rewriting system for ***AC*** is presented next; after giving its rules and proving some properties we use it to interpret the running examples. We then prove the equivalence of the rewriting scheme to the operational and denotational semantics.

Taking ***AC*** as a starting point, we explore various extensions and variants of the core language. We look at alternate evaluation schemes for ***AC***, introducing L-values and pointers, and adding functional features. The core of ***AC*** is slightly enriched and is shown to be a self-contained Turing complete language in which we can encode, for example, numerals and Booleans.

Finally, we summarize the contributions of the thesis, talk about related work, and indicate some possible lines of further research. This is followed by a presentation of the implementation of ***AC***, written in the Haskell programming language. We give instructions for obtaining and using the program, some examples of its use, and a partial listing of the program code. The program is available from the author's

website at www.cas.mcmaster.ca/~bendermm. Using this program, readers of this dissertation can work through any ***AC*** examples themselves and interactively explore the possibilities of reasoning with ***AC***.

## 0.2 Structure of the Thesis

The thesis is structured according to the following plan.

- Chapter 1 presents the background and philosophical motivation;
- Chapter 2 is devoted to the syntax and operational semantics of ***AC***;
- Chapter 3 gives a denotational semantics for ***AC***, as well as a proof of the equivalence of the operational and denotational semantics;
- Chapter 4 provides a rewriting system for ***AC***, and contains a proof that the rewriting rules are sound and complete; it also contains an important conjecture on confluence.[1]
- Chapter 5 presents five extensions and variants of ***AC***, as well as expansions of the earlier proofs to two of these;
- Chapter 6 discusses the contributions of the thesis, as well as related and future work;
- Finally, the Appendix presents the implementation of ***AC***.

## 0.3 Notation

Some comments on notation will help to organize our presentation. Syntax is represented by bold text, whereas semantics (the metalanguage) is normal (non-bold) mathematical text. Sans-serif symbols are metavariables—if they are bold, they range

[1]This conjecture has since been proved by the author.

over syntax; if not, over values. The equivalence of two syntactic entities, meaning that they consist of the same sequence of symbols (assuming full parenthesization to avoid ambiguity), is indicated by the symbol '$\equiv$'.

# Chapter 1

# Imperative Reasoning

To program a computer, we must provide it with unambiguous instructions to direct its actions, in order that it will arrive at the desired result. What qualifies as such an instruction? First, there are *commands*, which indicate exactly what the computer should do next. Commands (or statements) are usually presented sequentially, with conditions and jumps that allow us to branch or "jump" to other parts of the sequence.

Second, there are *goals*. These provide a "specification," and it is up to the computer (or compiler) to figure out how to arrive at a solution. Goals are generally written as *declarations* that specify the form that the solution must take, and are often presented in a "functional" style [Bac78].

This distinction between types of instructions forms the basis of the two types of programming language: a language that is based on commands is called *imperative*, and one that is based on goals is called *declarative*.

The distinction between these two approaches can be traced back all the way to two pioneers in computation theory, Alan Turing and Alonzo Church. Turing's approach [Tur36] to describing computation is via a machine (the *Turing machine*) that executes tasks sequentially, reading from and writing to a storage device (the "tape"). Church's system [Chu41] is somewhat more abstract, taking the mathematical notion

of *function* as basic, and employing only the operations of (functional) *abstraction* and *application* to express computational goals. Turing proves that their systems have the same computational power. He also argues convincingly that they are *universal* in that they can model any computable procedure (this is the *Church-Turing Thesis*). Although they are equivalent in power, Turing's and Church's conceptions have developed over time in quite different ways.

The great benefit of Turing's approach is its immediate suggestion of a real, workable computer; a variant, due to von Neumann, still lies at the core of virtually every computing device in use today [TN01, §4.1]. Imperative programming languages were essentially born out of practical need: to program a computer one must "speak its language"; since computers are basically built on Turing's idea, so are imperative languages. Thus in any imperative language one will find operations for sequencing (e.g., '**;**'), as well as for reading from and writing to memory (e.g., $\boldsymbol{X} := \mathbf{1}$). Imperative languages are thus closely connected to practice, but also to Turing's conception of computation. However, dealing with imperative languages' syntactic and semantic concepts can be tricky, and the Turing machine can be a clumsy theoretical tool for certain tasks.

Church's approach, the $\lambda$-calculus, is a syntactic system that stems from research into the foundations of mathematics. Its language of functional abstraction and application over variables is astonishingly small, elegant, and powerful. In addition, its use of variables and binding is very similar to traditional mathematical and logical languages, making it immediately amenable to theoretical study. Since its invention, many programming languages have been designed around the $\lambda$-calculus, i.e., so-called *functional languages*. Furthermore, the mathematical properties of the $\lambda$-calculus and related systems are well-explored (cf. for example Barendregt [Bar84]). On the other hand, a major drawback to functional languages is their lack of a "machine intuition", which makes them somewhat more difficult to implement and, arguably, to use.

Real computers are based on the imperative model, so compilers for functional

languages are need to translate Church-style computation into Turing's model. Conversely for computer scientists working in denotational semantics, to give a mathematical meaning to imperative programming languages means interpreting Turing-style computation in a Church-style (functional) language.

Can we "short-circuit" this interpretation in either direction? In other words, can we (a) build a computer on Church's notion, or (b) design a formal language that embodies Turing's conception? To question (a) we devote little discussion, but simply ask the interested reader to imagine how the concept of first-class functions might be implemented as the basis of a working computer. Certainly the way to a solution is not as clear as it is in the case of Turing machines. Architectures that are closer to the declarative paradigm have been explored by several researchers, but the results have generally been less than successful [HHJW07, §2].

As to question (b), it is somewhat surprising that there is no accepted canonical *theoretical reasoning language* that is fundamentally imperative in character. Considering that in the real world imperative programming is ubiquitous, why is it the case that we have no theoretical "kernel" for it?

The goal of this dissertation is to present a language that can serve just such a purpose. It is not presented as "the" basic language for imperative reasoning, but simply offered as a potential candidate. First, however, we must answer a pressing question: what, exactly, is "imperative reasoning"?

## 1.1 What is Imperative Reasoning?

A first attempt to define imperative reasoning could be made from the point of view of *machine behaviour*. If functional features are seen as "high-level," then the argument could be made that a pure imperative language should be as close to the machine as possible. There is certainly plenty of value in this view, and we intend to try to remain as close to such a machine-based intuition as we can. The problem is that

this perspective does not sufficiently *restrict* our definition: there are many different machine architectures and instruction sets, many different abstractions provided between machine and assembly languages, many different ways of implementing control structures; in short, there is simply too much freedom when working with machine intuition alone. Therefore we add this condition: we want a language that we can reason *within*, not just *about*. We want a language with a simple and intuitive rewriting system. This design goal is similar to that mentioned by Felleisen [FH92, §1]. From this perspective, Hoare-style logics [Hoa69, dB80, HKT00, Rey02] are not satisfactory; it is necessary, when working with such logics, to "step out" of the (programming) language of interest to reason about it.

A better point of view is afforded if we take "pure imperative" to mean "not functional" in much the same way as "pure functional" means "not imperative". The roots of this view are put forth in Backus' Turing Award lecture [Bac78]; its history is organized and presented in [HHJW07]. By adopting this perspective, we can identify much more clearly those fundamental parts of programming languages that are *truly* imperative. So we begin our investigations with the concept of *functional purity*: what is it exactly? This will lead us to a better understanding of what it is *not*.

Purely functional languages are *referentially transparent*. This maxim is repeated often in the literature [Rea89, p.10], [FH88, p.10], [BW88, p.4] and is seen as a key feature of pure functional languages because it allows for *call-by-name* or *lazy* evaluation. A notable example of a lazy (pure) functional programming language is Haskell [P+03, HHJW07].

Referential transparency (from now on just *transparency*) is a concept that goes back to Quine [Qui60, §30] and essentially embodies Leibniz's *principle of substitutivity of equals*:

**Definition 1.1** (Substitutivity). In any expression, substituting one subexpression for another with the same meaning preserves the overall meaning of the expression.

More precisely,

$$e_1 = e_2 \implies e(\cdots e_1 \cdots) = e(\cdots e_2 \cdots).$$

(Syntactic representations of pure functional languages (often) have a single exception to this principle: *variable capture*. The exception is unimportant because various ways to sidestep the issue are well documented [Bar84].)

This principle is important because it is true of all traditional mathematical languages. The main benefit of transparency is clear: with the benefit of free substitution, "computation" can proceed by substituting expressions for placeholders or *variables*. This idea is taken to a dazzling extreme in the $\lambda$-calculus.

The pure $\lambda$-calculus reduces functional reasoning to its smallest possible core.[1] Its (only!) operators are $\lambda$-*abstraction*, which represents function formation, and *application*, which represents the usual function application familiar to mathematicians; its only atoms are *variables*. Syntactically, application of a function to an argument is accomplished by substitution (modulo variable capture), taking full advantage of referential transparency. With just these, the full machinery of computation can be constructed. We take the $\lambda$-calculus as the paragon of theoretical reasoning languages:

- It is *small*, *elegant* and *intuitive*;
- Its operators (faithfully) represent well-understood, fundamental concepts;
- It is *well grounded*, having operational and denotational semantics that are *equivalent*;
- We can *rewrite* its terms using simple (provably valid) rules;
- It has interesting properties;
- It is easy to modify and extend [Bar84].

[1] Further reductions can be made, but in the author's opinion, only at the cost of intuitiveness and perspicuity.

Our aim is to develop a formal language for *imperative* reasoning that has as many of the above virtues as possible.

This brings us back to the question at hand: what is a pure imperative language? We can now take a definite position: it is a language whose features are fundamentally non-transparent, or *opaque*. That is, we are interested only in those operators for which substitutivity is the exception rather than the rule.

## 1.2 Referential Opacity

Rather than diving directly into programming language examples, we begin with an informal look at opacity in *natural* language. Consider the following sentence:

**The temperature is twenty degrees and rising.**

At first glance, the statement seems innocuous enough. Let us try to formalize it somewhat; introduce the variable $\boldsymbol{temp}$ for temperature, and predicates $\boldsymbol{twenty}(\boldsymbol{x})$ meaning that $\boldsymbol{x}$ is twenty degrees and $\boldsymbol{rising}(\boldsymbol{y})$ meaning that $\boldsymbol{y}$ is rising. Then we can write our sentence as

$$\boldsymbol{twenty}(\boldsymbol{temp}) \wedge \boldsymbol{rising}(\boldsymbol{temp}) \tag{1}$$

Again, this seems to make sense. Now, let us say that the current temperature is in fact $\mathbf{20°}$. We should be able to substitute this value for $\boldsymbol{temp}$ in (1) and determine its truth value; doing so results in

$$\boldsymbol{twenty}(\mathbf{20°}) \wedge \boldsymbol{rising}(\mathbf{20°})$$

The first part, $\boldsymbol{twenty}(\mathbf{20°})$, is of course true. But the second part makes no sense: $\mathbf{20°}$ is always just twenty degrees cannot "rise". When we say that the temperature is rising, we are not only talking about the current value of the temperature, but about its value over time (specifically its derivative). The predicate $\boldsymbol{rising}$ introduces

what is known as an *opaque context*: it does not suffice to look at the current (or extensional) value of its argument. Instead we must consider its "larger" meaning, called its *sense* or *intension*. ***twenty*** on the other hand creates a *transparent context*.

Such intensional phenomena abound in natural language, and have created interesting problems for philosophers and linguists for some time [Fre92], [Tho74]. Generally they can be recognized by apparent violations of substitutivity like in the example above. This is also the case for imperative programming languages, to which we now turn our attention.

Consider the expression:

$$\boldsymbol{X} \geq \mathbf{1}. \tag{2}$$

In itself, the above causes no problems; it is transparent in that, if we know for example that $\boldsymbol{X}$ is $\mathbf{2}$ then we can substitute to obtain

$$\mathbf{2} \geq \mathbf{1} \Rrightarrow \mathbf{true}$$

as we would expect. In an imperative language, we could perhaps write this as

$$\boldsymbol{X} := \mathbf{2};\ (\boldsymbol{X} \geq \mathbf{1}) \Rrightarrow \mathbf{true}. \tag{3}$$

Another way to state this fact is to say that $\boldsymbol{X} \geq \mathbf{1}$ in the "current machine state". The trouble starts when we start to think not only in the context of the current machine state or even other states, but rather about *varying or changing states in a computation*, for example,

$$\textbf{During loop } \mathcal{L},\ \boldsymbol{X} \geq \mathbf{1}. \tag{4}$$

The above states a *loop invariant* and only makes sense if we think of $\boldsymbol{X}$ as representing a (possibly) changing value, that is, a value that is subject to destructive update multiple times during the execution of $\mathcal{L}$. Here we see the first case of an apparent violation of substitutivity: substituting the current value of $\boldsymbol{X}$ into (4) gives

$$\textbf{During loop } \mathcal{L},\ \mathbf{2} \geq \mathbf{1}.$$

which is trivially true and clearly has lost the meaning of (4). The sentence "**During** $\mathcal{L}$, $\Box$" has introduced an opaque (or *intensional*) context.

We do not need to resort to natural language to create opaque contexts. They are inherent to *all* of the fundamental imperative operations. First, consider the sequence operator ';'. It introduces an opaque context on its right-hand side, as demonstrated implicitly by (3). Whereas (2) is true only in states wherein $\boldsymbol{X}$ has a value greater than or equal to **1**, (3) is *always* true. Specifically, if the current state assigns **0** to $\boldsymbol{X}$, then we cannot substitute **0** for the second occurrence of $\boldsymbol{X}$ in (3) to obtain

$$\boldsymbol{X} := \mathbf{2};\ (\mathbf{0} \geq \mathbf{1}) \Rrightarrow \mathbf{false}.$$

Next, examine the assignment statement itself; consider

$$\boldsymbol{X} := \boldsymbol{Y}. \tag{5}$$

It is transparent on its right-hand side; if the current state sets $\boldsymbol{Y}$ to **2** then (5) has the same meaning as

$$\boldsymbol{X} := \mathbf{2}.$$

But, as first discovered and studied by Janssen [JvEB77], the assignment statement is opaque on its left-hand side. If, for example, the current state sets $\boldsymbol{X}$ to **1**, we cannot substitute that into (5) to obtain

$$\mathbf{1} := \boldsymbol{Y},$$

because not only does the above not have the same meaning as (5), it has *no meaning at all!* This opaque context is quite interesting because it does not just indicate the value of a memory location "in all contexts" as in the earlier examples, but instead indicates the memory location "itself", called its *L-value* [Str73]. This penetrating insight of Janssen's is the starting point for the line of investigation ([Jan86, Hun90, HZ91]) that this thesis continues.

Another example of referential opacity is the while-loop construct, which has the form

$$\textbf{while t do b} \tag{6}$$

where **t** is the loop test and **b** is the loop body. It turns out that both **t** and **b** are intensional contexts, as they must be reevaluated at each iteration in a new machine state. For example, if **t** is $\boldsymbol{X = 1}$, then substituting **0** for $\boldsymbol{X}$ in (6) will result in a non-terminating program because the loop test will never be rechecked with a new value for $\boldsymbol{X}$. Note the similarity of this example to the loop invariant example.

## 1.3 Intension

Frege, in [Fre92], carefully analyzed the substitutivity problem. This led him to distinguish between two kinds of meaning: *sense* (*Sinn*), the "larger" or "global" meaning, and *reference* or *denotation* (*Bedeutung*), the "smaller", "local" or "current" meaning. He shows that, in cases where substitutivity does not hold in terms of the *denotations* of expressions, it can be *restored* if we consider the *senses* of those expressions.

Going back to example (1), we have seen that in

$$\boldsymbol{rising(temp)}$$

we cannot substitute an expression for ***temp*** that is simply co-denotational with ***temp***. We *can*, however, substitute an expression that has the same *sense* as temperature such as, say, ***therm*** = **"the value displayed by the thermometer"**. We say that they have the same sense because they are co-denotational in all possible "contexts,", "states," "settings" or—in Kripke's terminology [Kri59]—in all *possible worlds.*

Frege's work was primarily philosophical. He intended to distinguish and examine the two forms of meaning, but he did not go so far as to attempt to develop

a formal system to "implement" his ideas. This development had to wait until the work of Church [Chu51], who gave the first *syntactic* system that incorporates sense and denotation, and Carnap [Car47] who provided the first *semantic* interpretation of sense and denotation. Carnap introduced the names "intension" and "extension" respectively for his particular versions of these; he defined the intension of an expression as a *function from contexts of use (states) to values*, and the extension of an expression in some state as its intension function applied to that state. This was a valuable step in designing a workable conception of sense and denotation.

Kripke [Kri59], by providing the setting of *possible worlds* for modal logic, rounds out the semantic treatment. By combining his work with that of Carnap, we can treat intension as a function from possible worlds to values.

The next major step was accomplished by Montague [Mon70, Mon73], whose attempts to treat natural language by mathematical means led him to his *intensional logic* ***IL***, which takes the semantic work of Carnap and Kripke in the syntactic direction of Church. In ***IL***, not only can intension and extension be handled, but they are actually *incorporated as operators in the logic.* Montague's intension operator, '$\hat{}$', is a "higher-order" analogue of the necessity operator '$\Box$' of modal logic in the *same way* as Church's abstraction operator '$\boldsymbol{\lambda}$' is a higher-order analogue of the universal quantifier '$\forall$': following Gallin [Gal75], we can define

$$\Box\mathbf{t} \;=\; (\hat{}\mathbf{t} = \hat{}\mathbf{true})$$

just as Henkin [Hen63] defines

$$\forall \boldsymbol{x} \cdot \mathbf{t} \;=\; ((\boldsymbol{\lambda x} \cdot \mathbf{t}) = (\boldsymbol{\lambda x} \cdot \mathbf{true})).$$

Gallin [Gal75] provides a thorough treatment of ***IL*** along with some related systems. Montague utilized his logic to achieve great gains in the semantic study of natural language. But this is not what we are interested in.

Janssen and van Emde Boas, in [JvEB77], discovered that Montague's tools and techniques were useful for modeling certain aspects of imperative programming

languages. By identifying possible worlds with *machine states*, and slightly extending ***IL***, they provide strikingly elegant treatment of assignment statements and pointers. Due to the fact that ***IL***, including the variant they propose which others [GS90] have called Dynamic Intensional Logic or ***DIL***, has a compositional denotational semantics, the treatment they give is in the spirit of denotational semantics [Sto77].

This line of work was continued in Janssen's Ph.D thesis [Jan86], where a more complete fragment of a programming language was handled, including some tricky parts of Algol 68. A significant extension was accomplished by Hung and Zucker [Hun90, HZ91], who provide compositional denotational semantics of quite difficult language features such as

- Blocks with local identifiers:
  - "new-blocks" which create a new local program variable
  - "alias-blocks" which explicitly create aliasing between a local variable and an external variable
  - "macro-blocks" which model textual substitution for the local variable
- Procedure parameters, passed by value, by reference and by name, *treated together in a single system.*
- Pointers which can be dereferenced anywhere including the left-hand side of assignment statements

Janssen's system—specifically, Hung's version—is the genesis of Assignment Calculus ***AC***, the language to be presented in the remainder of this dissertation.

## 1.4 Intentions

The present work begins with the author's observation, while studying the work of Janssen and Hung, that

*Montague's intension operator generalizes the concept of (parameterless) procedure.*

A parameterless procedure is a function from states to states [Sto77, dB80]. An intension is a function from states to values. If we include states in the set of values, then the generalization is clear. The reason that it was not noticed by Janssen or Hung is that, in Montague's (and related) systems, *states cannot be treated as values.*

The system of dynamic intensional logic (***DIL***) of Janssen [Jan86], is an extension of Montague's original intensional logic ***IL***. Montague's system was intended for natural language semantics, and does not have operators that are suited to handling assignment and sequence.

In order to adapt ***IL*** to the setting, Janssen supplements its usual modal operators (intension and extension) with features that allow reading from and writing to the state. Reading from the state makes use of "access points" into the state called, as usual, *locations.* Updating the state makes use of a new invention, which Janssen calls the *state switcher.* The state switcher is a ternary operator '$\langle \mathbf{t}/\mathbf{u} \rangle \mathbf{v}$' that means, roughly, "the value of $\mathbf{v}$ in the current state modified so that $\mathbf{t}$ now contains $\mathbf{u}$". Therefore state switchers are actually a combination of *assignment* and *sequence.* For example, here is a rough translation of a small imperative program into its ***DIL*** equivalent:

$$\begin{array}{l} \boldsymbol{X} := \mathbf{1}; \\ \boldsymbol{Y} := \boldsymbol{X} \\ \textbf{return } \boldsymbol{Y} \end{array} \qquad \longmapsto \qquad \langle \boldsymbol{X}/\mathbf{1} \rangle \big( \langle \boldsymbol{Y}/\boldsymbol{X} \rangle (\boldsymbol{Y}) \big)$$

In ***AC*** we decouple the state switcher from its argument to regain the assignment and sequence operators themselves. This also has the consequence that states can be treated as values, which validates the observation mentioned above.

The second observation is that we can allow "storage" of intensions in the state. Stored procedures are a well-known but difficult feature of imperative languages [Sco70, §1], [SS71, §5]. They lead to general recursion, but also to a generalization of the treatment of pointers given by Janssen and Hung.

In fact, the system that results from this is sufficiently interesting that we have decided to study it in "isolation": we have removed all of the functional and logical components that were present in ***DIL***, such as variables, abstraction and application. The remainder of this thesis is devoted to defining, analyzing, working with, extending, and implementing the resulting language, ***AC***, as a *case study in pure imperative reasoning.*

# Chapter 2

# Assignment Calculus

In our attempt to create a pure imperative reasoning language, it is important to remain as close as possible to existing imperative programming languages. The selection of operators for $\boldsymbol{AC}$ must be as conservative as possible. So, what are the basic features of imperative languages?

An imperative programming language can be defined as a language $\boldsymbol{L}$ with the following characteristics:

1. The interpretation of $\boldsymbol{L}$ depends on some form of computer memory (state) through which data can be stored and retrieved.

2. The state consists of contents of discrete memory locations that can be read from or written to independently.

3. An $\boldsymbol{L}$-program consists of a *sequence* of explicit *commands* or instructions that fundamentally depend on (and often change) the state.

4. $\boldsymbol{L}$ contains some form of looping, branching or recursion mechanism to allow for repeated execution of program parts.[1]

[1]The fourth characteristic is, in a sense, optional; however, imperative languages that lack a repetition construct are less interesting because they are not Turing complete.

The first three characteristics are common to all imperative languages, whether these languages are theoretical or practical. Therefore, ***AC*** includes them directly.

The fourth characteristic, on the other hand, can be embodied in many different ways. We have conditional (location-based) branching or "jumping" in assembly languages, or "goto" statements in higher-level languages. There are a multitude of condition-based looping constructs, like "while" loops, "repeat-until" loops, "for" loops, etc. Finally, there is recursion, which depends on the notion of *procedure*: a named block of program text that can be invoked as many times as wanted, and can even call itself.

Practical imperative languages usually include more than one (if not all) of the above incarnations of characteristic 4. Generally speaking, there is no agreed-upon mechanism that is considered the most "basic". Jumping is the closest approximation to the way that actual computer hardware functions, but it is usually eschewed in high-level languages [Dij68] and its mathematical treatment is difficult [SW74, BT83]. Looping constructs are commonly used in small theoretical languages, but the choice of which one to use is arbitrary; besides, translating branching or recursion into looping is non-trivial whereas the inverse translation is quite simple. In terms of expressiveness, then, recursion seems best. However procedures tend to be a somewhat "bulky" addition to a small imperative language. This is due to the fact that the introduction of procedures usually requires, at the very least, a new set of identifiers for procedures and a procedure declaration construct.

***AC*** takes a different approach to characteristic 4. It employs Montague's intension operator as a *generalization of parameterless procedure.* Although Janssen and Hung had already employed intension and extension in ingenious and insightful ways, their ideas can be taken much further: we believe that

*Intension* is to *procedure formation* as $\lambda$-abstraction is to *function formation.*

In ***AC***, intensions are treated as first-class values: they can be defined anonymously,

assigned to locations, and invoked freely.

The goals of ***AC*** are twofold: firstly to continue the line of research initiated by Janssen and continued by Hung and Zucker in applying Montague's work to programming languages, and secondly to attempt to provide a small, elegant, useful *core language* for imperative-style reasoning—a *pure imperative reasoning language* as defined in Chapter 1.

## 2.1 Introduction to *AC*

The set of terms of ***AC*** is called ***Term***, and it is ranged over by the metavariables **t**, **u**, .... Before defining ***Term*** formally, which we do in Definition 2.5, we start by going over the basic constructs of ***AC*** and their intuitive meanings.

1. *Locations*: $\boldsymbol{X}$, $\boldsymbol{Y}$, ... These correspond to *memory locations* in a computer. '$\boldsymbol{X}$' alone is interpreted as "the contents of location $\boldsymbol{X}$". The collection of all locations constitutes the *store*.

2. *Assignment*: $\boldsymbol{X} := \mathbf{t}$. Overwrites the contents of location $\boldsymbol{X}$ with whatever **t** computes. The operation of assignment computes the *store* that results from such an update.

3. *Sequence*: **t ; u**. This statement is to be interpreted as follows. First compute **t**, which must return a store, then compute **u** in this new context. **t ; u ; v** is read **t ; (u ; v)**

4. *Intension*: **¡t**. This is the operation of *procedure formation*. It "holds" evaluation of **t** so that **¡t** is the procedure that, when invoked, returns whatever **t** computes.

5. *Extension*: **!t**. This operation is *procedure invocation*, that is, it "runs" the procedure computed by **t**.

We have decided to employ different notation than Montague for intension and extension because, although our operations are in a sense identical, the way in which we *use* them is significantly different. It is also the author's humble opinion that Montague's symbols are somewhat confusing, and therefore we have taken pains to find symbols whose meanings are clearer. An easy way to remember them is that '$\mathbf{i}$' looks like a lowercase 'i', for intension, and that '$\mathbf{!}$', an *ex*clamation mark, is used for extension.

The above features, considered in isolation, constitute *pure* $\boldsymbol{AC}$. In order to facilitate the development of examples, they are supplemented with the following:

1. Numerals: $\mathbf{0}, \mathbf{1}, \ldots$, ranged over by $\mathbf{m}, \mathbf{n}$,
2. Booleans: $\mathbf{true}, \mathbf{false}$, ranged over by $\mathbf{b}$,
3. Arithmetic operators: $\mathbf{t} + \mathbf{u}, \ldots$,
4. Boolean operators: $\mathbf{t} \wedge \mathbf{u}, \ldots$,
5. Number comparison operators: $\mathbf{t} < \mathbf{u}, \ldots$,
6. A conditional construct: $\mathbf{if\ t\ then\ u\ else\ v}$.

To reduce the use of parentheses, set the precedence of operators, in decreasing order, as follows: intension and extension, arithmetic and Boolean operators, assignment, sequence, conditional.

Here are some examples of $\boldsymbol{AC}$ terms:

**Examples 2.1.**

1. $\boldsymbol{X} := \mathbf{1};\ \boldsymbol{Y} := \boldsymbol{X};\ \boldsymbol{Y}$. This term sets $\boldsymbol{X}$ to $\mathbf{1}$, then sets $\boldsymbol{Y}$ to $\mathbf{1}$, and finally returns $\mathbf{1}$.
2. $\boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{X} := \mathbf{1};\ \mathbf{!}\boldsymbol{P}$. This term sets $\boldsymbol{P}$ to the procedure that returns $\boldsymbol{X}$, then sets $\boldsymbol{X}$ to $\mathbf{1}$. Finally, $\boldsymbol{P}$ is invoked thus returning the current value of $\boldsymbol{X}$ which is $\mathbf{1}$.

3. $P := \mathsf{i}(\textbf{if } X > 1 \textbf{ then } X \times (X := X - 1;\ !P) \textbf{ else } 1);\ !P$. This term computes the factorial of $X$ (which needs to have a value).

We will return to these examples often in the rest of the thesis, to demonstrate the various interpretations of $\boldsymbol{AC}$.

## 2.2 Types and Syntax of $AC$

$\boldsymbol{AC}$ is a simply typed language in the sense of Church [Chu40], meaning that all terms are typed *a priori*, and that there are no type variables.[2]

**Definition 2.2** (Types). The set of types $\boldsymbol{Type}$, ranged over by $\tau, \ldots$, is generated by:

$$\tau ::= \mathsf{B} \mid \mathsf{N} \mid \mathsf{S} \mid \mathsf{S} \to \tau,$$

where $\mathsf{B}$ is the type of Booleans, $\mathsf{N}$ is the type of natural numbers, $\mathsf{S}$ is the type of stores (or *states*, see Chapter 3), and $\mathsf{S} \to \tau$ is that of *intensions* (procedures) which return values of type $\tau$. Note that the base types $\mathsf{B}$ and $\mathsf{N}$ are added only to handle the supplementary features described above; for pure $\boldsymbol{AC}$, they can be removed.

Every term $\mathbf{t}$ of $\boldsymbol{AC}$ has a unique type $\tau$; if any ambiguity arises as to what that type is we will indicate it by a superscript: $\mathbf{t}^{\tau}$. Locations are also uniquely typed *a priori*.

To formalize the syntax of $\boldsymbol{AC}$, we begin by specifying the set $\boldsymbol{Loc}$ of locations, first by identifying what their types can be:

**Definition 2.3.** The set $\boldsymbol{LType}$ of types of locations is a finite subset of $\boldsymbol{Type}$ that does not contain $\mathsf{S}$ (but can contain $\mathsf{S} \to \tau$ for any $\tau$).

[2] By choosing a conservative approach to types, we intend to make it as easy as possible to extend and study the type system in future work.

We only allow a finite number of types of locations for technical reasons which will be made clear in Chapter 3 (see discussion in §3.2.1). Locations are (arbitrary) syntactic objects; the only criterion is that they be testable for (syntactic) identity.

**Definition 2.4.** $\boldsymbol{Loc}^\tau$ is the (countable) set of all locations of type $\tau$. The set of all locations is $\boldsymbol{Loc} = \bigcup_{\tau \in \boldsymbol{LType}} \boldsymbol{Loc}^\tau$.

The definitions of stores, terms (and later states) are all dependent on $\boldsymbol{Loc}$. Rather than making this dependence explicit, we will just assume that we are always working with some predefined set of locations.

Now we define the set of terms $\boldsymbol{Term}$ of $\boldsymbol{AC}$. For each type $\tau$, the set of terms of type $\tau$ is written $\boldsymbol{Term}^\tau$; these sets are defined in a mutual recursive manner as follows.

**Definition 2.5** (Syntax of $\boldsymbol{AC}$).

1. $\boldsymbol{X} \in \boldsymbol{Loc}^\tau \Rightarrow \boldsymbol{X} \in \boldsymbol{Term}^\tau$
2. $\mathbf{t} \in \boldsymbol{Term}^\tau \Rightarrow \mathbf{!t} \in \boldsymbol{Term}^{\mathbf{S}\to\tau}$
3. $\mathbf{t} \in \boldsymbol{Term}^{\mathbf{S}\to\tau} \Rightarrow \mathbf{?t} \in \boldsymbol{Term}^\tau$
4. $\boldsymbol{X} \in \boldsymbol{Loc}^\tau, \mathbf{t} \in \boldsymbol{Term}^\tau \Rightarrow \boldsymbol{X}^\tau := \mathbf{t}^\tau \in \boldsymbol{Term}^{\mathbf{S}}$
5. $\mathbf{t} \in \boldsymbol{Term}^{\mathbf{S}}, \mathbf{u} \in \boldsymbol{Term}^\tau \Rightarrow \mathbf{t;u} \in \boldsymbol{Term}^\tau$

As for the supplementary operators,

6. $\mathbf{b} \in \boldsymbol{Term}^{\mathbf{B}}$
7. $\mathbf{n} \in \boldsymbol{Term}^{\mathbf{N}}$
8. $\mathbf{t}, \mathbf{u} \in \boldsymbol{Term}^{\mathbf{B}} \Rightarrow \mathbf{t} \wedge \mathbf{u} \in \boldsymbol{Term}^{\mathbf{B}}$, sim. for other Boolean ops.
9. $\mathbf{t}, \mathbf{u} \in \boldsymbol{Term}^{\mathbf{N}} \Rightarrow \mathbf{t} < \mathbf{u} \in \boldsymbol{Term}^{\mathbf{B}}$, sim. for other comparison ops.
10. $\mathbf{t}, \mathbf{u} \in \boldsymbol{Term}^{\mathbf{N}} \Rightarrow \mathbf{t} + \mathbf{u} \in \boldsymbol{Term}^{\mathbf{N}}$, sim. for other arithmetic ops.
11. $\mathbf{t} \in \boldsymbol{Term}^{\mathbf{B}}, \mathbf{u}, \mathbf{v} \in \boldsymbol{Term}^\tau \Rightarrow \mathbf{if\ t\ then\ u\ else\ v} \in \boldsymbol{Term}^\tau$

The set of all terms is then defined as $\boldsymbol{Term} = \bigcup_{\tau \in \boldsymbol{Type}} \boldsymbol{Term}^\tau$.

**Remarks 2.6.**

1. The intension of a term $\mathbf{t}$ of type $\tau$ is of type $\mathbf{S} \to \tau$, and the extension of an (intensional) term $\mathbf{t}$ of type $\mathbf{S} \to \tau$ is of type $\tau$. This is the reflection of formation and invocation in the type system.
2. The assignment construct is of type $\mathbf{S}$. It does not result in a "value and a side-effect," but rather in the "effect" *itself.*
3. The sequence operator allows types other than $\mathbf{S}$ on its right-hand side. This aspect of $\boldsymbol{AC}$ is discussed further in §2.5.

Assignments are of type $\mathbf{S}$ due to the fact that they return stores, and the type of the location (on the left-hand side) and the assigned term must be in agreement. We do not allow locations of type $\mathbf{S}$ in the *usual* version of $\boldsymbol{AC}$ (see, however, §5.3), but we *do* allow locations of type $\mathbf{S} \to \tau$ for any $\tau$. This amounts to storage of intensions in the store, which accounts for much of $\boldsymbol{AC}$'s expressive power.

To see this, we need to develop the intuition for intension a little further. Operationally, we can think of $\mathbf{i}\mathbf{t}$ as representing the "text" of $\mathbf{t}$, which, in an actual computer, is the way that procedures and programs are stored. Now consider the term

$$\boldsymbol{X} := \mathbf{i}!\boldsymbol{X},$$

which is read "store in $\boldsymbol{X}$ the procedure that invokes $\boldsymbol{X}$". Once this action is performed, an invocation of $\boldsymbol{X}$

$$\bot \underset{\text{def}}{=} \boldsymbol{X} := \mathbf{i}!\boldsymbol{X};\ !\boldsymbol{X}$$

will proceed to invoke $\boldsymbol{X}$ again and again, leading to divergence. Placing intensions in the store leads to the possibility of unbounded recursion and thus non-termination. Note that $\bot$ can be defined to be of any type $\tau$ by selecting an $\boldsymbol{X}$ of type $\mathbf{S} \to \tau$, and also that $\bot$ is the simplest form of term that always diverges.

## 2.3 Operational Semantics of $AC$

In this section we provide an operational interpretation for $\boldsymbol{AC}$. The first step is to formalize the *store*, which so far has been used in a rough-and-ready intuitive form: Stores, ranged over by $\mathsf{s}$, are functions from $\boldsymbol{Loc}$ into $\boldsymbol{Term}$ that respect types in that members of $\boldsymbol{Loc}^{\tau}$ are only mapped to members of $\boldsymbol{Term}^{\tau}$. This characterization is accurate but not precise, in that stores actually map locations to a particular *subset* of $\boldsymbol{Term}$; it will however serve our purposes until it is formalized properly by Definition 2.19.

We access the contents of $\boldsymbol{X}$ in a store $\mathsf{s}$ by function application $\mathsf{s}(\boldsymbol{X})$, and we will use the standard notion of *function variant* to update the contents of a location.

**Definition 2.7** (Variant)**.** The variant of a function $f$ at $x$ for $d$ is the function $f[x/d]$, where $(f[x/d])(x) = d$ and $(f[x/d])(y) = f(y)$ for any $y \neq x$.

Updating store $\mathsf{s}$ so that $\boldsymbol{X}$ now contains $\mathsf{t}$ gives the new store $\mathsf{s}[\boldsymbol{X}/\mathsf{t}]$.

We now proceed to formalize the behaviour of $\boldsymbol{AC}$ by giving its operational semantics. To do this we will use Plotkin's *structural operational semantics* [Plo81], in the big-step [Win93] (or "natural" [Kah87]) style. The rules define the *computation relation* $\Downarrow \subset ((\boldsymbol{Term} \times \boldsymbol{Store}) \times (\boldsymbol{Term} \cup \boldsymbol{Store}))$. Really, since a term can compute *either* a store (if the term is of type $\mathbf{S}$) *or* another term (if it is of any type other than $\mathbf{S}$), the computation relation can be broken into two disjoint relations $\Downarrow_c \subset ((\boldsymbol{Term}^{\mathbf{S}} \times \boldsymbol{Store}) \times \boldsymbol{Store})$ and $\Downarrow_v \subset ((\boldsymbol{Term}^{\tau} \times \boldsymbol{Store}) \times \boldsymbol{Term})$. However, since the rules are so simple, we choose instead to use the metavariable $\mathsf{d}$ to range over *both* terms and stores, and give the rules as follows:

**Definition 2.8** (Operational semantics of $\boldsymbol{AC}$)**.** First, the rules for locations, assignment and sequence are standard [Win93].

$$\frac{\mathsf{s}(\boldsymbol{X}) = \mathsf{t}}{\boldsymbol{X},\ \mathsf{s} \Downarrow \mathsf{t}} \qquad \frac{\mathsf{t},\ \mathsf{s} \Downarrow \mathsf{u}}{\boldsymbol{X} := \mathsf{t},\ \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/\mathsf{u}]} \qquad \frac{\mathsf{t},\ \mathsf{s} \Downarrow \mathsf{s}' \qquad \mathsf{u},\ \mathsf{s}' \Downarrow \mathsf{d}}{\mathsf{t};\ \mathsf{u},\ \mathsf{s} \Downarrow \mathsf{d}}$$

Then, our rules for intension and extension:

$$\frac{}{\mathbf{\grave{t}},\ \mathbf{s} \Downarrow \mathbf{\grave{t}}} \qquad \frac{\mathbf{t},\ \mathbf{s} \Downarrow \mathbf{\grave{u}} \qquad \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{d}}{\mathbf{!t},\ \mathbf{s} \Downarrow \mathbf{d}}$$

Then the rest (also standard). Recall that these are not a part of pure $\boldsymbol{AC}$:

$$\frac{}{\mathbf{b},\ \mathbf{s} \Downarrow \mathbf{b}} \qquad \frac{}{\mathbf{n},\ \mathbf{s} \Downarrow \mathbf{n}}$$

$$\frac{\mathbf{t},\ \mathbf{s} \Downarrow \mathbf{n}_1 \qquad \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{n}_2}{\mathbf{t}+\mathbf{u},\ \mathbf{s} \Downarrow \mathbf{n}}$$

where $\mathbf{n}$ is the sum of $\mathbf{n}_1$ and $\mathbf{n}_2$; sim. for other arithmetic operators

$$\frac{\mathbf{t},\ \mathbf{s} \Downarrow \mathbf{b}_1 \qquad \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{b}_2}{\mathbf{t} \wedge \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{b}}$$

where $\mathbf{b}$ is the conjunction of $\mathbf{b}_1$ and $\mathbf{b}_2$; sim. for other Boolean operators

$$\frac{\mathbf{t},\ \mathbf{s} \Downarrow \mathbf{n}_1 \qquad \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{n}_2}{\mathbf{t}<\mathbf{u},\ \mathbf{s} \Downarrow \mathbf{b}}$$

where $\mathbf{b}$ is **true** iff $\mathbf{n}_1$ is less than $\mathbf{n}_2$; sim. for other comparison operators

$$\frac{\mathbf{t},\ \mathbf{s} \Downarrow \mathbf{true} \qquad \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{d}}{\mathbf{if\ t\ then\ u\ else\ v},\ \mathbf{s} \Downarrow \mathbf{d}} \qquad \frac{\mathbf{t},\ \mathbf{s} \Downarrow \mathbf{false} \qquad \mathbf{v},\ \mathbf{s} \Downarrow \mathbf{d}}{\mathbf{if\ t\ then\ u\ else\ v},\ \mathbf{s} \Downarrow \mathbf{d}}$$

As an example of how to interpret the above definitions, consider the assignment construct. Its rule should be read: If $\mathbf{t}$ computes $\mathbf{d}$ given store $\mathbf{s}$, then $\boldsymbol{X} := \mathbf{t}$, when evaluated in store $\mathbf{s}$, computes $\mathbf{s}[\boldsymbol{X}/\mathbf{d}]$.

The intuition for the rules for intension and extension are as follows:

> *Intension "holds back" the computation of a term,*
>
> *and extension "induces" computation.*

These rules provide an operational interpretation for Montague's intension and extension operators. While [HR77] translated Montague's operators into LISP fragments, we believe that ours is the first operational semantic description of these.

It is important to note that there are *no side-effects in* $\boldsymbol{AC}$; if a term is of type $\tau \neq \mathbf{S}$ then it only results in a value. This has interesting consequences; see §2.5 for a detailed discussion.

As a first simple result about our operational semantics, we show that computation is *unique and deterministic*:

**Lemma 2.9.** *There is at most one derivation for any* $\mathbf{t}, \mathsf{s}$.

*Proof.* Formally, the proof proceeds by induction on derivations. However, by inspection, we see that each construct appears on the bottom of exactly one rule except for the conditional. So the property is maintained by everything but the conditional. In that case, by IH, there is at most one derivation resulting in either **true** or **false**, so only one of the cases applies, which completes the proof. □

If there is no $\mathbf{d}$ such that $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{d}$, $\mathbf{t}$ is said to *diverge on store* $\mathsf{s}$.

Let us return to our examples from §2.1. First, here is the operational interpretation of example 1: for any $\mathsf{s}$,

$$\dfrac{\dfrac{1, \mathsf{s} \Downarrow 1}{\boldsymbol{X} := 1, \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/1]} \quad \dfrac{\dfrac{\dfrac{\mathsf{s}[\boldsymbol{X}/1](\boldsymbol{X}) = 1}{\boldsymbol{X}, \mathsf{s}[\boldsymbol{X}/1] \Downarrow 1} \quad \dfrac{\mathsf{s}[\boldsymbol{X}/1][\boldsymbol{Y}/1](\boldsymbol{Y}) = 1}{\boldsymbol{Y}, \mathsf{s}[\boldsymbol{X}/1][\boldsymbol{Y}/1] \Downarrow 1}}{\boldsymbol{Y} := \boldsymbol{X}, \mathsf{s}[\boldsymbol{X}/1] \Downarrow \mathsf{s}[\boldsymbol{X}/1][\boldsymbol{Y}/1]}}{\boldsymbol{Y} := \boldsymbol{X};\ \boldsymbol{Y}, \mathsf{s}[\boldsymbol{X}/1] \Downarrow 1}}{\boldsymbol{X} := 1;\ \boldsymbol{Y} := \boldsymbol{X};\ \boldsymbol{Y}, \mathsf{s} \Downarrow 1}$$

Next, example 2. For any $\mathsf{s}$, $\mathsf{s}' = \mathsf{s}[\boldsymbol{P}/\mathbf{i}\boldsymbol{X}]$ and $\mathsf{s}'' = \mathsf{s}[\boldsymbol{P}/\mathbf{i}\boldsymbol{X}][\boldsymbol{X}/1]$,

$$\dfrac{\dfrac{\mathbf{i}\boldsymbol{X}, \mathsf{s} \Downarrow \mathbf{i}\boldsymbol{X}}{\boldsymbol{P} := \mathbf{i}\boldsymbol{X}, \mathsf{s} \Downarrow \mathsf{s}'} \quad \dfrac{\dfrac{1, \mathsf{s}' \Downarrow 1}{\boldsymbol{X} := 1, \mathsf{s}' \Downarrow \mathsf{s}''} \quad \dfrac{\dfrac{\mathsf{s}''(\boldsymbol{P}) = \mathbf{i}\boldsymbol{X}}{\boldsymbol{P}, \mathsf{s}'' \Downarrow \mathbf{i}\boldsymbol{X}} \quad \dfrac{\mathsf{s}''(\boldsymbol{X}) = 1}{\boldsymbol{X}, \mathsf{s}'' \Downarrow 1}}{!\boldsymbol{P}, \mathsf{s}'' \Downarrow 1}}{\boldsymbol{X} := 1;\ !\boldsymbol{P}, \mathsf{s}'' \Downarrow 1}}{\boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{X} := 1;\ !\boldsymbol{P}, \mathsf{s} \Downarrow 1}$$

Example 3 is different than the other two examples in that it depends on the input store, specifically, on the value of $\boldsymbol{X}$. A term that has such a dependence is called *operationally non-rigid*, and one that has no such dependence is called *operationally*

*rigid.* Such classifications of terms are the subject of the next section. A full derivation of example 3 is unwieldy, but we can demonstrate a single step of the factorial calculation. Let

$$\mathbf{p} \equiv \textbf{if } \boldsymbol{X} > 1 \textbf{ then } \boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - 1;\ !\boldsymbol{P}) \textbf{ else } 1,$$

and take a store $\mathsf{s}$ s.t. $\mathsf{s}(\boldsymbol{P}) = \mathbf{ip}$ and $\mathsf{s}(\boldsymbol{X}) = 4$. The partial derivation of a step in the factorial is as follows:

$$\dfrac{\dfrac{\mathsf{s}(\boldsymbol{P})=\mathbf{ip}}{\boldsymbol{P},\mathsf{s} \Downarrow \mathbf{ip}} \quad \dfrac{\dfrac{\dfrac{\mathsf{s}(\boldsymbol{X})=4}{\boldsymbol{X},\mathsf{s} \Downarrow 4} \quad \dfrac{}{1,\mathsf{s} \Downarrow 1}}{(\boldsymbol{X} > 1),\mathsf{s} \Downarrow \textbf{true}} \quad \dfrac{\dfrac{\mathsf{s}(\boldsymbol{X})=4}{\boldsymbol{X},\mathsf{s} \Downarrow 4} \quad \dfrac{\dfrac{\dfrac{\dfrac{\mathsf{s}(\boldsymbol{X})=4}{\boldsymbol{X},\mathsf{s} \Downarrow 4} \quad \dfrac{}{1,\mathsf{s} \Downarrow 1}}{(\boldsymbol{X}-1),\mathsf{s} \Downarrow 3} \quad \vdots}{(\boldsymbol{X} := \boldsymbol{X}-1),\mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/3]} \quad \dfrac{\vdots}{!\boldsymbol{P},\mathsf{s}[\boldsymbol{X}/3] \Downarrow 6}}{(\boldsymbol{X} := \boldsymbol{X}-1;\ !\boldsymbol{P}),\mathsf{s} \Downarrow 6}}{(\boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X}-1;\ !\boldsymbol{P})),\mathsf{s} \Downarrow 24}}{(\textbf{if } \boldsymbol{X} > 1 \textbf{ then } \boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X}-1;\ !\boldsymbol{P}) \textbf{ else } 1),\mathsf{s} \Downarrow 24}}{!\boldsymbol{P},\mathsf{s} \Downarrow 24}$$

## 2.4 Rigidity, Modal Closedness, and Canonicity

This section is devoted to studying the properties of terms of $\boldsymbol{AC}$ and their operational derivations.

**Definition 2.10** (Operational equivalence). Two terms $\mathbf{t}$ and $\mathbf{u}$ are operationally equivalent if, for every store $\mathsf{s}$, either both $\mathbf{t},\mathsf{s}$ and $\mathbf{u},\mathsf{s}$ diverge or there is a $\mathbf{d}$ s.t. $\mathbf{t},\mathsf{s} \Downarrow \mathbf{d}$ and $\mathbf{u},\mathsf{s} \Downarrow \mathbf{d}$. We write $\mathbf{t} \overset{\circ}{=} \mathbf{u}$ if this is the case.

**Definition 2.11** (Operational rigidity). A term $\mathbf{t}$ is called *operationally rigid* iff there is a $\mathbf{u}$ s.t. all stores $\mathsf{s}$ give $\mathbf{t},\mathsf{s} \Downarrow \mathbf{u}$. A term for which this is not the case is called operationally *non-rigid.*

Operational rigidity is a condition that is difficult to characterize simply. For example, $\boldsymbol{X} := \boldsymbol{X};\ \mathbf{1}$ is rigid, but its operational derivation does depend on the value stored in $\boldsymbol{X}$. It is therefore useful to approximate rigidity with more readily identifiable properties. The first such approximation that we will consider is the set of locations that are accessed during computation.

**Definition 2.12** (Access set). If $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{d}$, the *access set* of $\mathbf{t}$ given $\mathsf{s}$ is the set consisting of all of the locations that are used in the derivation tree of $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{d}$. We write this as $\boldsymbol{acc}(\mathbf{t}, \mathsf{s})$.

More formally, we can define the access set by induction on derivations:

1. $\boldsymbol{acc}(\mathbf{b}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{n}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{i}\mathbf{t}, \mathsf{s}) = \emptyset$
2. $\boldsymbol{acc}(\boldsymbol{X}, \mathsf{s}) = \{\boldsymbol{X}\}$
3. $\boldsymbol{acc}(\boldsymbol{X} := \mathbf{t}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{t}, \mathsf{s}) \cup \{\boldsymbol{X}\}$.
4. If $\mathbf{t}^{\mathbf{s}\to\tau}, \mathsf{s} \Downarrow \mathbf{i}\mathbf{u}$, then $\boldsymbol{acc}(!\mathbf{t}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{t}, \mathsf{s}) \cup \boldsymbol{acc}(\mathbf{u}, \mathsf{s})$
5. If $\mathbf{t}^{\mathbf{s}}, \mathsf{s} \Downarrow \mathsf{s}'$, then $\boldsymbol{acc}(\mathbf{t}\,;\mathbf{u}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{t}, \mathsf{s}) \cup \boldsymbol{acc}(\mathbf{u}, \mathsf{s}')$
6. $\boldsymbol{acc}(\textbf{if } \mathbf{t} \textbf{ then } \mathbf{u} \textbf{ else } \mathbf{v}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{t}, \mathsf{s}) \cup \boldsymbol{acc}(\mathbf{u}, \mathsf{s}) \cup \boldsymbol{acc}(\mathbf{v}, \mathsf{s})$; sim. for all other operators.

In the case that $\mathbf{t}, \mathsf{s}$ diverges, we (arbitrarily) set $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}) = \boldsymbol{Loc}$.

**Lemma 2.13.** *If there is an* $\mathsf{s}$ *s.t.* $\boldsymbol{acc}(\mathbf{t}, \mathsf{s})$ *is empty, then for* every $\mathsf{s}'$, $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}')$ *is empty.*

*Proof.* By induction on derivations.

1. *Base cases*: By definition, for all $\mathsf{s}$, $\boldsymbol{acc}(\mathbf{b}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{n}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{i}\mathbf{t}, \mathsf{s}) = \emptyset$.
2. $\boldsymbol{X}$: For all $\mathsf{s}$, $\boldsymbol{acc}(\boldsymbol{X}, \mathsf{s}) = \{\boldsymbol{X}\} \neq \emptyset$; therefore this case is vacuous.
3. $\boldsymbol{X} := \mathbf{t}$: For all $\mathsf{s}$, $\boldsymbol{acc}(\boldsymbol{X} := \mathbf{t}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{t}, \mathsf{s}) \cup \{\boldsymbol{X}\} \neq \emptyset$: vacuous.

4. $!\mathbf{t}$: If $\boldsymbol{acc}(!\mathbf{t}, \mathsf{s}) = \emptyset$, then $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}) = \emptyset$ and, for the $\mathbf{u}$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow \mathsf{i}\mathbf{u}$, $\boldsymbol{acc}(\mathbf{u}, \mathsf{s}) = \emptyset$. By IH, for any $\mathsf{s}'$, $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}') = \emptyset$ and $\boldsymbol{acc}(\mathbf{u}, \mathsf{s}') = \emptyset$; therefore, $\boldsymbol{acc}(!\mathbf{t}, \mathsf{s}') = \emptyset$.
5. $\mathbf{t}\,;\mathbf{u}$: If $\boldsymbol{acc}(\mathbf{t}\,;\mathbf{u}, \mathsf{s}) = \emptyset$, then $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}) = \emptyset$ and, for the $\mathsf{s}'$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}'$, $\boldsymbol{acc}(\mathbf{u}, \mathsf{s}') = \emptyset$. By IH, for any $\mathsf{s}''$, $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}'') = \emptyset$ and for the $\mathsf{s}'''$ s.t. $\mathbf{t}, \mathsf{s}'' \Downarrow \mathsf{s}'''$, $\boldsymbol{acc}(\mathbf{u}, \mathsf{s}''') = \emptyset$; therefore, $\boldsymbol{acc}(\mathbf{t}\,;\mathbf{u}, \mathsf{s}'') = \emptyset$.
6. *Other operators, e.g., the conditional*: If $\boldsymbol{acc}(\textbf{if t then u else v}, \mathsf{s})$ is empty, then $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{u}, \mathsf{s}) = \boldsymbol{acc}(\mathbf{v}, \mathsf{s}) = \emptyset$. By IH, for any $\mathsf{s}'$, $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}')$, $\boldsymbol{acc}(\mathbf{u}, \mathsf{s}')$, and $\boldsymbol{acc}(\mathbf{v}, \mathsf{s}')$ are empty, so $\boldsymbol{acc}(\textbf{if t then u else v}, \mathsf{s}') = \emptyset$. □

The reason that the access set depends not only on $\mathbf{t}$ but also on the input store is demonstrated by the following example: let $\mathbf{t} \equiv !\boldsymbol{P}$, and let $\mathsf{s}$ be a store where $\mathsf{s}(\boldsymbol{P}) = \mathsf{i}\boldsymbol{X}$. Then $\mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}(\boldsymbol{X})$, so $\boldsymbol{acc}(\mathbf{t}, \mathsf{s}) = \{\boldsymbol{P}, \boldsymbol{X}\}$.

**Lemma 2.14.** *If* $\boldsymbol{acc}(\mathbf{t}, \mathsf{s})$ *is empty, then* $\mathbf{t}$ *is operationally rigid.*

*Proof.* The proof of Lemma 2.13, slightly modified, gives this result. □

To provide a *syntactic* approximation to rigidity, we follow Montague and define the set of *modally closed* terms as follows:

**Definition 2.15** (Modal Closedness)**.** The set of modally closed terms $\boldsymbol{MC}$, ranged over by $\mathbf{mc}$, is generated by

$$\begin{aligned}\mathbf{mc} ::= {} & \mathbf{b} \mid \mathbf{n} \mid \mathsf{i}\mathbf{t} \mid \mathbf{mc}_1 + \mathbf{mc}_2 \mid \mathbf{mc}_1 < \mathbf{mc}_2 \mid \mathbf{mc}_1 \wedge \mathbf{mc}_2 \\ & \mid \textbf{if } \mathbf{mc}_1 \textbf{ then } \mathbf{mc}_2 \textbf{ else } \mathbf{mc}_3\end{aligned}$$

with similar clauses for other arithmetic, comparison and Boolean operators.

The following lemma shows that modal closedness is a stronger condition than the access set being empty.

**Lemma 2.16.** $\mathbf{t} \in \boldsymbol{MC} \Longrightarrow \boldsymbol{acc}(\mathbf{t}, \mathsf{s}) = \emptyset$

*Proof.* By a simple structural induction on the forms of modally closed terms. □

Combining the two lemmas above, we have that modally closed terms are operationally rigid.

**Lemma 2.17.** $\mathbf{t} \in \boldsymbol{MC} \Longrightarrow \mathbf{t}$ *is operationally rigid.*

*Proof.* By Lemmas 2.14 and 2.16. □

Modal closedness captures the intuitive notion of a completed imperative computation, but it can leave arithmetic and Boolean computations unfinished. Terms in which all of these computations are also complete are called *canonical* and are defined by:

**Definition 2.18** (Canonical terms)**.** The set of canonical terms $\boldsymbol{Can}$, ranged over by $\mathbf{c}$, is generated by:

$$\mathbf{c} ::= \mathbf{b} \mid \mathbf{n} \mid \mathbf{it}$$

$\boldsymbol{Can}^{\tau}$ of course means the set of canonical terms of type $\tau$. Clearly, $\boldsymbol{Can} \subseteq \boldsymbol{MC}$; without supplementary operators $\boldsymbol{Can}$ is identical to $\boldsymbol{MC}$.

The characterization of terms given above can be presented concisely as

$$\mathbf{t} \in \boldsymbol{Can} \Rightarrow \mathbf{t} \in \boldsymbol{MC} \Rightarrow \boldsymbol{acc}(\mathbf{t}, \mathsf{s}) = \emptyset \Rightarrow \mathbf{t} \text{ is operationally rigid}$$

Now we get to the point of these definitions: we want to ensure that stores only contain canonical terms. Doing so allows us to ensure that the operational semantics is well-defined. The following definition is the promised refinement to our earlier definition of stores:

**Definition 2.19** (Properness of Stores)**.** A store $\mathsf{s}$ is called *proper* if it only maps locations to canonical terms. Define the set of stores $\boldsymbol{Store}$ as $\bigcup_{\tau \in \boldsymbol{LType}} (\boldsymbol{Loc}^{\tau} \to \boldsymbol{Can}^{\tau})$.

The main result of this chapter is that the operational semantics only produces canonical terms or proper stores:

**Theorem 2.20** (Properness of Operational Semantics). *If* $\mathsf{s}$ *is proper, then for any* $\mathbf{t}$ *where* $\mathbf{t}, \mathsf{s}$ *converges:*

1. *If* $\mathbf{t}$ *is of type* $\mathbf{S}$ *then there is a proper store* $\mathsf{s}'$ *s.t.* $\mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}'$;
2. *Otherwise, there is a* $\mathbf{c}$ *s.t.* $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{c}$.

*Proof.* By induction on derivations.

1. $\mathbf{b}$, $\mathbf{n}$, and $\mathbf{it}$ are already canonical.
2. $\boldsymbol{X}, \mathsf{s} \Downarrow \mathsf{s}(\boldsymbol{X})$. Since $\mathsf{s}$ is proper, $\mathsf{s}(\boldsymbol{X})$ is canonical.
3. $\mathbf{!t}, \mathsf{s} \Downarrow \mathbf{d}$ implies that there is a $\mathbf{u}$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{iu}$ and $\mathbf{u}, \mathsf{s} \Downarrow \mathbf{d}$; by IH, $\mathbf{d}$ is either canonical or proper depending on the type of $\mathbf{t}$.
4. $\boldsymbol{X} := \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/\mathbf{u}]$. Then $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{u}$, which means by IH that $\mathbf{u}$ is canonical, which, because $\mathsf{s}$ is already proper, gives that $\mathsf{s}[\boldsymbol{X}/\mathbf{u}]$ is also proper.
5. $\mathbf{t}\,;\mathbf{u}, \mathsf{s} \Downarrow \mathbf{d}$ provides that there is a $\mathsf{s}'$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}'$ and $\mathbf{u}, \mathsf{s}' \Downarrow \mathbf{d}$. By IH, $\mathsf{s}'$ is proper; this then gives by IH that $\mathbf{d}$ is either canonical or proper depending on $\mathbf{u}$'s type.
6. **if t then u else v**, $\mathsf{s} \Downarrow \mathbf{d}$.

   *Case 1:* $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{true}$ and $\mathbf{u}, \mathsf{s} \Downarrow \mathbf{d}$. By IH, $\mathbf{d}$ is either canonical or proper.

   *Case 2:* $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{false}$. Similar to case 1.
7. Other supplementary operators are treated in a similar manner as the conditional. □

This completes the operational characterization of $\boldsymbol{AC}$.

## 2.5 State Backtracking

There are some notable differences between $\boldsymbol{AC}$ and traditional imperative programming languages. The most important such difference is indicated by Remark 2.6.2: there are *no side-effects*. In $\boldsymbol{AC}$, terms represent either *stores* (effects), if they are of type $\mathbf{S}$, or *values* if they are of some other type.

Firstly, a language based on side-effects can easily be translated into $\boldsymbol{AC}$ by selecting one location (of each type), say $\boldsymbol{R}$, as a *result location*; any expression in that language that returns a pair $(\mathbf{value}, \mathbf{effect})$ would be roughly translated into $\mathbf{effect};\ \boldsymbol{R} := \mathbf{value}$. This is based on the simple observation that the state is a rather large structure that is fully capable of containing any results that we may need, and therefore that returning a value separately from the state is superfluous. There is no loss of generality in moving to a side-effect-free setting as we have done.

Secondly, in the absence of side-effects, a strange phenomenon occurs when we return a value: state backtracking.

State backtracking, or non-persistent or local state update, occurs in $\boldsymbol{AC}$ when we have terms of the form

$$\mathbf{t}\,;\mathbf{u}^{\tau} \quad \text{where} \quad \tau \neq \mathbf{S} \tag{1}$$

because state changes caused by $\mathbf{t}$ are "lost," "localized" or "unrolled" when the value of $\mathbf{u}$ is returned. Consider the following example:

$$\boldsymbol{Y} := (\boldsymbol{X} := 1;\ \ \boldsymbol{X}).$$

Changes to $\boldsymbol{X}$ are *local to the computation of* $\boldsymbol{Y}$, so in fact the above term is equivalent to

$$\boldsymbol{Y} := 1$$

as demonstrated by the following derivation tree:

$$\dfrac{\dfrac{\dfrac{1, \mathsf{s} \Downarrow 1}{\boldsymbol{X} := 1, \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/1]} \quad \dfrac{\mathsf{s}[\boldsymbol{X}/1](\boldsymbol{X}) = 1}{\boldsymbol{X}, \mathsf{s}[\boldsymbol{X}/1] \Downarrow 1}}{\boldsymbol{X} := 1;\ \boldsymbol{X}, \mathsf{s} \Downarrow 1}}{\boldsymbol{Y} := (\boldsymbol{X} := 1;\ \boldsymbol{X}), \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{Y}/1]}$$

Similar behaviour can be observed in Example 3, in the way that the key state update, $\boldsymbol{X} := \boldsymbol{X} - 1$, only applies within the scope of the multiplication operator at each recursive stage.

State backtracking is reminiscent of dynamic scoping [Hun90], and is closely related to the concept of *fluid variables* in LISP [MHGG79]. It is important to note that only state is reset by the backtracking operator, not control.

Because we include terms of the form (1), we can derive a clean rewriting system for $\boldsymbol{AC}$. It gives us a way to "push assignments into terms". This will be discussed and defined in detail in Chapter 4; for now, consider this example:

$$\boldsymbol{X} := 1;\ \boldsymbol{X} := 2 \times (\boldsymbol{X} + \boldsymbol{X}) \tag{2}$$

By intuition and the operational semantics, we know that this term is equivalent to $\boldsymbol{X} := 4$. But how can it be possible, without overly complicated rules, to *rewrite* (2) to it? By admitting terms like (1), we can express *intermediate steps* of the computation that could not otherwise be written. Using the rules from Chapter 4,

$$\begin{aligned}
& \boldsymbol{X} := 1;\ \boldsymbol{X} := 2 \times (\boldsymbol{X} + \boldsymbol{X}) \\
\Rrightarrow\ & \boldsymbol{X} := (\boldsymbol{X} := 1;\ 2 \times (\boldsymbol{X} + \boldsymbol{X})) \\
\Rrightarrow\ & \boldsymbol{X} := ((\boldsymbol{X} := 1;\ 2) \times (\boldsymbol{X} := 1;\ (\boldsymbol{X} + \boldsymbol{X}))) \\
\Rrightarrow\ & \boldsymbol{X} := (2 \times ((\boldsymbol{X} := 1;\ \boldsymbol{X}) + (\boldsymbol{X} := 1;\ \boldsymbol{X}))) \\
\Rrightarrow\ & \boldsymbol{X} := (2 \times (1 + 1)) \\
\Rrightarrow\ & \boldsymbol{X} := 4
\end{aligned}$$

The ability to use assignments in local contexts is thus a powerful syntactic tool. Based on its usefulness, the ease of its incorporation, the simplicity of its semantic definitions and rewriting rules, and the fact that it also occurs naturally when we allow cells of type $\mathbf{S}$ (see §5.3), we believe that

*State backtracking is a natural part of imperative reasoning.*

Another difference with traditional imperative languages, one that has been mentioned several times already, is in the way that $\boldsymbol{AC}$ treats procedures. (Parameterless) procedures in the traditional sense are taken to be *state transformers*, that is, functions $\mathbf{S} \to \mathbf{S}$, whereas in $\boldsymbol{AC}$ the treatment of procedures, using intensions which give functions $\mathbf{S} \to \tau$, is much more general. The generalization is actually closely related to the above discussion because it again involves state backtracking.

An intension of type $\mathbf{S} \to \tau$ represents a "functionalized procedure". When such an intension is invoked (extensionalized), it uses the current store to compute a value, but all changes to the store that occur while computing that value are local and are lost once the value is returned. This is again caused by the lack of side-effects in $\boldsymbol{AC}$.

A comparison between this sort of localization of state change and the approach taken in Separation Logic [Rey02] would be quite interesting.

In the next chapter, we will move beyond the simple operational interpretation given here and provide a full *denotational* semantics for $\boldsymbol{AC}$, as well as give a proof of the equivalence of the two forms of semantics.

# Chapter 3

# Semantics of *AC*

In the last chapter, we discussed the interpretation of ***AC*** operationally, that is, in terms of syntactic rules governing the behaviour of ***AC*** terms. While such an approach is simple, easy to understand, and therefore useful for developing intuition, it suffers from a major drawback: it is *syntax-directed.* Evaluation of a term results in either a new term or in a store that contains terms. Intension is treated as an unevaluated term; aside from the insight that intension can be seen as program text, this does not provide any help in understanding its actual *meaning.* Besides this, proofs of properties and equivalences of terms are cumbersome using only the operational semantics.

To overcome these difficulties, we will give a *denotational* semantics for ***AC***. The basic thrust of this approach is to define mathematical objects that serve as meanings for the syntactic objects of ***AC***, and then define mapping(s) from the syntactic objects to the semantic objects. In logic, the approach is standard and goes back to Tarski. In computer science, the development is more recent and is due to Strachey and Scott [SS71, Sto77].

Given a semantic domain $\mathbb{D}$ of *values*—numbers, functions, etc.—we will define a *meaning function* $[\![\cdot]\!]$ that maps terms of ***AC*** into $\mathbb{D}$, so that for any term $\mathbf{t}$ we

will have that $[\![\mathbf{t}]\!] \in \mathbb{D}$. $[\![\mathbf{t}]\!]$ is called $\mathbf{t}$'s meaning or *denotation*. We will also have to develop a mathematical analogue of the store, called a *state*; doing so is somewhat difficult due to the fact that we store intensions. Once the denotational semantics has been fully defined, a proof of the equivalence between it and the operational interpretation will be given.

Our meaning function will respect the *principle of compositionality* [Jan86], in that the meaning of a compound expression is a function of the meanings of its subexpressions.

**Definition 3.1** (Principle of compositionality). For every $n$-ary syntactic operator

$$\Phi \ : \ \textbf{\textit{Term}} \times \cdots \times \textbf{\textit{Term}} \ \rightarrow \ \textbf{\textit{Term}}$$

that forms $\boldsymbol{AC}$ terms from immediate subterms, there is a corresponding function

$$\Psi \ : \ \mathbb{D} \times \cdots \times \mathbb{D} \ \rightarrow \ \mathbb{D}$$

such that

$$[\![\Phi(\mathbf{t}_1, \ldots, \mathbf{t}_n)]\!] = \Psi([\![\mathbf{t}_1]\!], \ldots, [\![\mathbf{t}_n]\!]).$$

In other words, $[\![\cdot]\!]$ is a *homomorphism* from the term algebra into the semantic algebra.

We begin by going over the mathematical underpinnings of denotational semantics: the theory of domains.

## 3.1 Domains

Domains contain and organize the mathematical objects that serve as meanings for syntactic constructs. As far as *containing* the objects goes, domains are just sets. It is in the way that they organize the objects that domains are special: they provide a set of relationships between the objects that orders them according to their *information*

*content.* Domains consist of a set and an ordering relation on that set, together forming a *complete partially ordered set (cpo).*

The following definitions can be found in any textbook on denotational semantics or domain theory, e.g., [dB80].

**Definition 3.2** (Partial orders). A set $\mathbb{D}$ is *partially ordered* by a relation $\sqsubseteq$ iff $\sqsubseteq$ is (i) reflexive, (ii) antisymmetric and (iii) transitive, i.e. for all $x, y, z \in \mathbb{D}$,

$$x \sqsubseteq x \tag{i}$$

$$x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y \tag{ii}$$

$$x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z. \tag{iii}$$

If $\mathbb{D}$ is partially ordered by $\sqsubseteq$, then the pair $(\mathbb{D}, \sqsubseteq)$ is called a *partial order* (po). We usually refer to a po only by the name of its carrier set, with the order assumed. As usual $x \sqsubset y$ means that $x \sqsubseteq y \wedge x \neq y$.

For our purposes, we also assume that all partial orders $\mathbb{D}$ have a *bottom element* $\bot_{\mathbb{D}}$ s.t. $\forall x \in \mathbb{D} \cdot \bot_{\mathbb{D}} \sqsubseteq x$. $\bot$ represents divergence (non-termination).

**Definition 3.3** (Chains and bounds). A chain $\vec{x}$ is a sequence $(x_1, x_2, \ldots)$ s.t. $x_1 \sqsubseteq x_2 \sqsubseteq \cdots$. When it exists, the *supremum* (least upper bound) of $\vec{x}$ is denoted by $\bigsqcup_i x_i$ or $\bigsqcup \vec{x}$ and satisfies

$$\forall i \cdot x_i \sqsubseteq \bigsqcup \vec{x}$$

$$(\forall i \cdot x_i \sqsubseteq z) \implies \bigsqcup \vec{x} \sqsubseteq z$$

Define $(x_1, \ldots) \downarrow k = x_k$. We adopt the following shorthand for chains: $f(\vec{x}) = (f(x_0), f(x_1), \ldots)$ and $\vec{f}(x) = (f_0(x), f_1(x), \ldots)$.

**Definition 3.4** (Complete partial orders). A *complete partial order* (cpo) is a po $\mathbb{D}$ in which all chains $\vec{x}$ have their suprema $\bigsqcup \vec{x}$ in $\mathbb{D}$.

A useful result about chains is the following:

**Lemma 3.5** (Diagonalizing chains). *Given, for* $i, j \in \{0, 1, \ldots\}$, *elements* $x_{ij}$ *of a cpo* $\mathbb{D}$ *that satisfy* $x_{ij} \sqsubseteq x_{kl}$ *whenever* $i \leq k$ *and* $j \leq l$,

$$\lfloor i \rfloor \lfloor j \rfloor x_{ij} = \lfloor k \rfloor x_{kk}$$

*Proof.* See [dB80, Lemma 5.4]. □

We can make cpos of the sets of natural numbers and Booleans in a trivial way: by making each pair of elements other than $\bot$ *incomparable*, e.g., $\bot \sqsubseteq \mathit{tt}$ and $\bot \sqsubseteq \mathit{ff}$ but $\mathit{tt} \not\sqsubseteq \mathit{ff}$ and $\mathit{ff} \not\sqsubseteq \mathit{tt}$. Such a cpo is called *discrete*. $\mathbb{N}$, $\mathbb{B}$ and $\boldsymbol{Loc}^\tau$ will be taken to be discrete cpos. We take arithmetic and Boolean operations to be *strict* when applied to these cpos: for example, if *either* $x = \bot$ or $y = \bot$, then $x \wedge y = \bot$.

As an alternative to $\mathbb{N}$, we can order the natural numbers according to their usual numerical ordering. We will call this the cpo $\mathbb{C}$ of *ordered numbers*:

$$\bot_{\mathbb{C}} = 0 \sqsubset 1 \sqsubset \cdots \sqsubset \omega$$

The inclusion of $\omega$ is necessary because of the condition that cpos include all suprema. (We will not actually be using this cpo until §3.3.)

Now we set about building new cpos out of existing ones. First, to any cpo we can add a new bottom element "below" all of its original elements.

**Definition 3.6** (Lifting). Given a cpo $\mathbb{D}$, the *lifted* cpo $\mathbb{D}_\bot$ consists of all of the elements of $\mathbb{D}$, with the same order relation, supplemented with a new element $\bot_{\mathbb{D}_\bot}$ that is weaker than any element of $\mathbb{D}$. The difference between the two cpos is illustrated by

$$\bot_{\mathbb{D}_\bot} \sqsubset \bot_{\mathbb{D}} \sqsubseteq d \in \mathbb{D}.$$

A finite Cartesian product of cpos is itself a cpo:

**Definition 3.7** (Product cpo). Given cpos $\mathbb{D}_1, \ldots, \mathbb{D}_n$, the product $[\mathbb{D}_1 \times \cdots \times \mathbb{D}_n]$ is a cpo ordered pointwise,

$$(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \iff (\forall i \in \{1, \ldots, n\} \cdot x_i \sqsubseteq y_i)$$

whose bottom element is $(\bot, \ldots, \bot)$. Least upper bounds are determined pointwise: $\bigsqcup_i (\vec{x}_1 \downarrow i, \ldots, \vec{x}_n \downarrow i) = (\bigsqcup \vec{x}_1, \ldots, \bigsqcup \vec{x}_n)$.

To provide a useful cpo structure for types like $\mathbf{S} \to \boldsymbol{\tau}$, we restrict the class of functions under consideration to these:

**Definition 3.8** (Continuous functions). A function $f$ is *monotonic* iff it preserves order, i.e., if $x \sqsubseteq y$ then $f(x) \sqsubseteq f(y)$. $f$ is *continuous* iff, in addition to being monotonic, it preserves suprema: $f(\bigsqcup \vec{x}) = \bigsqcup f(\vec{x})$.

Now we show that the set of continuous functions from a cpo to another cpo can be viewed as a cpo.

**Definition 3.9** (Function cpo). Given two cpos $\mathbb{D}_1$ and $\mathbb{D}_2$, the set of continuous functions from $\mathbb{D}_1$ to $\mathbb{D}_2$ is itself a cpo $[\mathbb{D}_1 \to \mathbb{D}_2]$ with ordering given as follows:

$$f \sqsubseteq g \iff (\forall x \in \mathbb{D}_1 \cdot f(x) \sqsubseteq g(x))$$

The bottom element is $\lambda x \cdot \bot$ and $\bigsqcup \vec{f} = \lambda x \cdot \bigsqcup \vec{f}(x)$.

For our purposes, we will often need to make use of *lifted function spaces* $[\cdot \to \cdot]_\bot$ rather than simple function spaces. The reason for this is discussed after Definition 3.18.

### 3.1.1 Reflexive Domains

Defining ***State***, the mathematical analogue of ***Store***, causes no problems when we are storing values such as numbers or Booleans. But a great deal must be done in order to handle *procedure-valued* locations, which are a basic feature of ***AC***. The reason for this is that states can "contain" procedures, which are intensions or *functions* $\boldsymbol{State} \to \mathbb{D}$ for some $\mathbb{D}$. The usual definition of state as a function from locations to values is thus inadequate as we must somehow store a function from states to

states within a single state. Another way of putting it is that ***State*** has a recursive definition,

$$\textbf{\textit{State}} \simeq (\cdots \textbf{\textit{State}} \cdots), \tag{1}$$

where '$\simeq$' represents isomorphism. This is similar to the situation found in the untyped $\lambda$-calculus, where values must also all be functions so that they can be applied to themselves, i.e., the domain $\mathbb{D}$ must satisfy

$$\mathbb{D} \simeq (\mathbb{D} \to \mathbb{D}). \tag{2}$$

Scott's breakthrough solution to (2) and, more generally, (1) in [Sco70] led to the development of *domain theory* [AJ94, SHLG94]. (Interestingly, in [Sco70] Scott explicitly indicates *storage of commands* as one of the primary motivations for his work.) Domains satisfying such equations are called *reflexive domains*; we construct the set of states, ***State***, as one such.

The particulars of construction of such domains is technically involved and is of little consequence to our purposes, but a rough explanation of the approach is as follows. Say that we had a set of states that consist of two locations: one that stores numbers, and one that stores procedures. Then ***State***'s recursive specification could be written

$$\textbf{\textit{State}} \simeq \mathbb{N} \times [\textbf{\textit{State}} \to \textbf{\textit{State}}]$$

To build our solution, we start by setting $\textbf{\textit{State}}_0 = \{\bot\}$, and then by successively having that $\textbf{\textit{State}}_{n+1} = \mathbb{N} \times [\textbf{\textit{State}}_n \to \textbf{\textit{State}}_n]$. Then we provide a system of embeddings from each $\textbf{\textit{State}}_i$ into $\textbf{\textit{State}}_{i+1}$, and projections from $\textbf{\textit{State}}_{i+1}$ to $\textbf{\textit{State}}_i$. Taking care to ensure that certain conditions are met, ***State*** is constructed as the *inverse limit* of the sequence of approximating domains $\textbf{\textit{State}}_0, \textbf{\textit{State}}_1, \ldots$. Along with this inverse limit, we have a bijection $\textbf{\textit{State}} \to (\mathbb{N} \times [\textbf{\textit{State}} \to \textbf{\textit{State}}])$ that allows us to move between the "collapsed" and "unfolded" view of ***State***. This is somewhat inexact but captures the spirit of the approach.

All that matters to us is that such domains, along with some useful functions on them, exist. The reader interested in the particulars can consult [Sch86] for a nice presentation, or [AJ94] or [SHLG94] for more in-depth information. We start by outlining the domain equations that we can solve.

**Definition 3.10** (Domain signatures). Let ***CPO*** be the class of all cpos. The set of *signatures* ***Sig*** is the set of functions $\Sigma : \boldsymbol{CPO} \to \boldsymbol{CPO}$ that is recursively given by

| | | | | |
|---|---|---|---|---|
| 1. | $\lambda\mathbb{D} \cdot \mathbb{D}$ | $\in$ | ***Sig*** | |
| 2. | $\lambda\mathbb{D} \cdot \mathbb{D}_0$ | $\in$ | ***Sig*** | where $\mathbb{D}_0 \in \{\mathbb{N}, \mathbb{B}, \mathbb{C}, \boldsymbol{Loc}^\tau\}$ |
| 3. | $\lambda\mathbb{D} \cdot \mathbb{D}_\perp$ | $\in$ | ***Sig*** | |
| 4. | $\lambda\mathbb{D} \cdot [\Sigma_1(\mathbb{D}) \times \cdots \times \Sigma_n(\mathbb{D})]$ | $\in$ | ***Sig*** | |
| 5. | $\lambda\mathbb{D} \cdot [\Sigma_1(\mathbb{D}) \to \Sigma_2(\mathbb{D})]$ | $\in$ | ***Sig*** | |

We can create reflexive domains based on any signature.

**Theorem 3.11** (Reflexive domains). *For any $\Sigma \in \boldsymbol{Sig}$, a cpo $\mathbb{D}$ can be constructed s.t. there is a continuous bijection $f : \mathbb{D} \to \Sigma(\mathbb{D})$. We write $\underline{\Sigma}$ for $\mathbb{D}$, $\underrightarrow{\Sigma}$ for $f$ and $\underleftarrow{\Sigma}$ for $f^{-1}$.*

*Proof.* See [SHLG94], Theorem 6.11. □

$\underrightarrow{\Sigma}$ is called $\Sigma$'s *embedding* function, and $\underleftarrow{\Sigma}$ is $\Sigma$'s *projection* function. The embedding can be seen as an unfolding operation, whereas the projection can be seen as a collapsing operation. This view will be useful to us given the way we will be using reflexive domains.

See [Sch86] for an explicit construction of the above solution, as the inverse limit of the sequence $\{\perp\}, \Sigma(\{\perp\}), \Sigma(\Sigma(\{\perp\})), \ldots$. We can also show that the construction gives the smallest possible solution in that it can be embedded into any other solution domain.

(It is also important to note that the set $\mathbb{D}_0$ in the above definition is restricted to cpos that are consistently complete and algebraic [SHLG94]. Since all of our cpos satisfy these criteria we skip the details.)

As a final comment, note that any "non-reflexive" domain corresponds to a signature $\Sigma$ that is a constant function on ***CPO***. With our mathematical toolbox well stocked, we now turn to the semantics of ***AC***.

## 3.2 Semantics

### 3.2.1 States and semantic domains

Recall the definitions of ***Type***, ***LType*** and ***Loc*** given in Definitions 2.2, 2.3 and 2.4. Leaving the domain of states, ***State***, assumed for a moment, we first present the set of semantic domains. To each type $\tau$ is associated a domain $\{\!|\tau|\!\}$; a term of type $\tau$ interpreted in a state ***State*** will result in a value from $\{\!|\tau|\!\}$.

**Definition 3.12** (Semantic domains). To each type $\tau \in$ ***Type***, we associate a domain $\{\!|\tau|\!\}$ as follows:

1. $\{\!|\mathbf{B}|\!\}$ is the cpo of truth values $\mathbb{B}$, ranged over by $\mathsf{b}$,
2. $\{\!|\mathbf{N}|\!\}$ is the cpo of natural numbers $\mathbb{N}$, ranged over by $\mathsf{n}$,
3. $\{\!|\mathbf{S}|\!\}$ is the cpo of states ***State***, ranged over by $\sigma$,
4. $\{\!|\mathbf{S} \to \tau|\!\}$ is the cpo of lifted functions $[\textbf{\textit{State}} \to \{\!|\tau|\!\}]_\perp$.

Let $\textbf{\textit{Dom}} = \bigcup_{\tau \in \textbf{\textit{Type}}} \{\!|\tau|\!\}$. For convenience we will write $\perp_\tau$ for $\perp_{\{\!|\tau|\!\}}$.

But how do we define ***State***? Roughly, as the product of function spaces $\textbf{\textit{Loc}}^\tau \to \{\!|\tau|\!\}$. The hitch, of course, is in the fact that $\{\!|\tau|\!\}$ itself can depend on ***State***! The solution starts with an interpretation of types as signatures.

**Definition 3.13.** The signature of a type $\tau$, $\Sigma^\tau$, is given by:

1. $\Sigma^{\mathbf{B}} = \lambda\mathbb{D} \cdot \mathbb{B}$
2. $\Sigma^{\mathbf{N}} = \lambda\mathbb{D} \cdot \mathbb{N}$
3. $\Sigma^{\mathbf{S}} = \lambda\mathbb{D} \cdot \mathbb{D}$
4. $\Sigma^{\mathbf{S}\to\tau} = \lambda\mathbb{D} \cdot [\mathbb{D} \to \Sigma^\tau(\mathbb{D})]_\perp$

(Notice that $\{\!|\tau|\!\}$ can now be defined as $\Sigma^\tau(\boldsymbol{State})$.) Recall that ***LType*** is a finite set; assume some ordering of its elements $(\tau_1, \cdots, \tau_n)$.

**Definition 3.14** (State)**.** The *state signature*, $\Sigma^{\boldsymbol{State}}$, is

$$\Sigma^{\boldsymbol{State}} = \lambda\mathbb{D} \cdot \left[[\boldsymbol{Loc}^{\tau_1} \to \Sigma^{\tau_1}(\mathbb{D})] \times \cdots \times [\boldsymbol{Loc}^{\tau_n} \to \Sigma^{\tau_n}(\mathbb{D})]\right]$$

Define ***State*** to be the resulting domain $\underline{\Sigma}^{\boldsymbol{State}}$, and we can unfold and collapse it using $\underline{\sum}_{\rightarrow}^{\boldsymbol{State}}$ and $\underline{\sum}_{\leftarrow}^{\boldsymbol{State}}$.

Now the reason for ***LType*** having been restricted to finite size can be clarified: there is no way to solve the domain equation for ***State*** if the product is infinite ([Sch86]). To complete the definition of ***State***, we need a way to access and update the contents of locations.

**Definition 3.15.** Using the enumeration of ***LType*** given above, given $\boldsymbol{X} \in \boldsymbol{Loc}^{\tau_i}$,

$$\sigma(\boldsymbol{X}) \underset{\text{def}}{=} (\underline{\sum}_{\rightarrow}^{\boldsymbol{State}}(\sigma) \downarrow i)(\boldsymbol{X})$$

$$\sigma[\boldsymbol{X}/d] \underset{\text{def}}{=} \underline{\sum}_{\leftarrow}^{\boldsymbol{State}}(\underline{\sum}_{\rightarrow}^{\boldsymbol{State}}(\sigma) \downarrow 1, \ldots, (\underline{\sum}_{\rightarrow}^{\boldsymbol{State}}(\sigma) \downarrow i)[\boldsymbol{X}/d], \ldots, \underline{\sum}_{\rightarrow}^{\boldsymbol{State}}(\sigma) \downarrow n)$$

Notice that $\perp_{\mathbf{S}} = \underline{\sum}_{\leftarrow}^{\boldsymbol{State}}(\lambda\boldsymbol{X}^{\tau_1} \cdot \perp_{\tau_1}, \ldots, \lambda\boldsymbol{X}^{\tau_n} \cdot \perp_{\tau_n})$.

States are the vehicle through which we assign values to locations; as such, they can be viewed as contexts in which ***AC*** terms are interpreted. ***AC*** is thus a *modal* calculus in that states can be viewed as *possible worlds* in the sense of Kripke [Kri59].

### 3.2.2 Denotational Semantics

We can now present the primary tool of this chapter: the *meaning function* that assigns values to terms relative to states.

**Definition 3.16** (Meaning function). The *meaning function* $[\![\cdot]\!]$ is a function that maps, for each $\tau$, $\mathit{Term}^\tau$ into $[\mathit{State} \to \{\!|\tau|\!\}]_\perp = \{\!|\mathsf{S} \to \tau|\!\}$. It takes a term $\mathbf{t}$ and returns a function from states to values; this function is called the *sense* or intensional meaning of $\mathbf{t}$. Applying this function to a state $\sigma$ results in the *denotation* or extensional meaning of $\mathbf{t}$ with respect to $\sigma$.

**Convention 3.17** (Metavariables). The bijective mapping between Boolean constants and Boolean values, and that between numerals and numbers, is expressed as an implicit binding between similar metavariables: if, in some mathematical context, the metavariables $\mathbf{b}$ and $\mathsf{b}$ both occur, then they represent a corresponding Boolean constant and value, and similarly for the case of $\mathbf{n}$ and $\mathsf{n}$.

We are now ready to give the semantics of $\boldsymbol{AC}$.

**Definition 3.18** (Denotational semantics of $\boldsymbol{AC}$). First, we have that the meaning function is *strict*: $[\![\mathbf{t}]\!]\perp^{\mathsf{S}} = \perp$. For any $\sigma \neq \perp$,

1. $[\![\boldsymbol{X}]\!]\sigma = \sigma(\boldsymbol{X})$
2. $[\![\mathbf{i}\mathbf{t}]\!]\sigma = [\![\mathbf{t}]\!]$
3. $[\![\mathbf{!t}]\!]\sigma = [\![\mathbf{t}]\!]\sigma\sigma$
4. $[\![\boldsymbol{X} := \mathbf{t}]\!]\sigma = \sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma]$ if $[\![\mathbf{t}]\!]\sigma \neq \perp$, $\perp$ otherwise
5. $[\![\mathbf{t}\,;\mathbf{u}]\!]\sigma = [\![\mathbf{u}]\!]\,([\![\mathbf{t}]\!]\sigma)$

and the supplementary operators

6. $[\![\mathbf{b}]\!]\sigma = \mathsf{b}$
7. $[\![\mathbf{n}]\!]\sigma = \mathsf{n}$
8. $[\![\mathbf{t} \wedge \mathbf{u}]\!]\sigma = [\![\mathbf{t}]\!]\sigma \wedge [\![\mathbf{u}]\!]\sigma$
9. $[\![\mathbf{t} + \mathbf{u}]\!]\sigma = [\![\mathbf{t}]\!]\sigma + [\![\mathbf{u}]\!]\sigma$
10. $[\![\mathbf{t} < \mathbf{u}]\!]\sigma = [\![\mathbf{t}]\!]\sigma < [\![\mathbf{u}]\!]\sigma$
11. $$[\![\textbf{if t then u else v}]\!]\sigma = \begin{cases} [\![\mathbf{u}]\!]\sigma & \text{if } [\![\mathbf{t}]\!]\sigma = \mathit{tt} \\ [\![\mathbf{v}]\!]\sigma & \text{if } [\![\mathbf{t}]\!]\sigma = \mathit{ff} \\ \bot & \text{otherwise} \end{cases}$$

Some discussion is in order to explain the above definition. First, we have that $\boldsymbol{X}$ means the $\boldsymbol{X}$-projection of the state, which gives us a way to access a value in a location in the state. $\mathbf{\grave{i}t}$, the intension operation, forms a procedure by abstracting the state, whereas $\mathbf{!t}$ invokes the procedure denoted by $\mathbf{t}$ by applying the state to it. Notice that intension is treated, as it is by Montague, as a function from possible worlds (states) to values, and that extension applies such a function to the "current" state. One of the interesting consequences of this is that the *sense* of $\mathbf{t}$ is the same as the *denotation* of $\mathbf{\grave{i}t}$.

$\boldsymbol{X} := \mathbf{t}$ denotes the variant of the state $\sigma$ at $\boldsymbol{X}$ for $[\![\mathbf{t}]\!]\sigma$, providing a way to update the contents of a location. $\mathbf{t;u}$ evaluates $\mathbf{u}$ in the state determined by $\mathbf{t}$. Finally, the supplementary operators are given a standard treatment.

Now that we have developed the denotational semantics of $\boldsymbol{AC}$, the reason for the way we have defined states can be illuminated by an example:

$$[\![\boldsymbol{P} := \grave{\mathbf{i}}(\boldsymbol{X} := \mathbf{1})]\!]\sigma = \sigma[\boldsymbol{P}/\lambda\sigma' \cdot \sigma'[\boldsymbol{X}/1]].$$

Without reflexive domains, "storing" functions on ***State*** within ***State*** would be impossible.

We can also clarify the need for the use of the lifted function space for function types. Consider the following two terms, where $\boldsymbol{X}$ is of type $\mathbf{S} \to \boldsymbol{\tau}$ and $\boldsymbol{Y}$ is of type

$\mathbf{s}\to(\mathbf{s}\to\tau)$.

$$[\![\mathbf{i}(\boldsymbol{X} := \mathbf{i}!\boldsymbol{X};\ !\boldsymbol{X})]\!]\sigma = \lambda\sigma' \cdot \bot^{\tau}$$
$$[\![\boldsymbol{Y} := \mathbf{i}!\boldsymbol{Y};\ !\boldsymbol{Y}]\!]\sigma = \bot^{\mathbf{s}\to\tau}$$

If we took $\bot^{\mathbf{s}\to\tau}$ to be the bottom element of the cpo of continuous functions from ***State*** to $\mathbb{D}_{\tau}$, then the two terms above would have the same denotation. However, operationally the first term terminates whereas the second diverges. This is the reason that we add a new bottom element to it: to differentiate between a *non-terminating term of procedure type* and a *procedure that, if invoked, does not terminate.*

The fact that the meaning of any term is a strict function ($[\![\mathbf{t}]\!]\bot = \bot$) can be explained by the following example: say that we have a term $\mathbf{t}\,;\,\mathbf{i}\mathbf{u}$ where $\mathbf{t}$ is a divergent term. By the operational semantics given in the last chapter, we would assume that the entire term should diverge; however, without the strictness condition it is equivalent to $\mathbf{i}\mathbf{u}$. A similar situation arises when we consider the term $\boldsymbol{X} := \mathbf{t}$ where $\mathbf{t}$ is again divergent. Operationally the entire term should once again diverge, but without the condition in clause 4, this would not be the case. These restrictions enforce the linear, sequential evaluation of terms that is mandated by the operational semantics.

They also allow us to consider only a subcpo of ***State***, i.e., the cpo of states $\sigma$ that do not assign $\bot$ to any location *unless* $\sigma = \bot$. Clearly this cpo of states is closed under the semantic definitions above, and it is easy to show that it is in fact a cpo. (This definition of state follows a comment of de Bakker [dB80, pp.80–81].) We choose not to remove the other members from ***State*** because we will be making use of them when we look at *lazy* schemes of evaluation for ***AC*** in Chapter 5.

Some semantic characterizations of ***AC*** terms can now be given.

**Definition 3.19** (Semantic equivalence). We say that two terms $\mathbf{t}$ and $\mathbf{u}$ are *semantically equivalent* if $[\![\mathbf{t}]\!] = [\![\mathbf{u}]\!]$, that is, if for all states $\sigma$, $[\![\mathbf{t}]\!]\sigma = [\![\mathbf{u}]\!]\sigma$; if this is the case we write $\mathbf{t} \overset{s}{=} \mathbf{u}$.

**Definition 3.20** (Semantic rigidity; $[\![\cdot]\!]$ function). A term $\mathbf{t}$ is *semantically rigid* iff for all $\sigma_1, \sigma_2 \neq \bot$, $[\![\mathbf{t}]\!]\sigma_1 = [\![\mathbf{t}]\!]\sigma_2 \neq \bot$. If $\mathbf{t}$ is known to be semantically rigid, we write $[\![\mathbf{t}]\!] \underset{\text{def}}{=} [\![\mathbf{t}]\!]\sigma$ for some arbitrarily chosen $\sigma \neq \bot$.

Compare Definitions 2.10 and 2.11. The relationship between the operational and denotational versions of these definitions is explored by Corollaries 3.38 and 3.39. Notice that, for a semantically rigid term $\mathbf{t}$, $[\![\mathbf{t}]\!]$ is its sense and $[\![\mathbf{t}]\!]$ is its denotation.

Let us consider again the examples from Chapter 2, this time from a semantic perspective. First, example 1:

$$\begin{aligned}
&[\![\boldsymbol{X} := \mathbf{1};\ \boldsymbol{Y} := \boldsymbol{X};\ \boldsymbol{Y}]\!]\sigma \\
&= [\![\boldsymbol{Y} := \boldsymbol{X};\ \boldsymbol{Y}]\!]([\![\boldsymbol{X} := \mathbf{1}]\!]\sigma) \\
&= [\![\boldsymbol{Y} := \boldsymbol{X};\ \boldsymbol{Y}]\!]\sigma[\boldsymbol{X}/1] \\
&= [\![\boldsymbol{Y}]\!]([\![\boldsymbol{Y} := \boldsymbol{X}]\!]\sigma[\boldsymbol{X}/1]) \\
&= [\![\boldsymbol{Y}]\!]\sigma[\boldsymbol{X}/1][\boldsymbol{Y}/[\![\boldsymbol{X}]\!]\sigma[\boldsymbol{X}/1]] \\
&= [\![\boldsymbol{Y}]\!]\sigma[\boldsymbol{X}/1][\boldsymbol{Y}/1] \\
&= 1
\end{aligned}$$

The only remarkable thing about the above is how unremarkable it is: it demonstrates just how faithful ***AC*** is to standard denotational approaches like [Sto77], [Sch86], [dB80], etc. Next, example 2, which contains the first use of stored intensions.

$$\begin{aligned}
&[\![\boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{X} := \mathbf{1};\ \mathbf{!}\boldsymbol{P}]\!]\sigma \\
&= [\![\boldsymbol{X} := \mathbf{1};\ \mathbf{!}\boldsymbol{P}]\!]([\![\boldsymbol{P} := \mathbf{i}\boldsymbol{X}]\!]\sigma) \\
&= [\![\boldsymbol{X} := \mathbf{1};\ \mathbf{!}\boldsymbol{P}]\!]\sigma[\boldsymbol{P}/[\![\boldsymbol{X}]\!]] \\
&= [\![\mathbf{!}\boldsymbol{P}]\!]([\![\boldsymbol{X} := \mathbf{1}]\!]\sigma[\boldsymbol{P}/[\![\boldsymbol{X}]\!]]) \\
&= [\![\mathbf{!}\boldsymbol{P}]\!]\sigma[\boldsymbol{P}/[\![\boldsymbol{X}]\!]][\boldsymbol{X}/1] \\
&= [\![\boldsymbol{P}]\!]\sigma[\boldsymbol{P}/[\![\boldsymbol{X}]\!]][\boldsymbol{X}/1]\sigma[\boldsymbol{P}/[\![\boldsymbol{X}]\!]][\boldsymbol{X}/1] \\
&= [\![\boldsymbol{X}]\!]\sigma[\boldsymbol{P}/[\![\boldsymbol{X}]\!]][\boldsymbol{X}/1] \ = \ 1
\end{aligned}$$

For example 3, we will, as before, let $\mathbf{p} \equiv \textbf{if } \boldsymbol{X} > \mathbf{1} \textbf{ then } \boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - \mathbf{1};\ !\boldsymbol{P}) \textbf{ else } \mathbf{1}$, and take a state $\sigma[\boldsymbol{X}/4]$ as our starting point. The factorial calculation proceeds as follows:

$$
\begin{aligned}
& [\![\boldsymbol{P} := \mathbf{i}\mathbf{p};\ !\boldsymbol{P}]\!]\sigma[\boldsymbol{X}/4] \\
&= [\![!\boldsymbol{P}]\!]\sigma[\boldsymbol{X}/4][\boldsymbol{P}/[\![\mathbf{p}]\!]] && (*) \\
&= [\![\boldsymbol{P}]\!]\sigma[\boldsymbol{X}/4][\boldsymbol{P}/[\![\mathbf{p}]\!]]\sigma[\boldsymbol{X}/4][\boldsymbol{P}/[\![\mathbf{p}]\!]] \\
&= [\![\textbf{if } \boldsymbol{X} > \mathbf{1} \textbf{ then } \boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - \mathbf{1};\ !\boldsymbol{P}) \textbf{ else } \mathbf{1}]\!]\sigma[\boldsymbol{X}/4][\boldsymbol{P}/[\![\mathbf{p}]\!]] \\
&= \textit{if } 4 > 1 \textit{ then } [\![\boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - \mathbf{1};\ !\boldsymbol{P})]\!]\sigma[\boldsymbol{X}/4][\boldsymbol{P}/[\![\mathbf{p}]\!]] \textit{ else } 1 \\
&= 4 \times [\![!\boldsymbol{P}]\!]\sigma[\boldsymbol{X}/3][\boldsymbol{P}/[\![\mathbf{p}]\!]]; \quad \text{reapplying steps from (*) repeatedly:} \\
&= 4 \times 3 \times [\![!\boldsymbol{P}]\!]\sigma[\boldsymbol{X}/2][\boldsymbol{P}/[\![\mathbf{p}]\!]] \\
&= 4 \times 3 \times 2 \times [\![!\boldsymbol{P}]\!]\sigma[\boldsymbol{X}/1][\boldsymbol{P}/[\![\mathbf{p}]\!]] \\
&= 4 \times 3 \times 2 \times \textit{if } 1 > 1 \textit{ then } [\![\boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - \mathbf{1};\ !\boldsymbol{P})]\!]\sigma[\boldsymbol{X}/1][\boldsymbol{P}/[\![\mathbf{p}]\!]] \textit{ else } 1 \\
&= 4 \times 3 \times 2 \times 1 \ = \ 24
\end{aligned}
$$

## 3.3 Equivalence of Interpretations

In this section, we will demonstrate that the operational and denotational interpretations of $\boldsymbol{AC}$ are in agreement. This is important for two reasons. First, the operational interpretation, which is meant to capture the intuitive computational meaning of terms, contains new results for intension and extension operators. It is important that they be shown equivalent with the denotational semantics for these operators, which are standard [Gal75, Jan86]. Second, the proof of equivalence itself uncovers interesting aspects of $\boldsymbol{AC}$'s operators, and provides insight into the nature of computation in $\boldsymbol{AC}$.

We begin by defining a semantic interpretation of stores as states. First, recall (Definition 2.19) that stores map locations to canonical terms (Definition 2.18).

**Lemma 3.21.** *Canonical terms are semantically rigid.*

*Proof.* By cases on the form of a canonical term $\mathbf{c}$, we show that for any $\sigma_1, \sigma_2 \neq \bot$ it is the case that $[\![\mathbf{c}]\!]\sigma_1 = [\![\mathbf{c}]\!]\sigma_2$.

1. $[\![\mathbf{b}]\!]\sigma_1 = \mathsf{b} = [\![\mathbf{b}]\!]\sigma_2$ for any $\sigma_1, \sigma_2 \neq \bot$.
2. $[\![\mathbf{n}]\!]\sigma_1 = \mathsf{n} = [\![\mathbf{n}]\!]\sigma_2$ for any $\sigma_1, \sigma_2 \neq \bot$.
3. $[\![\mathbf{i}\,\mathbf{t}]\!]\sigma_1 = [\![\mathbf{t}]\!] = [\![\mathbf{i}\,\mathbf{t}]\!]\sigma_2$ for any $\sigma_1, \sigma_2 \neq \bot$. □

Then our semantic interpretation of stores is built by taking the meaning of the contents of each location:

**Definition 3.22** (Semantics of stores)**.** To each store $\mathsf{s}$, we can associate a state $\sigma$ by having, for each $\boldsymbol{X} \in \boldsymbol{Loc}$, $\sigma(\boldsymbol{X}) = [\![\mathsf{s}(\boldsymbol{X})]\!]$. We want to define $[\![\mathsf{s}]\!] = \sigma$ because we would like to think of $\sigma$ as $\mathsf{s}$'s *denotation*; to do so, we set $[\![\mathsf{s}]\!] = \lambda\sigma' \cdot \sigma$.

Next we give a denotational interpretation of operational results.

**Definition 3.23.** Given some $\mathbf{t}$ and $\mathsf{s}$, we define (using Lemma 2.9)

$$(\!|\mathbf{t}|\!)\mathsf{s} = \begin{cases} [\![\mathbf{d}]\!] & \text{if there is a } \mathbf{d} \text{ s.t. } \mathbf{t}, \mathsf{s} \Downarrow \mathbf{d} \\ \bot & \text{otherwise} \end{cases}$$

Using these definitions, we can express the equivalence theorem briefly: the operational and denotational semantics are equivalent iff, for all $\mathsf{s}$,

$$(\!|\mathbf{t}|\!)\mathsf{s} = [\![\mathbf{t}]\!][\![\mathsf{s}]\!]. \tag{3}$$

This is proved in two stages: the '$\sqsubseteq$' direction and the '$\sqsupseteq$' direction. The first is relatively straightforward and is proved by induction on operational derivations. In fact, we can show the stronger result that, whenever $\mathbf{t}, \mathsf{s}$ converges operationally, then (3) holds.

**Lemma 3.24.** *If* $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{d}$, *then* $[\![\mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathbf{d}]\!]$

*Proof.* By induction on derivations.

1. If $\mathbf{t}$ is either $\mathbf{b}$, $\mathbf{n}$ or $\mathbf{i}\mathbf{u}$, then $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{t}$, and $\mathbf{t}$ is canonical and therefore semantically rigid by Lemma 3.21. Thus $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{t}$, and $[\![\mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathbf{t}]\!]$.
2. $\boldsymbol{X}, \mathsf{s} \Downarrow \mathsf{s}(\boldsymbol{X})$. Then $[\![\boldsymbol{X}]\!][\![\mathsf{s}]\!] = [\![\mathsf{s}]\!](\boldsymbol{X}) = [\![\mathsf{s}(\boldsymbol{X})]\!]$ by Definition 3.22.
3. $\mathbf{!t}, \mathsf{s} \Downarrow \mathbf{d}$. Then by definition there is a $\mathbf{u}$ s.t.

$$
\begin{aligned}
& \mathbf{t}, \mathsf{s} \Downarrow \mathbf{i}\mathbf{u} \ \wedge \ \mathbf{u}, \mathsf{s} \Downarrow \mathbf{d} \\
\Rightarrow \ & [\![\mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathbf{i}\mathbf{u}]\!] \ \wedge \ [\![\mathbf{u}]\!][\![\mathsf{s}]\!] = [\![\mathbf{d}]\!] && \text{(by IH twice)} \\
\Rightarrow \ & [\![\mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathbf{u}]\!] \ \wedge \ [\![\mathbf{u}]\!][\![\mathsf{s}]\!] = [\![\mathbf{d}]\!] && \text{(by Def. 3.18-2)} \\
\Rightarrow \ & [\![\mathbf{t}]\!][\![\mathsf{s}]\!][\![\mathsf{s}]\!] = [\![\mathbf{d}]\!] && \text{(by substituting for } [\![\mathbf{u}]\!]) \\
\Leftrightarrow \ & [\![\mathbf{!t}]\!][\![\mathsf{s}]\!] = [\![\mathbf{d}]\!]. && \text{(by Def. 3.18-3)}
\end{aligned}
$$

4. $\boldsymbol{X} := \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/\mathbf{u}]$. Then

$$
\begin{aligned}
& \mathbf{t}, \mathsf{s} \Downarrow \mathbf{u} \\
\Rightarrow \ & [\![\mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathbf{u}]\!] && \text{(by IH)} \\
\Leftrightarrow \ & [\![\boldsymbol{X} := \mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathsf{s}]\!][\boldsymbol{X}/[\![\mathbf{u}]\!]] && \text{(by Def. 3.18-4)} \\
\Leftrightarrow \ & [\![\boldsymbol{X} := \mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathsf{s}[\boldsymbol{X}/\mathbf{u}]]\!] && \text{(by Def. 3.22)}
\end{aligned}
$$

5. $\mathbf{t}\,;\mathbf{u}, \mathsf{s} \Downarrow \mathbf{d}$. Then by definition there is a $\mathsf{s}'$ s.t.

$$
\begin{aligned}
& \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}' \ \wedge \ \mathbf{u}, \mathsf{s}' \Downarrow \mathbf{d} \\
\Rightarrow \ & [\![\mathbf{t}]\!][\![\mathsf{s}]\!] = [\![\mathsf{s}']\!] \ \wedge \ [\![\mathbf{u}]\!][\![\mathsf{s}']\!] = [\![\mathbf{d}]\!] && \text{(by IH twice)} \\
\Leftrightarrow \ & [\![\mathbf{u}]\!]([\![\mathbf{t}]\!][\![\mathsf{s}]\!]) = [\![\mathbf{d}]\!] && \text{(by substituting for } [\![\mathsf{s}']\!]) \\
\Leftrightarrow \ & [\![\mathbf{t}\,;\mathbf{u}]\!][\![\mathsf{s}]\!] = [\![\mathbf{d}]\!]. && \text{(by Def. 3.18-5)}
\end{aligned}
$$

6. **if t then u else v**, $\mathsf{s} \Downarrow \mathbf{d}$. Two cases arise.

*Case 1:* $\mathbf{t}, \mathbf{s} \Downarrow \mathbf{true}$. Then

$$
\begin{array}{lll}
 & \mathbf{t}, \mathbf{s} \Downarrow \mathbf{true} \ \wedge \ \mathbf{u}, \mathbf{s} \Downarrow \mathbf{d} & \\
\Rightarrow & [\![\mathbf{t}]\!][\![\mathbf{s}]\!] = [\![\mathbf{true}]\!] \ \wedge \ [\![\mathbf{u}]\!][\![\mathbf{s}]\!] = [\![\mathbf{d}]\!] & \text{(by IH twice)} \\
\Leftrightarrow & [\![\mathbf{t}]\!][\![\mathbf{s}]\!] = tt \ \wedge \ [\![\mathbf{u}]\!][\![\mathbf{s}]\!] = [\![\mathbf{d}]\!] & \text{(by Def. 3.18-6)} \\
\Leftrightarrow & [\![\mathbf{if\ t\ then\ u\ else\ v}]\!][\![\mathbf{s}]\!] = [\![\mathbf{d}]\!] & \text{(by Def. 3.18-11)}
\end{array}
$$

*Case 2:* $\mathbf{t}, \mathbf{s} \Downarrow \mathbf{false}$. Similar to case 1.

7. Other operators follow a similar pattern. For example, if $\mathbf{t} + \mathbf{u}, \mathbf{s} \Downarrow \mathbf{n}$, then there are $\mathbf{n}_1, \mathbf{n}_2$ s.t.

$$
\begin{array}{lll}
 & \mathbf{t}, \mathbf{s} \Downarrow \mathbf{n}_1 \ \wedge \ \mathbf{u}, \mathbf{s} \Downarrow \mathbf{n}_2 \ \wedge \ n_1 + n_2 = n & \\
\Rightarrow & [\![\mathbf{t}]\!][\![\mathbf{s}]\!] = [\![\mathbf{n}_1]\!] \ \wedge \ [\![\mathbf{u}]\!][\![\mathbf{s}]\!] = [\![\mathbf{n}_2]\!] \ \wedge \ n_1 + n_2 = n & \text{(by IH twice)} \\
\Leftrightarrow & [\![\mathbf{t}]\!][\![\mathbf{s}]\!] = n_1 \ \wedge \ [\![\mathbf{u}]\!][\![\mathbf{s}]\!] = n_2 \ \wedge \ n_1 + n_2 = n & \text{(by Def. 3.18-7)} \\
\Leftrightarrow & [\![\mathbf{t}]\!][\![\mathbf{s}]\!] + [\![\mathbf{u}]\!][\![\mathbf{s}]\!] = n & \\
\Leftrightarrow & [\![\mathbf{t} + \mathbf{u}]\!][\![\mathbf{s}]\!] = n & \text{(by Def. 3.18-9)} \\
\Leftrightarrow & [\![\mathbf{t} + \mathbf{u}]\!][\![\mathbf{s}]\!] = [\![\mathbf{n}]\!] & \text{(by Def. 3.18-7)}
\end{array}
$$

□

Proving the '$\sqsupseteq$' direction of (3) is somewhat more involved, as it requires that we be able to perform induction on the number of "recursive unfoldings" in the meaning of a term. For most languages with recursion, this is no great obstacle because the identifiers that are involved in a mutual recursive system cannot be altered during the course of the recursion. A proof of a similar theorem for such languages would proceed by repeatedly substituting for the identifiers the values that they denote. For an example of such a proof, see [dB80, Theorem 5.22].

We cannot proceed along these lines because *locations involved in a recursive system can be overwritten during the course of the recursion.* For example, given the

examples we have seen so far, we might expect the term

$$\boldsymbol{P} := \mathsf{i}(!\boldsymbol{Q};\ !\boldsymbol{P});\ !\boldsymbol{P} \tag{4}$$

to always diverge. But if $\boldsymbol{Q}$ changes $\boldsymbol{P}$, for example,

$$\boldsymbol{Q} := \mathsf{i}(\boldsymbol{P} := \mathsf{i}\boldsymbol{X}),$$

then (4) is equivalent to the term $\boldsymbol{X}$.

In order to count recursive unfoldings, we use a different method: we augment $\boldsymbol{State}$ with a "virtual location" that holds a counter (as in, a member of the cpo of ordered numbers $\mathbb{C}$) that will be decremented every time an extension operation is performed. This choice of method for bounding recursive unfoldings was chosen primarily for technical reasons; other methods might work but this one seems to allow the cleanest proof of Theorem 3.37. It also has the benefit of being intuitively based on "counting procedure calls".

**Definition 3.25** ($\boldsymbol{State}^{\mathbb{C}}$). Recall the enumeration $(\boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_n)$ of $\boldsymbol{LType}$. The *augmented state signature*, $\Sigma^{\boldsymbol{State}^{\mathbb{C}}}$, is (cf. Definition 3.14)

$$\Sigma^{\boldsymbol{State}^{\mathbb{C}}} = \lambda\mathbb{D} \cdot \left[[\boldsymbol{Loc}^{\boldsymbol{\tau}_1} \to \Sigma^{\boldsymbol{\tau}_1}(\mathbb{D})] \times \cdots \times [\boldsymbol{Loc}^{\boldsymbol{\tau}_n} \to \Sigma^{\boldsymbol{\tau}_n}(\mathbb{D})] \times \mathbb{C}\right]$$

Define $\boldsymbol{State}^{\mathbb{C}}$ to be the resulting domain $\underline{\Sigma}^{\boldsymbol{State}^{\mathbb{C}}}$; again we can unfold and collapse it using $\underline{\sum}_{\rightarrow}^{\boldsymbol{State}^{\mathbb{C}}}$ and $\underline{\sum}_{\leftarrow}^{\boldsymbol{State}^{\mathbb{C}}}$. We range over $\boldsymbol{State}^{\mathbb{C}}$ using the notation $\sigma^c$, where $c \in \mathbb{C}$ indicates the value stored in the virtual counter location. (In our notation, states "wear their counter on their sleeve," so to speak.) This gives a set of states for which the same definition can be given for projection and variant:

**Definition 3.26** (Augmented projection and variant). (Cf. Definition 3.15)

$$\sigma^c(\boldsymbol{X}) \underset{\text{def}}{=} (\underline{\sum}_{\rightarrow}^{\boldsymbol{State}^{\mathbb{C}}}(\sigma^c) \downarrow i)(\boldsymbol{X})$$

$$\sigma^c[\boldsymbol{X}/d] \underset{\text{def}}{=} \underline{\sum}_{\leftarrow}^{\boldsymbol{State}^{\mathbb{C}}}(\underline{\sum}_{\rightarrow}^{\boldsymbol{State}^{\mathbb{C}}}(\sigma^c) \downarrow 1, \ldots, (\underline{\sum}_{\rightarrow}^{\boldsymbol{State}^{\mathbb{C}}}(\sigma^c) \downarrow i)[\boldsymbol{X}/d], \ldots, \underline{\sum}_{\rightarrow}^{\boldsymbol{State}^{\mathbb{C}}}(\sigma^c) \downarrow n), c)$$

Barring the use of subscripts, $\sigma^{c_1}$ and $\sigma^{c_2}$ will be taken to be states that agree on all locations except the counter. For any augmented state $\sigma^c$, we call $c$ its *order*.

To make use of the new aspect of state, we modify the semantics of ***AC*** so that terms behave differently on states of order less than $\omega$. To do so, we first have to modify our semantic domains so that they are based on $\boldsymbol{State}^{\mathbb{C}}$ rather than ***State***. (Recall Definitions 3.12 and 3.13.)

**Definition 3.27.** $\{\!|\tau|\!\}^{\mathbb{C}} = \Sigma^{\tau}(\boldsymbol{State}^{\mathbb{C}})$, and $\boldsymbol{Dom}^{\mathbb{C}} = \bigcup_{\tau \in \boldsymbol{Type}} \{\!|\tau|\!\}^{\mathbb{C}}$.

**Definition 3.28** (Bounded-recursion semantics)**.** The bounded meaning function (we use the same notation as for the non-bounded version; see comments after the definition for an explanation) $[\![\cdot]\!] : \boldsymbol{Term} \to \boldsymbol{Dom}^{\mathbb{C}}$ is defined as follows: first, we have that the meaning function is *strict on c*: $[\![\mathbf{t}]\!]\sigma^0 = \bot$. (This includes the case of the bottom state.) For any $\sigma^c$ where $c \neq 0$,

1. $[\![\boldsymbol{X}]\!]\sigma^c = \sigma^c(\boldsymbol{X})$
2. $[\![\mathbf{i}\,\mathbf{t}]\!]\sigma^c = [\![\mathbf{t}]\!]$
3. $[\![!\mathbf{t}]\!]\sigma^c = [\![\mathbf{t}]\!]\sigma^c\sigma^{c-1}$
4. $[\![\boldsymbol{X} := \mathbf{t}]\!]\sigma^c = \sigma^c[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma^c]$ if $[\![\mathbf{t}]\!]\sigma^c \neq \bot$, $\bot$ otherwise
5. $[\![\mathbf{t};\ \mathbf{u}]\!]\sigma^c = [\![\mathbf{u}]\!]([\![\mathbf{t}]\!]\sigma^c)$

and the supplementary operators

6. $[\![\mathbf{b}]\!]\sigma^c = \mathsf{b}$
7. $[\![\mathbf{n}]\!]\sigma^c = \mathsf{n}$
8. $[\![\mathbf{t} \wedge \mathbf{u}]\!]\sigma^c = [\![\mathbf{t}]\!]\sigma^c \wedge [\![\mathbf{u}]\!]\sigma^c$
9. $[\![\mathbf{t} + \mathbf{u}]\!]\sigma^c = [\![\mathbf{t}]\!]\sigma^c + [\![\mathbf{u}]\!]\sigma^c$
10. $[\![\mathbf{t} < \mathbf{u}]\!]\sigma^c = [\![\mathbf{t}]\!]\sigma^c < [\![\mathbf{u}]\!]\sigma^c$
11. $$[\![\textbf{if t then u else v}]\!]\sigma^c = \begin{cases} [\![\mathbf{u}]\!]\sigma^c & \text{if } [\![\mathbf{t}]\!]\sigma^c = tt \\ [\![\mathbf{v}]\!]\sigma^c & \text{if } [\![\mathbf{t}]\!]\sigma^c = ff \\ \bot & \text{otherwise} \end{cases}$$

The only real difference in the definitions is in clause 3—as promised, the bounded-recursion semantics just counts the number of extensions evaluated in a term's meaning. Consider example 3, evaluated in a state of order $c$. It is a bounded factorial function: it can compute the factorial of any number $n < c$, but will result in $\perp$ if $n \geq c$.

We define the rigid meaning function $[\![\cdot]\!]$ as follows. (Compare Definitions 3.20 and 3.22.)

**Definition 3.29** (Augmented rigid meaning function)**.** For any semantically rigid $\mathbf{t}$, let $[\![\mathbf{t}]\!]$ be $[\![\mathbf{t}]\!]\sigma^c$ for an arbitrary $\sigma^c$ satisfying $c \neq 0$. In the case of stores, define $[\![\mathbf{s}]\!]^c$ to be the $\sigma^c$ that satisfies $\sigma^c(\boldsymbol{X}) = [\![\mathbf{s}(\boldsymbol{X})]\!]$ for all $\boldsymbol{X}$. For uniformity of notations, we will also extend (abuse) this notation slightly by interpreting, for rigid $\mathbf{t}$, $[\![\mathbf{t}]\!]^c$ as $[\![\mathbf{t}]\!]$ because $c$ is irrelevant in this case.

To regain the original behaviour of $\boldsymbol{AC}$, we will simply set the counter to $\omega$ (since $\omega - 1 = \omega$). This is a handy observation since, for every $\sigma \in \boldsymbol{State}$, we now have a chain

$$\sigma^0, \sigma^1, \ldots, \sigma^\omega$$

in $\boldsymbol{State}^{\mathbf{C}}$.

Using these new semantics we can prove interesting properties of terms, like the following:

**Proposition 3.30.** $[\![\boldsymbol{X} := \mathbf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!]\sigma^c = \perp$ *for any* $\sigma^c$.

*Proof.* By induction on $c$, we show that for all $\sigma^c$ s.t. $c < \omega$, $[\![\boldsymbol{X} := \mathbf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!]\sigma^c = \perp$. The basis is clear. For the inductive step, assume that $[\![\boldsymbol{X} := \mathbf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!]\sigma^{c-1} = \perp$.

Then

$$\begin{aligned}
[\![\boldsymbol{X} := \mathsf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!]\sigma^c &= [\![!\boldsymbol{X}]\!]\sigma^c[\boldsymbol{X}/[\![!\boldsymbol{X}]\!]] \\
&= [\![\boldsymbol{X}]\!]\sigma^c[\boldsymbol{X}/[\![!\boldsymbol{X}]\!]]\sigma^{c-1}[\boldsymbol{X}/[\![!\boldsymbol{X}]\!]] \\
&= [\![!\boldsymbol{X}]\!]\sigma^{c-1}[\boldsymbol{X}/[\![!\boldsymbol{X}]\!]] \\
&= [\![\boldsymbol{X} := \mathsf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!]\sigma^{c-1} \\
&= \bot \text{ by IH.}
\end{aligned}$$

To complete the proof we must show that it is true for any $\sigma^\omega$. Note that $\sigma^\omega = \lfloor c \rfloor \sigma^c$. Corollary 3.34 will show that $[\![\mathbf{t}]\!]$ is continuous for any $\mathbf{t}$; it follows that

$$\begin{aligned}
[\![\boldsymbol{X} := \mathsf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!]\sigma^\omega &= [\![\boldsymbol{X} := \mathsf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!](\lfloor c \rfloor \sigma^c) \\
&= \lfloor c \rfloor [\![\boldsymbol{X} := \mathsf{i}!\boldsymbol{X};\ !\boldsymbol{X}]\!]\sigma^c \\
&= \lfloor c \rfloor \bot = \bot.
\end{aligned}$$

□

As mentioned above, when the input state to the meaning function is of order $\omega$, the bounded-recursion semantics behaves the same way as the original semantics. This is clear by simple inspection of the definitions, so we will not prove it explicitly because we would need to develop machinery that is more complex than the thing we are trying to prove. Instead, we just retroactively state that $\sigma \in \boldsymbol{State}$ is really just $\sigma^\omega \in \boldsymbol{State}^{\mathbb{C}}$.

We next show that, for all $\mathbf{t}$, $[\![\mathbf{t}]\!]$ is *continuous*, so that $[\![\mathbf{t}]\!]\sigma^0, [\![\mathbf{t}]\!]\sigma^1, \ldots$ forms a chain whose supremum is $[\![\mathbf{t}]\!]\sigma^\omega$. But before we can do so, we need a couple of useful lemmas.

**Lemma 3.31** (Continuity of state operations).

1. $\sigma_1^{c_1} \sqsubseteq \sigma_2^{c_2} \Rightarrow \sigma_1^{c_1}(\boldsymbol{X}) \sqsubseteq \sigma_2^{c_2}(\boldsymbol{X})$
2. $(\lfloor i \rfloor \sigma_i^{c_i})(\boldsymbol{X}) = \lfloor i \rfloor (\sigma_i^{c_i}(\boldsymbol{X}))$
3. $\sigma_1^{c_1} \sqsubseteq \sigma_2^{c_2} \Rightarrow \sigma_1^{c_1}[\boldsymbol{X}/\mathsf{d}] \sqsubseteq \sigma_2^{c_2}[\boldsymbol{X}/\mathsf{d}]$

4. $(\lfloor i \rfloor \sigma_i^{c_i})[\boldsymbol{X}/\mathsf{d}] = \lfloor i \rfloor \sigma_i^{c_i}[\boldsymbol{X}/\mathsf{d}]$
5. $\mathsf{d}_1 \sqsubseteq \mathsf{d}_2 \Rightarrow \sigma^c[\boldsymbol{X}/\mathsf{d}_1] \sqsubseteq \sigma^c[\boldsymbol{X}/\mathsf{d}_2]$
6. $\sigma^c[\boldsymbol{X}/\lfloor i \rfloor \mathsf{d}_i] = \lfloor i \rfloor \sigma^c[\boldsymbol{X}/\mathsf{d}_i]$

*Proof.* Immediate from Definition 3.25, Definition 3.26 and Theorem 3.11. □

**Lemma 3.32.** *Given a chain of ordered numbers $c_1, c_2, \ldots$, where $c_i \neq 0$ for all $i$,*

$$(\lfloor i \rfloor c_i) - 1 = \lfloor i \rfloor (c_i - 1)$$

*Proof.* In the case where $\lfloor i \rfloor c_i \neq \omega$, the result is obvious. Otherwise, if the chain $c_1, c_2, \ldots$ has no greatest member, then there clearly cannot be a greatest member in the chain $c_1 - 1, c_2 - 1, \ldots$, thus making their shared supremum $\omega$; if the greatest member in the chain $c_1, c_2, \ldots$ is $\omega$, then the greatest member in the chain $c_1 - 1, c_2 - 1, \ldots$ is $\omega - 1 = \omega$. □

With these lemmas in place, we are ready to prove the continuity of $[\![\mathbf{t}]\!]$. We will actually prove a stronger statement: the meaning function only produces elements of $\boldsymbol{Dom}^{\mathbf{C}}$, which (when they are functions) are by definition continuous.

**Theorem 3.33.** $[\![\mathbf{t}^\tau]\!] \in \{\!|\mathbf{s} \to \tau|\!\}$ *for all* $\mathbf{t} \in \boldsymbol{Term}$

*Proof.* By structural induction on $\mathbf{t}$. There are 3 obligations:

(i) $\sigma_1^{c_1} \sqsubseteq \sigma_2^{c_2} \Rightarrow [\![\mathbf{t}]\!]\sigma_1^{c_1} \sqsubseteq [\![\mathbf{t}]\!]\sigma_2^{c_2}$ for all $\sigma_1^{c_1}, \sigma_2^{c_2}$;

(ii) $[\![\mathbf{t}]\!](\lfloor i \rfloor \sigma_i^{c_i}) = \lfloor i \rfloor ([\![\mathbf{t}]\!]\sigma_i^{c_i})$ for all chains $\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots$;

(iii) $[\![\mathbf{t}^\tau]\!]\sigma^c \in \{\!|\tau|\!\}^{\mathbf{C}}$ for all $\sigma^c$.

To simplify the proof, we discard some cases out of hand. For obligation (i), the result is immediate if $c_1 = 0$. For obligation (ii), if all of the $c_i$ are 0 then the result is trivial; we therefore assume that *none* of the $c_i$ are 0 based on the observation that

all of the elements for which $c_i = 0$ are at the start of the chain and can be removed without affecting its least upper bound. For obligation (iii), the result is again trivial if $c = 0$. We now proceed to the proof itself.

1. $\mathbf{t}$ is either $\mathbf{b}$, $\mathbf{n}$ or $\mathbf{i}\mathbf{u}$. Then $\mathbf{t}$ is canonical and therefore, by Lemma 3.21, it is rigid; therefore,

   (i) $[\![\mathbf{t}]\!]\sigma_1^{c_1} = [\![\mathbf{t}]\!] = [\![\mathbf{t}]\!]\sigma_2^{c_2}$

   (ii) $[\![\mathbf{t}]\!](\lfloor_i\rfloor\sigma_i^{c_i}) = [\![\mathbf{t}]\!] = \lfloor_i\rfloor\,[\![\mathbf{t}]\!] = \lfloor_i\rfloor\,[\![\mathbf{t}]\!]\sigma_i^{c_i}$

   (iii) $[\![\mathbf{b}]\!]\sigma^c = \mathsf{b} \in \{\!|\mathbf{B}|\!\}^{\mathbf{C}}$ by Definitions 3.13 and 3.27, and similarly for $\mathbf{n}$. For intension, $[\![\mathbf{i}\mathbf{u}^{\tau}]\!]\sigma^c = [\![\mathbf{u}]\!] \in \{\!|\mathbf{S}\to\tau|\!\}^{\mathbf{C}}$ by IH.

2. $\boldsymbol{X}$.

   (i) $[\![\boldsymbol{X}]\!]\sigma_1^{c_1} = \sigma_1^{c_1}(\boldsymbol{X}) \sqsubseteq \sigma_2^{c_2}(\boldsymbol{X})$ by Lemma 3.31-1
   $= [\![\boldsymbol{X}]\!]\sigma_2^{c_2}$

   (ii) $[\![\boldsymbol{X}]\!](\lfloor_i\rfloor\sigma_i^{c_i}) = (\lfloor_i\rfloor\sigma_i^{c_i})(\boldsymbol{X}) = \lfloor_i\rfloor\,(\sigma_i^{c_i}(\boldsymbol{X}))$ by Lemma 3.31-2
   $= \lfloor_i\rfloor\,[\![\boldsymbol{X}]\!]\sigma_i^{c_i}$

   (iii) $[\![\boldsymbol{X}^{\tau}]\!]\sigma^c = \sigma^c(\boldsymbol{X}) \in \{\!|\tau|\!\}^{\mathbf{C}}$ by Definition 3.26.

3. $!\mathbf{t}$.

   (i) $[\![!\mathbf{t}]\!]\sigma_1^{c_1} = [\![\mathbf{t}]\!]\sigma_1^{c_1}\sigma_1^{c_1-1} \sqsubseteq [\![\mathbf{t}]\!]\sigma_2^{c_2}\sigma_1^{c_1-1}$ by IH (i) and Def. 3.9
   $\sqsubseteq [\![\mathbf{t}]\!]\sigma_2^{c_2}\sigma_2^{c_2-1}$ by IH (iii) (mono. of $[\![\mathbf{t}]\!]\sigma_2^{c_2}$)
   $= [\![!\mathbf{t}]\!]\sigma_2^{c_2}$

(ii) $[\![\mathbf{!t}]\!](\lfloor i \rfloor \sigma_i^{c_i}) = [\![\mathbf{t}]\!](\lfloor i \rfloor \sigma_i^{c_i})(\lfloor j \rfloor \sigma_j^{c_j-1})$ by Lemma 3.32

$= (\lfloor i \rfloor [\![\mathbf{t}]\!]\sigma_i^{c_i})(\lfloor j \rfloor \sigma_j^{c_j-1})$ by IH (ii)

$= \lfloor i \rfloor [\![\mathbf{t}]\!]\sigma_i^{c_i}(\lfloor j \rfloor \sigma_j^{c_j-1})$ by Def. 3.9

$= \lfloor i \rfloor \lfloor j \rfloor [\![\mathbf{t}]\!]\sigma_i^{c_i}\sigma_j^{c_j-1}$ by IH (iii)

$= \lfloor i \rfloor [\![\mathbf{t}]\!]\sigma_i^{c_i}\sigma_i^{c_i-1}$ by Lemma 3.5

$= \lfloor i \rfloor [\![\mathbf{!t}]\!]\sigma_i^{c_i}$

(iii) $[\![\mathbf{!t}^{\mathbf{S}\to\tau}]\!]\sigma^c = [\![\mathbf{t}]\!]\sigma^c\sigma^{c-1}$. $[\![\mathbf{t}]\!]\sigma^c \in \{\!|\mathbf{S}\to\tau|\!\}^{\mathbf{C}}$ by IH (iii), therefore $[\![\mathbf{t}]\!]\sigma^c\sigma^{c-1} \in \{\!|\tau|\!\}^{\mathbf{C}}$ by Definitions 3.13 and 3.27.

4. $\boldsymbol{X} := \mathbf{t}$.

(i) $[\![\boldsymbol{X} := \mathbf{t}]\!]\sigma_1^{c_1} = \sigma_1^{c_1}[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma_1^{c_1}]$

$\sqsubseteq \sigma_1^{c_1}[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma_2^{c_2}]$ by IH (i) & Lemma 3.31-5

$\sqsubseteq \sigma_2^{c_2}[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma_2^{c_2}]$ by Definition 3.26

$= [\![\boldsymbol{X} := \mathbf{t}]\!]\sigma_2^{c_2}$

(ii) $[\![\boldsymbol{X} := \mathbf{t}]\!](\lfloor i \rfloor \sigma_i^{c_i}) = (\lfloor i \rfloor \sigma_i^{c_i})[\boldsymbol{X}/[\![\mathbf{t}]\!](\lfloor j \rfloor \sigma_j^{c_j})]$

$= \lfloor i \rfloor \sigma_i^{c_i}[\boldsymbol{X}/[\![\mathbf{t}]\!](\lfloor j \rfloor \sigma_j^{c_j})]$ by Lemma 3.31-4

$= \lfloor i \rfloor \sigma_i^{c_i}[\boldsymbol{X}/\lfloor j \rfloor [\![\mathbf{t}]\!]\sigma_j^{c_j}]$ by IH (ii)

$= \lfloor i \rfloor \lfloor j \rfloor \sigma_i^{c_i}[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma_j^{c_j}]$ by Lemma 3.31-6

$= \lfloor i \rfloor \sigma_i^{c_i}[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma_i^{c_i}]$ by Lemma 3.5

$= \lfloor i \rfloor [\![\boldsymbol{X} := \mathbf{t}]\!]\sigma_i^{c_i}$

(iii) $[\![\boldsymbol{X} := \mathbf{t}]\!]\sigma^c = \sigma^c[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma^c] \in \{\!|\mathbf{S}|\!\}^{\mathbf{C}}$ because, by IH (iii), $[\![\mathbf{t}]\!]\sigma^c \in \boldsymbol{Dom}^{\mathbf{C}}$; the result follows from Definition 3.26.

5. $\mathbf{t};\mathbf{u}$.

(i) $[\![\mathbf{t};\mathbf{u}]\!]\sigma_1^{c_1} = [\![\mathbf{u}]\!]([\![\mathbf{t}]\!]\sigma_1^{c_1}) \sqsubseteq [\![\mathbf{u}]\!]([\![\mathbf{t}]\!]\sigma_2^{c_2})$ by IH (i), twice
$= [\![\mathbf{t};\mathbf{u}]\!]\sigma_2^{c_2}$

(ii) $[\![\mathbf{t};\mathbf{u}]\!](\lfloor_i\rfloor\sigma_i^{c_i}) = [\![\mathbf{u}]\!]([\![\mathbf{t}]\!](\lfloor_i\rfloor\sigma_i^{c_i})) = [\![\mathbf{u}]\!](\lfloor_i\rfloor[\![\mathbf{t}]\!]\sigma_i^{c_i})$ by IH (ii)
$= \lfloor_i\rfloor[\![\mathbf{u}]\!]([\![\mathbf{t}]\!]\sigma_i^{c_i})$ by IH (ii)
$= \lfloor_i\rfloor[\![\mathbf{t};\mathbf{u}]\!]\sigma_i^{c_i}$

(iii) $[\![\mathbf{t}^{\mathbf{S}};\mathbf{u}^{\tau}]\!]\sigma^c = [\![\mathbf{u}]\!]([\![\mathbf{t}]\!]\sigma^c)$. By IH (iii), $[\![\mathbf{t}]\!]\sigma^c \in \{\!|\mathbf{S}|\!\}^{\mathbf{C}}$, and by IH, $[\![\mathbf{u}]\!] \in \{\!|\mathbf{S}\to\tau|\!\}^{\mathbf{C}}$; therefore, $[\![\mathbf{u}]\!]([\![\mathbf{t}]\!]\sigma^c) \in \{\!|\tau|\!\}^{\mathbf{C}}$.

We skip the supplementary operators because their treatment is standard and they are all known to be continuous (see, e.g., [Sto77]). □

**Corollary 3.34.** $[\![\mathbf{t}]\!]$ *is continuous for all* $\mathbf{t} \in \mathbf{Term}$.

*Proof.* Immediate from the definition of $\{\!|\cdot|\!\}$. □

We can now finish the work that was started in Lemma 3.24 by proving the '$\sqsupseteq$' direction of (3). First we extend the $(\!|\cdot|\!)$ function to work with the augmented domains. (Compare Definition 3.23.)

**Definition 3.35.** Given some $\mathbf{t}$ and $\mathsf{s}$, define

$$(\!|\mathbf{t}|\!)^c\mathsf{s} = \begin{cases} [\![\mathbf{d}]\!]^c & \text{if there is a } \mathbf{d} \text{ s.t. } \mathbf{t},\mathsf{s} \Downarrow \mathbf{d} \\ \bot & \text{otherwise} \end{cases}$$

**Lemma 3.36.** *For all* $\mathbf{t}$, $\mathsf{s}$, *and* $c$, $[\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c \sqsubseteq (\!|\mathbf{t}|\!)^c\mathsf{s}$.

*Proof.* By induction on the pair $(c, |\mathbf{t}|)$ where $c$ is the state order and $|\mathbf{t}|$ is the complexity of $\mathbf{t}$. (The pairs are ordered so that $(c_1, n_1) < (c_2, n_2)$ iff $c_1 < c_2 \vee (c_1 = c_2 \wedge n_1 < n_2)$.) The exact statement of the inductive hypothesis is: if $\mathbf{d} \sqsubseteq [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c$ for

$\mathsf{d} \neq \bot$, then there is a $\mathbf{d}$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{d}$ and $\mathsf{d} \sqsubseteq [\![\mathbf{d}]\!]^c$. (This stronger form of the IH is needed in case 3 below.)

Bases occur when $c = 0$, in which case $[\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c = \bot$ and therefore the statement is vacuous. Other bases arise when we are dealing with atomic terms, or the intension operator which does not require the inductive hypothesis:

1. $\mathbf{t}$ is either $\mathbf{b}$, $\mathbf{n}$ or $\mathbf{i}\mathbf{u}$. Then $[\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c = [\![\mathbf{t}]\!]^c$, and $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{t}$.
2. $\boldsymbol{X}$. $[\![\boldsymbol{X}]\!][\![\mathsf{s}]\!]^c = [\![\mathsf{s}]\!]^c(\boldsymbol{X}) = [\![\mathsf{s}(\boldsymbol{X})]\!]^c$ by Definition 3.29, and $\boldsymbol{X}, \mathsf{s} \Downarrow \mathsf{s}(\boldsymbol{X})$.

The inductive cases proceed as follows.

1. $!\mathbf{t}$.

$$
\begin{array}{lll}
 & \mathsf{d} \sqsubseteq [\![!\mathbf{t}]\!][\![\mathsf{s}]\!]^c & \\
\Leftrightarrow & \exists \mathsf{f} \cdot \mathsf{f} = [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c \wedge \mathsf{d} \sqsubseteq \mathsf{f}[\![\mathsf{s}]\!]^{c-1} & \\
\Rightarrow & \exists \mathsf{f}, \mathbf{u} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathbf{i}\mathbf{u} \wedge \mathsf{f} \sqsubseteq [\![\mathbf{i}\mathbf{u}]\!] \wedge \mathsf{d} \sqsubseteq \mathsf{f}[\![\mathsf{s}]\!]^{c-1} & \text{by IH and Th'm 2.20} \\
\Rightarrow & \exists \mathbf{u} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathbf{i}\mathbf{u} \wedge \mathsf{d} \sqsubseteq [\![\mathbf{i}\mathbf{u}]\!][\![\mathsf{s}]\!]^{c-1} & \text{by Definition 3.9} \\
\Rightarrow & \exists \mathbf{u} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathbf{i}\mathbf{u} \wedge \mathsf{d} \sqsubseteq [\![\mathbf{u}]\!][\![\mathsf{s}]\!]^{c-1} & \\
\Rightarrow & \exists \mathbf{u}, \mathbf{d} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathbf{i}\mathbf{u} \wedge \mathbf{u}, \mathsf{s} \Downarrow \mathbf{d} \wedge \mathsf{d} \sqsubseteq [\![\mathbf{d}]\!]^{c-1} & \text{by IH} \\
\Leftrightarrow & \exists \mathbf{d} \cdot !\mathbf{t}, \mathsf{s} \Downarrow \mathbf{d} \wedge \mathsf{d} \sqsubseteq [\![\mathbf{d}]\!]^{c-1} \sqsubseteq [\![\mathbf{d}]\!]^c &
\end{array}
$$

2. $\boldsymbol{X} := \mathbf{t}$.

$$
\begin{array}{lll}
 & \mathsf{d} \sqsubseteq [\![\boldsymbol{X} := \mathbf{t}]\!][\![\mathsf{s}]\!]^c & \\
\Leftrightarrow & \exists \mathsf{d}' \cdot \mathsf{d}' = [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c \wedge \mathsf{d} \sqsubseteq [\![\mathsf{s}]\!]^c[\boldsymbol{X}/\mathsf{d}'] & \\
\Rightarrow & \exists \mathsf{d}', \mathbf{c} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathbf{c} \wedge \mathsf{d}' \sqsubseteq [\![\mathbf{c}]\!] \wedge \mathsf{d} \sqsubseteq [\![\mathsf{s}]\!]^c[\boldsymbol{X}/\mathsf{d}'] & \text{by IH and Th'm 2.20} \\
\Rightarrow & \exists \mathbf{c} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathbf{c} \wedge \mathsf{d} \sqsubseteq [\![\mathsf{s}]\!]^c[\boldsymbol{X}/[\![\mathbf{c}]\!]] & \text{by Lemma 3.31-5} \\
\Rightarrow & \exists \mathbf{c} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathbf{c} \wedge \mathsf{d} \sqsubseteq [\![\mathsf{s}[\boldsymbol{X}/\mathbf{c}]]\!]^c & \text{by Definition 3.29} \\
\Leftrightarrow & \exists \mathbf{c} \cdot \boldsymbol{X} := \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/\mathbf{c}] \wedge \mathsf{d} \sqsubseteq [\![\mathsf{s}[\boldsymbol{X}/\mathbf{c}]]\!]^c &
\end{array}
$$

3. $\mathbf{t};\mathbf{u}$.

$$
\begin{array}{lll}
& \mathsf{d} \sqsubseteq [\![\mathbf{t};\mathbf{u}]\!][\![\mathsf{s}]\!]^c & \\
\Leftrightarrow & \exists \sigma^{c_1} \cdot \sigma^{c_1} = [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c \wedge \mathsf{d} \sqsubseteq [\![\mathbf{u}]\!]\sigma^{c_1} & \text{by strictness of } [\![\mathbf{u}]\!] \\
\Rightarrow & \exists \sigma^{c_1}, \mathsf{s}' \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}' \wedge \sigma^{c_1} \sqsubseteq [\![\mathsf{s}']\!]^c \wedge \mathsf{d} \sqsubseteq [\![\mathbf{u}]\!]\sigma^{c_1} & \text{by IH and Th'm 2.20} \\
\Rightarrow & \exists \mathsf{s}' \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}' \wedge \mathsf{d} \sqsubseteq [\![\mathbf{u}]\!][\![\mathsf{s}']\!]^c & \text{by mono. of } [\![\mathbf{u}]\!] \\
\Rightarrow & \exists \mathsf{s}', \mathbf{d} \cdot \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}' \wedge \mathbf{u}, \mathsf{s}' \Downarrow \mathbf{d} \wedge \mathsf{d} \sqsubseteq [\![\mathbf{d}]\!]^c & \text{by IH} \\
\Leftrightarrow & \exists \mathbf{d} \cdot \mathbf{t};\mathbf{u}, \mathsf{s} \Downarrow \mathbf{d} \wedge \mathsf{d} \sqsubseteq [\![\mathbf{d}]\!]^c &
\end{array}
$$

We again skip the supplementary operators. We have now proved the proposition for all $\mathbf{t}$ and $\sigma^c$ where $c < \omega$. The result for $\omega$ is obtained by appealing to continuity (Corollary 3.34): for all $c$, we have shown that $[\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c \sqsubseteq (\!|\mathbf{t}|\!)^c\mathsf{s} \sqsubseteq (\!|\mathbf{t}|\!)^\omega\mathsf{s}$, and thus that $\lfloor i \rfloor [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^i \sqsubseteq (\!|\mathbf{t}|\!)^\omega\mathsf{s}$. By continuity, $\lfloor i \rfloor [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^i = [\![\mathbf{t}]\!](\lfloor i \rfloor [\![\mathsf{s}]\!]^i) = [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^\omega$, completing the proof. □

**Theorem 3.37** (Equivalence of interpretations). $(\!|\mathbf{t}|\!)\mathsf{s} = [\![\mathbf{t}]\!][\![\mathsf{s}]\!]$.

*Proof.* By Lemmas 3.24 and 3.36.

Some interesting corollaries of Theorem 3.37 follow. Their proofs are immediate from the theorem.

**Corollary 3.38.** $\mathbf{t}$ *is operationally rigid iff it is semantically rigid. Then we can simply say that* $\mathbf{t}$ *is* rigid.

**Corollary 3.39.** *If* $\mathbf{t} \overset{\circ}{=} \mathbf{u}$, *then* $\mathbf{t} \overset{\mathsf{s}}{=} \mathbf{u}$.

The converse of the above is not true, taking as an example $\mathbf{t} \equiv \mathbf{i2}$ and $\mathbf{u} \equiv \mathbf{i}(1+1)$. However, we can "factor" the operational result through the denotational semantics:

**Corollary 3.40.** *If* $\mathbf{t} \overset{\mathsf{s}}{=} \mathbf{u}$, *then* $(\!|\mathbf{t}|\!)\mathsf{s} = (\!|\mathbf{u}|\!)\mathsf{s}$ *for all* $\mathsf{s}$.

# Chapter 4

# Term Rewriting

In this chapter we explore ***AC*** by examining meaning-preserving transformations of terms. What we will develop is essentially the "calculus" part of Assignment Calculus—a term rewriting system whose rules are meant to capture and exploit its essential equivalences. We do not aim for the most powerful rewriting rules, but rather to enunciate a basic kernel of rules that can at least match the operational semantics.

In developing these rules, we find significant guidance in Janssen's [Jan86] and Hung's [Hun90] work on *state-switcher reductions*, which are a form of rewriting system for (the modal side of) Janssen's ***DIL***. Many of our rules are adapted from or are generalizations of theirs. The goal of their work was to eliminate state-switchers from predicates, resulting in *state-switcher-free preconditions* as meanings of programs. Our goal could roughly be seen, from their perspective, as striving for state-switcher-(assignment-) free *execution results*, though this is an oversimplification: although it is true when we are computing a *value*, in the case where we compute a state, we are really performing assignment-*accumulation* (i.e., computing a *state-term*, see Definition 4.9).

To connect the results of this chapter with those of chapters 2 and 3, we will

provide a proof of equivalence of the rewriting with the operational and denotational semantics, demonstrating that the three are in agreement, and thus tying together all of the interpretations of $\boldsymbol{AC}$.

## 4.1 Rewrite rules and properties

In order to make the rewriting definitions simpler, we adopt the convention that terms are syntactically identical regardless of parenthesization of the sequencing operator; to wit,

**Convention 4.1.** $(\mathbf{t};\mathbf{u});\mathbf{v} \equiv \mathbf{t};(\mathbf{u};\mathbf{v})$

This convention makes it much easier to express rewriting rules that govern the interaction of assignment operators. Its semantic validity (see Theorem 4.6) is easily demonstrated:

$$[\![(\mathbf{t};\mathbf{u});\mathbf{v}]\!] = [\![\mathbf{v}]\!] \circ ([\![\mathbf{u}]\!] \circ [\![\mathbf{t}]\!]) = ([\![\mathbf{v}]\!] \circ [\![\mathbf{u}]\!]) \circ [\![\mathbf{t}]\!] = [\![\mathbf{t};(\mathbf{u};\mathbf{v})]\!].$$

The heart of the rewriting system is the rewriting function $\Rightarrow : \boldsymbol{Term} \to \boldsymbol{Term}$. Recall the definition (2.15) of modally closed terms $\mathbf{mc}$.

**Definition 4.2.** The rewriting function $\Rightarrow$ is given by

| | | | |
|---|---|---|---|
| 1. | $!\,!\mathbf{t}$ | $\Rightarrow$ | $\mathbf{t}$ |
| 2. | $\boldsymbol{X} := \mathbf{mc}_1;\ \mathbf{mc}_2$ | $\Rightarrow$ | $\mathbf{mc}_2$ |
| 3. | $\boldsymbol{X} := \mathbf{t};\ \boldsymbol{X}$ | $\Rightarrow$ | $\mathbf{t}$ |
| 4. | $\boldsymbol{X} := \mathbf{mc};\ \boldsymbol{Y}$ | $\Rightarrow$ | $\boldsymbol{Y}$ |
| 5. | $\boldsymbol{X} := \mathbf{t};\ \boldsymbol{X} := \mathbf{u}$ | $\Rightarrow$ | $\boldsymbol{X} := (\boldsymbol{X} := \mathbf{t};\ \mathbf{u})$ |
| 6. | $\boldsymbol{X} := \mathbf{mc};\ \boldsymbol{Y} := \mathbf{t}$ | $\Rightarrow$ | $\boldsymbol{Y} := (\boldsymbol{X} := \mathbf{mc};\ \mathbf{t});\ \boldsymbol{X} := \mathbf{mc}$ |
| 7. | $\boldsymbol{X} := \mathbf{mc};\ !\mathbf{t}$ | $\Rightarrow$ | $\boldsymbol{X} := \mathbf{mc};\ !(\boldsymbol{X} := \mathbf{mc};\ \mathbf{t})$ |

Along with some rules for the supplementary operators:

| | | | |
|---|---|---|---|
| 8. | $\boldsymbol{X} := \mathbf{t};\ (\mathbf{u} + \mathbf{v})$ | $\Rrightarrow$ | $(\boldsymbol{X} := \mathbf{t};\ \mathbf{u}) + (\boldsymbol{X} := \mathbf{t};\ \mathbf{v})$, etc. |
| 9. | $\mathbf{n}_1 + \mathbf{n}_2$ | $\Rrightarrow$ | $\mathbf{n}$ for the $\mathbf{n}$ s.t. $n_1 + n_2 = n$, etc. |
| 10. | $\mathbf{b}_1 \wedge \mathbf{b}_2$ | $\Rrightarrow$ | $\mathbf{b}$ for the $\mathbf{b}$ s.t. $b_1 \wedge b_2 = b$, etc. |
| 11. | $\mathbf{n}_1 < \mathbf{n}_2$ | $\Rrightarrow$ | $\mathbf{b}$ for the $\mathbf{b}$ s.t. $n_1 < n_2 = b$, etc. |
| 12. | **if true then t else u** | $\Rrightarrow$ | $\mathbf{t}$ |
| 13. | **if false then t else u** | $\Rrightarrow$ | $\mathbf{u}$ |

Rewriting a term $\mathbf{t}$ consists of repeatedly applying $\Rightarrow$ to subterms of $\mathbf{t}$ (zero or more times).

**Definition 4.3.** The rewrite relation $\Rightarrow \subset (\boldsymbol{Term} \times \boldsymbol{Term})$ is defined by: $\mathbf{t} \Rightarrow \mathbf{u}$ iff $\mathbf{u}$ results from applying $\Rrightarrow$ to a subterm of $\mathbf{t}$.

**Definition 4.4.** If $\mathbf{t} \Rightarrow \cdots \Rightarrow \mathbf{u}$ (including if $\mathbf{t} \equiv \mathbf{u}$), then we write $\mathbf{t} \Rrightarrow \mathbf{u}$; that is, $\Rrightarrow$ is the reflexive-transitive closure of $\Rightarrow$.

**Remarks 4.5.**

1. Rule 1 expresses a basic property of Montague's intension and extension operators. In our setting, it embodies the *execution of a procedure.* A natural question is whether $\mathbf{\dot{!}!t}$ can similarly be rewritten to $\mathbf{t}$—the answer is yes, but only if $\mathbf{t}$ is rigid. Such a rule is not necessary to our purposes here, but it is interesting because it is a kind of modal analogue to $\eta$-conversion in $\lambda$-calculus.

2. Rule 2 shows how the assignment statement interacts with modally closed terms. Since terms that are modally closed are rigid, the assignment has no effect *unless its right-hand side does not terminate.* In that case, the entire term would diverge; this is the reason that the right-hand side of the assignment is forced to be modally closed.

3. Rules 3 and 4 show how locations can be "valuated" by assignment statements. The previous comments on right-hand-side termination are still valid (as seen

in rule 4), but in the case of rule 3 the right-hand side is preserved so we do not have to enforce its modal closedness.

4. Rules 5 and 6 pertain to the interaction between two assignments. In the case that both assign to the same location (rule 5), the result of the first assignment is "overwritten" in the state by the result of the second. The only thing that remains of the first assignment is its effect on the right-hand side of the second one. Here, as in rule 3, the right-hand side of the first assignment is not restricted, because its termination properties are preserved.

   As for rule 6, it gives us a way to "push one assignment through another". The argument here is forced to be modally closed, not due to any question of termination, but because it guarantees that the second assignment has no effect on the right-hand side of the first. Note that this rule, when combined with rule 2, immediately provides a way to "swap" two assignment statements.

5. Rule 7 is a very important rule: the *recursion rule*. It may be difficult to see immediately why we identify this rule with recursion; the following special case, which combines the use of rules 7, 3 and 1, illustrates the concept more clearly:

$$X := \mathsf{it};\ !X \quad \Rrightarrow \quad X := \mathsf{it};\ \mathsf{t}$$

   This amounts to simple substitution of a procedure body for its identifier, while "keeping a copy" of the procedure body available for further substitutions.

6. Rule 8 is the general pattern for pushing an assignment statement into any *transparent construct*: the assignment is simply pushed into each of its subterms. Combined with remark 3 above, assignment statements, when pushed into transparent terms, work exactly like a normal substitution operator.

Our first order of business is to show that the rewrite function does not change the meaning of a term.

**Theorem 4.6** (Validity of rewrite rules). $\mathbf{t} \Rrightarrow \mathbf{u} \Longrightarrow [\![\mathbf{t}]\!] = [\![\mathbf{u}]\!]$.

*Proof.* By cases on the rewrite rules, we show that for all $\sigma \neq \bot$, if $\mathbf{t} \Rrightarrow \mathbf{u}$ then $[\![\mathbf{t}]\!]\sigma = [\![\mathbf{u}]\!]\sigma$. All steps are justified directly by the semantic definitions (Definition 3.18) or by basic properties of states.

1. $\mathbf{!it} \Rrightarrow \mathbf{t}$.

$$[\![\mathbf{!it}]\!]\sigma = [\![\mathbf{it}]\!]\sigma\sigma = [\![\mathbf{t}]\!]\sigma$$

2. $\boldsymbol{X} := \mathbf{mc}_1;\ \mathbf{mc}_2 \Rrightarrow \mathbf{mc}_2$.

$$[\![\boldsymbol{X} := \mathbf{mc}_1;\ \mathbf{mc}_2]\!]\sigma = [\![\mathbf{mc}_2]\!]([\![\boldsymbol{X} := \mathbf{mc}_1]\!]\sigma) = [\![\mathbf{mc}_2]\!] = [\![\mathbf{mc}_2]\!]\sigma$$

3. $\boldsymbol{X} := \mathbf{t};\ \boldsymbol{X} \Rrightarrow \mathbf{t}$.

$$\begin{aligned}
[\![\boldsymbol{X} := \mathbf{t};\ \boldsymbol{X}]\!]\sigma &= [\![\boldsymbol{X}]\!]([\![\boldsymbol{X} := \mathbf{t}]\!]\sigma) \\
&= [\![\boldsymbol{X}]\!]\sigma' \text{ where } \sigma' = \begin{cases} \sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma] & \text{if } [\![\mathbf{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
&= \begin{cases} \sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma](\boldsymbol{X}) & \text{if } [\![\mathbf{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
&= [\![\mathbf{t}]\!]\sigma
\end{aligned}$$

4. $\boldsymbol{X} := \mathbf{mc};\ \boldsymbol{Y} \Rrightarrow \boldsymbol{Y}$.

$$\begin{aligned}
[\![\boldsymbol{X} := \mathbf{mc};\ \boldsymbol{Y}]\!]\sigma = [\![\boldsymbol{Y}]\!]([\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma) &= [\![\boldsymbol{Y}]\!](\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]]) \\
&= \sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]](\boldsymbol{Y}) \\
&= \sigma(\boldsymbol{Y}) \\
&= [\![\boldsymbol{Y}]\!]\sigma
\end{aligned}$$

5. $\boldsymbol{X := t;\ X := u} \Rightarrow \boldsymbol{X := (X := t;\ u)}$

$$
\begin{aligned}
& [\![\boldsymbol{X := t;\ X := u}]\!]\sigma \\
= {} & [\![\boldsymbol{X := u}]\!]([\![\boldsymbol{X := t}]\!]\sigma) \\
= {} & \begin{cases} [\![\boldsymbol{X := u}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{t}]\!]\sigma] & \text{if } [\![\boldsymbol{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
= {} & \begin{cases} \sigma[\boldsymbol{X}/[\![\boldsymbol{t}]\!]\sigma][\boldsymbol{X}/[\![\boldsymbol{u}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{t}]\!]\sigma]] & \text{if } [\![\boldsymbol{t}]\!]\sigma \neq \bot \text{ and } [\![\boldsymbol{u}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{t}]\!]\sigma] \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
= {} & \begin{cases} \sigma[\boldsymbol{X}/[\![\boldsymbol{u}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{t}]\!]\sigma]] & \text{if } [\![\boldsymbol{t}]\!]\sigma \neq \bot \text{ and } [\![\boldsymbol{u}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{t}]\!]\sigma] \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
= {} & \begin{cases} \sigma[\boldsymbol{X}/[\![\boldsymbol{u}]\!]([\![\boldsymbol{X := t}]\!]\sigma)] & \text{if } [\![\boldsymbol{u}]\!]([\![\boldsymbol{X := t}]\!]\sigma) \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
= {} & \begin{cases} \sigma[\boldsymbol{X}/[\![\boldsymbol{X := t;\ u}]\!]\sigma] & \text{if } [\![\boldsymbol{X := t;\ u}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
= {} & [\![\boldsymbol{X := (X := t;\ u)}]\!]\sigma
\end{aligned}
$$

6. $\boldsymbol{X := mc;\ Y := t} \Rightarrow \boldsymbol{Y := (X := mc;\ t);\ X := mc}$

$$
\begin{aligned}
& [\![\boldsymbol{X := mc;\ Y := t}]\!]\sigma \\
= {} & [\![\boldsymbol{Y := t}]\!]([\![\boldsymbol{X := mc}]\!]\sigma) \\
= {} & [\![\boldsymbol{Y := t}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{mc}]\!]] \\
= {} & \begin{cases} \sigma[\boldsymbol{X}/[\![\boldsymbol{mc}]\!]][\boldsymbol{Y}/[\![\boldsymbol{t}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{mc}]\!]]] & \text{if } [\![\boldsymbol{t}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{mc}]\!]] \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
= {} & \begin{cases} \sigma[\boldsymbol{Y}/[\![\boldsymbol{t}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{mc}]\!]]][\boldsymbol{X}/[\![\boldsymbol{mc}]\!]] & \text{if } [\![\boldsymbol{t}]\!]\sigma[\boldsymbol{X}/[\![\boldsymbol{mc}]\!]] \neq \bot \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

$$= \begin{cases} [\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma[\boldsymbol{Y}/[\![\mathbf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]]] & \text{if } [\![\mathbf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]] \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$= \begin{cases} [\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma[\boldsymbol{Y}/[\![\mathbf{t}]\!]([\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma)] & \text{if } [\![\mathbf{t}]\!]([\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$= \begin{cases} [\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma[\boldsymbol{Y}/[\![\boldsymbol{X} := \mathbf{mc};\ \mathbf{t}]\!]\sigma] & \text{if } [\![\boldsymbol{X} := \mathbf{mc};\ \mathbf{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$= [\![\boldsymbol{X} := \mathbf{mc}]\!]([\![\boldsymbol{Y} := (\boldsymbol{X} := \mathbf{mc};\ \mathbf{t})]\!])\sigma$$

$$= [\![\boldsymbol{Y} := (\boldsymbol{X} := \mathbf{mc};\ \mathbf{t});\ \boldsymbol{X} := \mathbf{mc}]\!]\sigma$$

7. $\boldsymbol{X} := \mathbf{mc};\ !\mathbf{t} \Rrightarrow \boldsymbol{X} := \mathbf{mc};\ !(\boldsymbol{X} := \mathbf{mc};\ \mathbf{t})$

$$\begin{aligned} [\![\boldsymbol{X} := \mathbf{mc};\ !\mathbf{t}]\!]\sigma &= [\![!\mathbf{t}]\!]([\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma) \\ &= [\![!\mathbf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]] \\ &= [\![\mathbf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]] \\ &= [\![\mathbf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]][\boldsymbol{X}/[\![\mathbf{mc}]\!]]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]] \\ &= [\![\mathbf{t}]\!]([\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]])\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]] \\ &= [\![\boldsymbol{X} := \mathbf{mc};\ \mathbf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]] \\ &= [\![!(\boldsymbol{X} := \mathbf{mc};\ \mathbf{t})]\!]\sigma[\boldsymbol{X}/[\![\mathbf{mc}]\!]] \\ &= [\![!(\boldsymbol{X} := \mathbf{mc};\ \mathbf{t})]\!]([\![\boldsymbol{X} := \mathbf{mc}]\!]\sigma) \\ &= [\![\boldsymbol{X} := \mathbf{mc};\ !(\boldsymbol{X} := \mathbf{mc};\ \mathbf{t})]\!]\sigma \end{aligned}$$

8. $\boldsymbol{X} := \mathbf{t};\ (\mathbf{u} + \mathbf{v}) \Rrightarrow (\boldsymbol{X} := \mathbf{t};\ \mathbf{u}) + (\boldsymbol{X} := \mathbf{t};\ \mathbf{v})$

$$\begin{aligned} &[\![\boldsymbol{X} := \mathbf{t};\ (\mathbf{u} + \mathbf{v})]\!]\sigma \\ &= [\![\mathbf{u} + \mathbf{v}]\!]([\![\boldsymbol{X} := \mathbf{t}]\!]\sigma) \\ &= \begin{cases} [\![\mathbf{u} + \mathbf{v}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma] & \text{if } [\![\mathbf{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

$$= \begin{cases} [\![\mathbf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma] + [\![\mathbf{v}]\!]\sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma] & \text{if } [\![\mathbf{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$= [\![\mathbf{u}]\!]([\![\boldsymbol{X} := \mathbf{t}]\!]\sigma) + [\![\mathbf{v}]\!]([\![\boldsymbol{X} := \mathbf{t}]\!]\sigma)$$

$$= [\![\boldsymbol{X} := \mathbf{t};\ \mathbf{u}]\!]\sigma + [\![\boldsymbol{X} := \mathbf{t};\ \mathbf{v}]\!]\sigma$$

$$= [\![(\boldsymbol{X} := \mathbf{t};\ \mathbf{u}) + (\boldsymbol{X} := \mathbf{t};\ \mathbf{v})]\!]\sigma$$

The other cases for supplementary operators are obvious. The extension of the proof from '$\models\!\!\Rightarrow$' to '$\Rightarrow$' is provided by substitutivity: the compositionality of the meaning function (Definition 3.1) provides that if terms are semantically equivalent (Definition 3.19) they can be freely interchanged. This means that since '$\models\!\!\Rightarrow$' is valid it can be applied to subterms without worry. □

We can gain some valuable insight into how to use the rewriting rules by using them to interpret our three examples from chapters 2 and 3. First, example 1, using only the one-step ( $\Rightarrow$ ) rewrites:

$$\begin{aligned} & \boldsymbol{X} := 1;\ \boldsymbol{Y} := \boldsymbol{X};\ \boldsymbol{Y} \\ \text{(Rule 6)}\ \Rightarrow\ & \boldsymbol{Y} := (\boldsymbol{X} := 1;\ \boldsymbol{X});\ \boldsymbol{X} := 1;\ \boldsymbol{Y} \\ \text{(Rule 4)}\ \Rightarrow\ & \boldsymbol{Y} := (\boldsymbol{X} := 1;\ \boldsymbol{X});\ \boldsymbol{Y} \\ \text{(Rule 3)}\ \Rightarrow\ & \boldsymbol{Y} := 1;\ \boldsymbol{Y} \\ \text{(Rule 3)}\ \Rightarrow\ & 1 \end{aligned}$$

For example 2 we skip some obvious steps by using the $\Rrightarrow$ relation.

$$\begin{aligned} & \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{X} := 1;\ !\boldsymbol{P} \\ \Rrightarrow\ & \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ !\boldsymbol{P} \\ \Rrightarrow\ & \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ !(\boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ \boldsymbol{P}) \\ \Rrightarrow\ & \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathbf{i}\boldsymbol{X};\ !\mathbf{i}\boldsymbol{X} \end{aligned}$$

$$\Rrightarrow\ \boldsymbol{X} := 1;\ \boldsymbol{P} := \mathsf{i}\boldsymbol{X};\ \boldsymbol{X}$$
$$\Rrightarrow\ 1$$

We use example 3 to show that rewriting can also be used to perform partial evaluation of terms, by "unfolding" the factorial. This demonstrates that term rewriting allows more freedom than the operational semantics as it allows us to work with non-rigid terms to gain valuable results. In this way, term rewriting is a sort of intermediate interpretation between the operational and the denotational.

Once again letting $\mathsf{p} \equiv \textbf{if } \boldsymbol{X} > 1 \textbf{ then } \boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - 1;\ !\boldsymbol{P}) \textbf{ else } 1$,

$$\begin{aligned}
&\boldsymbol{P} := \mathsf{i}\mathsf{p};\ !\boldsymbol{P} \\
\Rrightarrow\ &\boldsymbol{P} := \mathsf{i}\mathsf{p};\ (\textbf{if } \boldsymbol{X} > 1 \textbf{ then } \boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - 1;\ !\boldsymbol{P}) \textbf{ else } 1) \\
\Rrightarrow\ &\textbf{if } \boldsymbol{X} > 1 \textbf{ then } \boldsymbol{X} \times (\boldsymbol{X} := \boldsymbol{X} - 1;\ \boldsymbol{P} := \mathsf{i}\mathsf{p};\ !\boldsymbol{P}) \textbf{ else } 1. \\
\Rrightarrow\ &\cdots
\end{aligned}$$

This gives a nice demonstration of the recursion rule (rule 7).

## 4.2 Equivalence of Interpretations

In this section we demonstrate that the rewriting system provided by the $\Rrightarrow$ relation is equivalent to the operational and denotational interpretations. Before stating this theorem, however, we need to address some technicalities.

In order to use the rewriting rules to arrive at the same result as the operational semantics, we need to take into account the *store*. That means that we need a way to take the required information from the store and *actualize* it as a term. For example, take the term $\mathsf{t} \equiv \boldsymbol{X} + \boldsymbol{X}$ and a store $\mathsf{s}$ that maps $\boldsymbol{X}$ to $1$; then, $\mathsf{t}, \mathsf{s} \Downarrow 2$. We can accomplish this in rewriting by *prepending* $\mathsf{t}$ with $\boldsymbol{X} := 1$, which gives $\boldsymbol{X} := 1;\ (\boldsymbol{X} + \boldsymbol{X}) \Rrightarrow 2$ as needed.

To formalize this notion, we use the definition of access set from chapter 2 to identify the locations that we need for the computation. Then, we use the following definition to actualize that part of the store.

**Definition 4.7** (Actualization). Given a finite set of locations $C = \{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\}$, the $C$-actualization of $\mathsf{s}$ is defined as

$$\mathsf{s}\langle C\rangle \underset{\text{def}}{=} \boldsymbol{X}_1 := \mathsf{s}(\boldsymbol{X}_1) \ ; \ \ldots \ ; \ \boldsymbol{X}_n := \mathsf{s}(\boldsymbol{X}_n).$$

A simple lemma expresses the semantic soundness of this idea:

**Lemma 4.8.** *For all* $C, \mathsf{s}$, $[\![\mathsf{s}\langle C\rangle]\!][\![\mathsf{s}]\!] = [\![\mathsf{s}]\!]$.

*Proof.* Obvious. □

Terms of the form returned by $\mathsf{s}\langle C\rangle$ are of enough interest to classify them formally; we call them *state-terms*. Recall the definition of canonical terms $\mathbf{c}$ in Definition 2.18.

**Definition 4.9.** The set of state-terms ***STerm***, ranged over by $\mathbf{s}$, is the set of terms of the form

$$\boldsymbol{X}_1 := \mathsf{c}_1 \ ; \ \ldots \ ; \ \boldsymbol{X}_n := \mathsf{c}_n$$

(for pairwise distinct $\boldsymbol{X}_i$).

We can reorder state-terms at will:

**Lemma 4.10.** *Any state-term* $\boldsymbol{X}_1 := \mathsf{c}_1 \ ; \ \ldots \ ; \ \boldsymbol{X}_n := \mathsf{c}_n$ *can be* reordered, *that is, rewritten using the rewrite rules, to another state-term*

$$\boldsymbol{X}_{i_1} := \mathsf{c}_{i_1} \ ; \ \ldots \ ; \ \boldsymbol{X}_{i_n} := \mathsf{c}_{i_n}$$

*where* $\{i_1, \ldots, i_n\}$ *is any permutation of* $\{1, \ldots, n\}$.

*Proof.* Any two such assignments can be interchanged easily:

$$\begin{aligned} \boldsymbol{X} := \mathsf{c}_1;\ \boldsymbol{Y} := \mathsf{c}_2 &\Rrightarrow \boldsymbol{Y} := (\boldsymbol{X} := \mathsf{c}_1;\ \mathsf{c}_2);\ \boldsymbol{X} := \mathsf{c}_1 && \text{by rule 6} \\ &\Rrightarrow \boldsymbol{Y} := \mathsf{c}_2;\ \boldsymbol{X} := \mathsf{c}_1 && \text{by rule 2.} \end{aligned}$$

The general result can be obtained by repeated swapping of assignments. □

$\mathsf{s}\langle \boldsymbol{acc}(\mathsf{t},\mathsf{s})\rangle$ provides all of the information we need to ensure that the term rewriting rules can achieve the same result as the operational semantics. With this, we are ready to attack the main theorem of the chapter.

**Theorem 4.11** (Operational adequacy)**.**

1. *If* $\mathsf{t},\mathsf{s} \Downarrow \mathsf{c}$, *then* $\mathsf{s}\langle \boldsymbol{acc}(\mathsf{t},\mathsf{s})\rangle ; \mathsf{t} \Rrightarrow \mathsf{c}$;
2. *If* $\mathsf{t},\mathsf{s} \Downarrow \mathsf{s}'$, *then* $\mathsf{s}\langle \boldsymbol{acc}(\mathsf{t},\mathsf{s})\rangle ; \mathsf{t} \Rrightarrow \mathsf{s}'\langle \boldsymbol{acc}(\mathsf{t},\mathsf{s})\rangle$.

*Proof.* By induction on derivations, we will prove the stronger statements that, for *any* $C \supseteq \boldsymbol{acc}(\mathsf{t},\mathsf{s})$,

- If $\mathsf{t},\mathsf{s} \Downarrow \mathsf{c}$, then $\mathsf{s}\langle C\rangle ; \mathsf{t} \Rrightarrow \mathsf{c}$;
- If $\mathsf{t},\mathsf{s} \Downarrow \mathsf{s}'$, then $\mathsf{s}\langle C\rangle ; \mathsf{t} \Rrightarrow \mathsf{s}'\langle C\rangle$.

Cases proceed as follows:

1. $\mathsf{t}$ is either $\mathsf{b}$, $\mathsf{n}$ or $\mathsf{iu}$. The result follows by repeated application of rule 2, removing each of the assignment statements in $\mathsf{s}\langle C\rangle$ in turn, and resulting in $\mathsf{t}$ as required.
2. $\boldsymbol{X}$. In this case, we know, because $\boldsymbol{acc}(\boldsymbol{X},\mathsf{s}) = \{\boldsymbol{X}\}$, that $\boldsymbol{X} \in C$. Reorder (using Lemma 4.10) $\mathsf{s}\langle C\rangle$ so that the assignment $\boldsymbol{X} := \mathsf{s}(\boldsymbol{X})$ occurs first. Now, applying rule 4 repeatedly will remove all assignments to locations other than $\boldsymbol{X}$, until the term is of the form $\boldsymbol{X} := \mathsf{s}(\boldsymbol{X});\ \boldsymbol{X}$; applying rule 3 then gives the required result $\mathsf{s}(\boldsymbol{X})$.

3. $!\mathbf{t}$. If $!\mathbf{t}, \mathsf{s} \Downarrow \mathbf{d}$, then there is a $\mathbf{u}$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow !\mathbf{u}$ and $\mathbf{u}, \mathsf{s} \Downarrow \mathbf{d}$. Let $C \supseteq \mathit{acc}(!\mathbf{t}, \mathsf{s}) = \mathit{acc}(\mathbf{t}, \mathsf{s}) \cup \mathit{acc}(\mathbf{u}, \mathsf{s})$.

$$
\begin{array}{lll}
\mathsf{s}\langle C\rangle\,;!\mathbf{t} & \Rrightarrow\ \mathsf{s}\langle C\rangle\,;!(\mathsf{s}\langle C\rangle\,;\mathbf{t}) & \text{by repeatedly using rule 7 and reordering} \\
& \Rrightarrow\ \mathsf{s}\langle C\rangle\,;!!\mathbf{u} & \text{by IH} \\
& \Rrightarrow\ \mathsf{s}\langle C\rangle\,;\mathbf{u} & \text{by rule 1}
\end{array}
$$

If $\mathbf{d} = \mathsf{s}'$ for some $\mathsf{s}'$, then IH tells us that $\mathsf{s}\langle C\rangle\,;\mathbf{u} \Rrightarrow \mathsf{s}'\langle C\rangle$, as desired. If $\mathbf{d} \equiv \mathbf{c}$, on the other hand, IH provides that $\mathsf{s}\langle C\rangle\,;\mathbf{u} \Rrightarrow \mathbf{c}$ as needed.

4. $\boldsymbol{X} := \mathbf{t}$. If $\boldsymbol{X} := \mathbf{t}, \mathsf{s}$ converges, then there is a $\mathbf{c}$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow \mathbf{c}$, and $\boldsymbol{X} := \mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}[\boldsymbol{X}/\mathbf{c}]$. Let $C \supseteq \mathit{acc}(\boldsymbol{X} := \mathbf{t}, \mathsf{s}) = \mathit{acc}(\mathbf{t}, \mathsf{s}) \cup \{\boldsymbol{X}\}$.

$$
\begin{array}{lll}
& \mathsf{s}\langle C\rangle;\ \boldsymbol{X} := \mathbf{t} & \\
\Rrightarrow & \mathsf{s}\langle C - \{\boldsymbol{X}\}\rangle;\ \boldsymbol{X} := \mathsf{s}(\boldsymbol{X});\ \boldsymbol{X} := \mathbf{t} & \text{by reordering} \\
\Rrightarrow & \mathsf{s}\langle C - \{\boldsymbol{X}\}\rangle;\ \boldsymbol{X} := (\boldsymbol{X} := \mathsf{s}(\boldsymbol{X});\ \mathbf{t}) & \text{by rule 5} \\
\Rrightarrow & \boldsymbol{X} := (\mathsf{s}\langle C - \{\boldsymbol{X}\}\rangle;\ \boldsymbol{X} := \mathsf{s}(\boldsymbol{X});\ \mathbf{t});\ \mathsf{s}\langle C - \{\boldsymbol{X}\}\rangle & \text{by rule 6 repeatedly} \\
\Rrightarrow & \boldsymbol{X} := (\mathsf{s}\langle C\rangle\,;\mathbf{t});\ \mathsf{s}\langle C - \{\boldsymbol{X}\}\rangle & \text{by reordering} \\
\Rrightarrow & \boldsymbol{X} := \mathbf{c};\ \mathsf{s}\langle C - \{\boldsymbol{X}\}\rangle & \text{by IH} \\
\Rrightarrow & \mathsf{s}[\boldsymbol{X}/\mathbf{c}]\langle C\rangle & \text{by reordering}
\end{array}
$$

5. $\mathbf{t}\,;\mathbf{u}$. If $\mathbf{t}\,;\mathbf{u}, \mathsf{s} \Downarrow \mathbf{d}$, then there is a $\mathsf{s}'$ s.t. $\mathbf{t}, \mathsf{s} \Downarrow \mathsf{s}'$ and $\mathbf{u}, \mathsf{s}' \Downarrow \mathbf{d}$. Let $C \supseteq \mathit{acc}(\mathbf{t}\,;\mathbf{u}, \mathsf{s}) = \mathit{acc}(\mathbf{t}, \mathsf{s}) \cup \mathit{acc}(\mathbf{u}, \mathsf{s}')$.

We have that $\mathsf{s}\langle C\rangle\,;\mathbf{t}\,;\mathbf{u} \Rrightarrow \mathsf{s}'\langle C\rangle\,;\mathbf{u}$ by IH. If $\mathbf{d} = \mathsf{s}''$, then by IH, $\mathsf{s}'\langle C\rangle\,;\mathbf{u} \Rrightarrow \mathsf{s}''\langle C\rangle$; if $\mathbf{d} \equiv \mathbf{c}$, then IH gives $\mathsf{s}'\langle C\rangle\,;\mathbf{u} \Rrightarrow \mathbf{c}$, as required.

The cases for the supplementary operators go through easily. □

The above theorem only works in one direction, in that it only shows that the rewriting system is at least as strong as the operational semantics. To complete the

proof of equivalence, we will make use of the equivalence between the operational and denotational interpretations.

One property that we would like the rewriting system to have, but that we can only conjecture for now, is *confluence*:

**Conjecture 4.12** (Confluence). [1] If $\mathbf{t} \Rrightarrow \mathbf{u}$ and $\mathbf{t} \Rrightarrow \mathbf{v}$, then there is a $\mathbf{w}$ s.t. $\mathbf{u} \Rrightarrow \mathbf{w}$ and $\mathbf{v} \Rrightarrow \mathbf{w}$.

The lack of a confluence result for our rewrite rules means that we do not know whether there is a $\mathbf{w}$ s.t. $\mathbf{u} \Rrightarrow \mathbf{w}$ and $\mathbf{v} \Rrightarrow \mathbf{w}$ in the above conjecture. Theorem 4.6 allows us to sidestep this issue by stepping into the semantics: we know, at least, that $\mathbf{u} \overset{\mathbf{s}}{=} \mathbf{v}$ because the rewrite rules preserve meaning. This allows us to define the following *rewrite-semantics* function:

**Definition 4.13** (Rewrite semantics). Given some $\mathbf{t}$ and $\mathbf{s}$, we define

$$\langle\!|\mathbf{t}|\!\rangle\mathbf{s} = \begin{cases} [\![\mathbf{c}]\!] & \text{if there is a } \mathbf{c} \text{ s.t. } \mathbf{s}\langle \boldsymbol{acc}(\mathbf{t},\mathbf{s})\rangle \,;\, \mathbf{t} \Rrightarrow \mathbf{c} \\ [\![\mathbf{s}']\!][\![\mathbf{s}]\!] & \text{if there is an } \mathbf{s}' \text{ s.t. } \mathbf{s}\langle \boldsymbol{acc}(\mathbf{t},\mathbf{s})\rangle \,;\, \mathbf{t} \Rrightarrow \mathbf{s}' \\ \bot & \text{otherwise} \end{cases}$$

**Lemma 4.14.** *For all* $\mathbf{s}$, $\langle\!|\mathbf{t}|\!\rangle\mathbf{s} \sqsubseteq [\![\mathbf{t}]\!][\![\mathbf{s}]\!]$.

*Proof.* Immediate from Definition 4.13, Theorem 4.6, and Lemma 4.8. □

**Lemma 4.15.** *For all* $\mathbf{s}$, $(\!|\mathbf{t}|\!)\mathbf{s} \sqsubseteq \langle\!|\mathbf{t}|\!\rangle\mathbf{s}$.

*Proof.* Immediate from Definition 3.23, Definition 4.13, and Theorem 4.11. □

[1] This conjecture has since been proved by the author.

Putting these results together with Theorem 3.37 gives, for all $\mathbf{t}, \mathbf{s}$,

$$\begin{array}{ccc} & (\!|\mathbf{t}|\!)\mathbf{s} & \\ \nearrow & & \searrow \\ [\![\mathbf{t}]\!][\![\mathbf{s}]\!] & = & (\!|\mathbf{t}|\!)\mathbf{s} \end{array}$$

which proves

**Theorem 4.16** (Equivalence of $\boldsymbol{AC}$ interpretations). *For all* $\mathbf{t}, \mathbf{s}$,

$$\begin{array}{ccc} & (\!|\mathbf{t}|\!)\mathbf{s} & \\ // & & \backslash\backslash \\ [\![\mathbf{t}]\!][\![\mathbf{s}]\!] & = & (\!|\mathbf{t}|\!)\mathbf{s} \end{array}$$

So the three interpretations of $\boldsymbol{AC}$ are in agreement. Note that Lemmas 4.14 and 4.15 make the '$\sqsubseteq$' direction of Theorem 3.37 (the "easy" direction) redundant—the equality in the first diagram above could be replaced by a weaker-than relationship. Therefore Lemma 3.24 is superfluous; we have chosen to include it anyway because of the simplicity and clarity of its proof, and because it makes the equivalence of the operational and denotational semantics "self-contained".

# Chapter 5

# Extensions and Variants of *AC*

In the previous chapters, we have shown that ***AC*** has simple and intuitive syntax and interpretations, and that these interpretations are all in agreement with one another. This provides a solid foundation, but it is important that we also show that ***AC*** is capable of handling more complex imperative features. We believe that one of the most interesting aspects of ***AC*** is that it can be extended and modified in simple ways to include interesting and powerful features.

The goal of this chapter is to show how, by making straightforward adjustments to syntactic and semantic aspects of ***AC***, we can incorporate some of these features. In fact, most of the variants presented below result from simply *removing restrictions and conditions from **AC**'s definitions.* We show, in §§5.1–5.4, how to handle *L-values*, *lazy schemes of evaluation*, *state backtracking*, and *procedure composition.*

Finally, §5.5 offers a (cursory) presentation of a combined imperative/functional language, by adding a typed $\lambda$-calculus to ***AC***. This brings ***AC*** closer to the language that it is based on, Janssen's ***DIL*** [Jan86].

## 5.1 L-values

The first variant that we will consider results from a generalization of the assignment operator to allow arbitrary terms on its left-hand side. This allows us to compute which location is going to be assigned to. This is quite a useful feature as it opens the door to pointers and arrays used as L-values, which are common in practical imperative languages. This work is the re-incorporation into ***AC*** of features that were introduced by Hung into ***DIL*** in [Hun90] and [HZ91]. The novelty is that we also give them an operational interpretation, and that the semantics of pointers is significantly simplified by our use of reflexive state.

To avoid confusion, we will introduce a new symbol for the generalized assignment statement,

$$\mathbf{t}^{\mathbf{s}\to\tau} \leftarrow \mathbf{u}^{\tau},$$

where the usual assignment operator can be interpreted as follows:

$$\boldsymbol{X} := \mathbf{t} \underset{\text{def}}{=} \mathbf{i}\boldsymbol{X} \leftarrow \mathbf{t}. \tag{1}$$

Note that we are making use of the intension operator in a different way here: in order to talk about a location "itself" as opposed to its contents, as discussed in §1.2. The operational interpretation (compare the rule for assignment in Definition 2.8) is

$$\frac{\mathbf{t},\ \mathbf{s} \Downarrow \mathbf{i}\boldsymbol{X} \qquad \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{v}}{\mathbf{t} \leftarrow \mathbf{u},\ \mathbf{s} \Downarrow \mathbf{s}[\boldsymbol{X}/\mathbf{v}]} \tag{2}$$

and its denotation (compare Definition 3.18-4) is

$$[\![\mathbf{t} \leftarrow \mathbf{u}]\!]\sigma = \begin{cases} \bot & \text{if } [\![\mathbf{u}]\!]\sigma = \bot \text{ or } \neg\exists \boldsymbol{X} \cdot [\![\mathbf{t}]\!]\sigma = [\![\boldsymbol{X}]\!] \\ \sigma[\boldsymbol{X}/[\![\mathbf{u}]\!]\sigma] & \text{otherwise, for the } \boldsymbol{X} \text{ s.t. } [\![\mathbf{t}]\!]\sigma = [\![\boldsymbol{X}]\!]. \end{cases} \tag{3}$$

Taking the old rules in Definition 4.2 as they are, but transcribed following (1), We only need to add a single rewrite rule

$$\boldsymbol{X} := \mathbf{mc};\ \mathbf{t} \leftarrow \mathbf{u} \Rrightarrow \boldsymbol{X} := \mathbf{mc};\ (\boldsymbol{X} := \mathbf{mc};\ \mathbf{t}) \leftarrow \mathbf{u}. \tag{4}$$

The operational interpretation is now strictly less powerful than the denotational, as illustrated by the term

$$\mathsf{i}(Y := X;\ Y) \leftarrow \mathbf{t}$$

which cannot be evaluated operationally (because $\mathsf{i}(Y := X;\ Y), \mathsf{s} \Downarrow \mathsf{i}(Y := X;\ Y)$), but whose denotation is the same as $X := \mathbf{t}$. This can be rectified by eliminating terms of this form. We do so by introducing a *subtype* $\mathbf{S} \triangleright \tau$ of $\mathbf{S} \to \tau$ for terms of the form $\mathsf{i}X^{\tau}$.

Now we can give the formal syntax of the L-value extension to $\boldsymbol{AC}$. Add the following clause to Definition 2.5:

$$2'.\quad \mathsf{i}X^{\tau} \in \mathit{Term}^{\mathbf{S}\triangleright\tau}$$

and modify clause 4 in the same definition to

$$4.\quad \mathbf{t}^{\mathbf{S}\triangleright\tau} \leftarrow \mathbf{u}^{\tau} \in \mathit{Term}^{\mathbf{S}}$$

Replace the rule for assignment in Definition 2.8 by (2), and replace rule 4 in Definition 3.18 by the following. (The semantics is a little simpler than that given above because we do not have to deal with the case where the left-hand side does not represent a location.)

$$[\![\mathbf{t}^{\mathbf{S}\triangleright\tau} \leftarrow \mathbf{u}^{\tau}]\!]\sigma = \begin{cases} \bot & \text{if } [\![\mathbf{u}]\!]\sigma = \bot \\ \sigma[X/[\![\mathbf{u}]\!]\sigma] & \text{otherwise, for the } X \text{ s.t. } [\![\mathbf{t}]\!]\sigma = [\![X]\!]. \end{cases}$$

Finally, add the rewriting rule (4) to Definition 4.2. The final alteration is to the definition of access set (Definition 2.12:

$$\text{If } \mathbf{t}, \mathsf{s} \Downarrow \mathsf{i}X, \text{ then } \mathit{acc}(\mathbf{t} \leftarrow \mathbf{u}, \mathsf{s}) = \mathit{acc}(\mathbf{t}, \mathsf{s}) \cup \mathit{acc}(\mathbf{u}, \mathsf{s}) \cup \{X\}.$$

We can now extend the equivalence result (Theorem 3.37) to the new L-value friendly version of $\boldsymbol{AC}$.

**Theorem 5.1.** *Theorem 4.16 holds for* ***AC*** *modified as above.*

*Proof.* We need to modify or extend five results: Theorem 2.20, Theorem 3.33, Lemma 3.36, Theorem 4.6 and Theorem 4.11. The first and second are trivial extensions to the existing proofs. The third is accomplished by modifying inductive case 2 of the proof in a straightforward manner, utilizing the fact that $[\![\boldsymbol{X}]\!] \sqsubseteq [\![\boldsymbol{Y}]\!] \Rightarrow \boldsymbol{X} \equiv \boldsymbol{Y}$, which is easily shown; basically, we just have to add

$$
\begin{array}{lll}
 & \mathsf{d} \sqsubseteq [\![\mathsf{t} \leftarrow \mathsf{u}]\!][\![\mathsf{s}]\!]^c & \\
\Rightarrow & \exists \boldsymbol{X} \cdot [\![\boldsymbol{X}]\!] = [\![\mathsf{t}]\!][\![\mathsf{s}]\!]^c & \\
\Rightarrow & \exists \boldsymbol{X}, \boldsymbol{Y} \cdot \mathsf{t}, \mathsf{s} \Downarrow \mathsf{i}\boldsymbol{Y} \wedge [\![\mathsf{i}\boldsymbol{Y}]\!] \sqsubseteq [\![\boldsymbol{X}]\!] & \text{by IH and Th'm 2.20} \\
\Rightarrow & \exists \boldsymbol{X}, \boldsymbol{Y} \cdot \mathsf{t}, \mathsf{s} \Downarrow \mathsf{i}\boldsymbol{Y} \wedge [\![\boldsymbol{Y}]\!] \sqsubseteq [\![\boldsymbol{X}]\!] & \\
\Rightarrow & \exists \boldsymbol{X} \cdot \mathsf{t}, \mathsf{s} \Downarrow \mathsf{i}\boldsymbol{X} & \text{by the property mentioned above}
\end{array}
$$

to the existing proof of the case. We should add that (3) is extended to a bounded-recursion version in the usual way, that is, it does not affect the state order (see Definition 3.28).

The fourth proof is modified by adding a case for the new rewrite rule (4), which proceeds as follows:

$$
\begin{array}{l}
[\![\boldsymbol{X} := \mathsf{mc};\ \mathsf{t} \leftarrow \mathsf{u}]\!]\sigma \\
= [\![\mathsf{t} \leftarrow \mathsf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]] \\
= \begin{cases} \bot & \text{if } [\![\mathsf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]] = \bot \\ \sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]][\boldsymbol{Y}/[\![\mathsf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]]] & \text{o/w, where } [\![\mathsf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]] = [\![\boldsymbol{Y}]\!] \end{cases} \\
= \begin{cases} \bot & \text{if } [\![\mathsf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]] = \bot \\ \sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]][\boldsymbol{Y}/[\![\mathsf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]]] & \text{o/w, where } [\![\boldsymbol{X} := \mathsf{mc};\ \mathsf{t}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]] = [\![\boldsymbol{Y}]\!] \end{cases} \\
= [\![(\boldsymbol{X} := \mathsf{mc};\ \mathsf{t}) \leftarrow \mathsf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{mc}]\!]] \\
= [\![\boldsymbol{X} := \mathsf{mc};\ (\boldsymbol{X} := \mathsf{mc};\ \mathsf{t}) \leftarrow \mathsf{u}]\!]\sigma
\end{array}
$$

The fifth proof requires a modification to case 4, essentially by prepending it with this: If $\mathbf{t} \leftarrow \mathbf{u}, \mathbf{s}$ converges, then there is a $\boldsymbol{X}$ s.t. $\mathbf{t}, \mathbf{s} \Downarrow \mathbf{i}\boldsymbol{X}$. Let $C \supseteq \boldsymbol{acc}(\mathbf{t} \leftarrow \mathbf{u}, \mathbf{s}) \supseteq \boldsymbol{acc}(\mathbf{t}, \mathbf{s})$.

$$
\begin{array}{rll}
 & \mathbf{s}\langle C\rangle;\ \mathbf{t} \leftarrow \mathbf{u} & \\
\Rrightarrow & \mathbf{s}\langle C\rangle;\ (\mathbf{s}\langle C\rangle\,;\mathbf{t}) \leftarrow \mathbf{u} & \text{by (4) repeatedly + reordering} \\
\Rrightarrow & \mathbf{s}\langle C\rangle;\ \mathbf{i}\boldsymbol{X} \leftarrow \mathbf{u} & \text{by IH} \\
\equiv & \mathbf{s}\langle C\rangle;\ \boldsymbol{X} := \mathbf{u} &
\end{array}
$$

The rest of the case goes just as it is given. □

The addition of L-values to ***AC*** is inspired by the work of Hung and Zucker [HZ91] on introducing L-values and pointers into Janssen's ***DIL***. They introduce a hierarchy within ***State***, where $\boldsymbol{State}_0$ assigns values to base-type locations, $\boldsymbol{State}_1$ assigns base-type locations to pointers, and then $\boldsymbol{State} = \boldsymbol{State}_0 \cup \boldsymbol{State}_1$. (In earlier work ([JvEB77]) Janssen and Van Emde Boas had developed a similar hierarchy of *countable* height; however, their pointers could not be dereferenced on the left-hand side of an assignment statement.) We do not need to introduce this hierarchy because we already allow the storage of intensions. Thus, our contribution is the observation that:

*A pointer is just a special kind of intension.*

## 5.2 Lazy Evaluation

In analogy with the call-by-name or lazy evaluation schemes for $\lambda$-calculus based languages, in this section we will examine some modified evaluation strategies for ***AC***. In the context of imperative reasoning, what could "laziness" entail? An interesting study of this concept was undertaken by J. Launchbury in [Lau93], where he

gives examples of imperative algorithms on lazy data structures and shows that lazy imperative evaluation can be a useful and powerful tool.

In ***AC***, laziness is achieved by eliminating two restrictions that are present in the denotational semantics of ***AC***:

1. $[\![\mathbf{t} := \mathbf{u}]\!]\sigma = \bot$ whenever $[\![\mathbf{u}]\!]\sigma = \bot$,
2. $[\![\mathbf{t}]\!]\sigma = \bot$ whenever $\sigma = \bot$.

Let $\bot$ be some non-terminating term, say $\boldsymbol{X} := \mathbf{i}!\boldsymbol{X};\ !\boldsymbol{X}$. The two conditions above are akin to asking whether these two equivalences hold:

1. $\boldsymbol{X} := \bot;\ \mathbf{t} \stackrel{?}{=} \bot$
2. $\bot;\mathbf{t} \stackrel{?}{=} \bot$

If condition 1 is removed, then equivalence 1 does not hold. We call this *lazy assignment.* If condition 2 is removed, equivalence 2 does not hold. This is called *lazy sequence.* We consider them each in turn.

### 5.2.1 Lazy assignment

In lazy-assignment ***AC***, we allow states $\sigma$ where $\sigma(\boldsymbol{X}) = \bot$ for *some* locations $\boldsymbol{X}$, and we only require the right-hand value of an assignment statement to converge *if it is used.* This generalization of states was mentioned (and discarded) by de Bakker in [dB80, pp.80–81]. It is similar to the lazy $\lambda$-calculus where arguments to functions are not required to terminate before they are passed (substituted) into the applicand [Plo75].

Intuitively, we can interpret this form of laziness as follows: a new execution thread is started each time an assignment statement is encountered. A thread is required to terminate only before its output is needed; any threads with unused output are killed.

The net effect of this change on the denotational semantics is simply the removal of a restriction in the definition of assignment; the new definition is more straightforward,

$$[\![\boldsymbol{X} := \mathbf{t}]\!]\sigma = \sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\sigma],$$

and replaces clause 4 in Definition 3.18. The rewriting is similarly modified by relaxing these clauses in Definition 4.2:

2. $\boldsymbol{X} := \mathbf{t};\ \mathbf{mc} \quad \Rrightarrow \quad \mathbf{mc}$
4. $\boldsymbol{X} := \mathbf{t};\ \boldsymbol{Y} \quad \Rrightarrow \quad \boldsymbol{Y}$

The relaxation in question is that the right-hand sides of the assignment statements of interest are no longer required to be modally closed—see Remark 4.5-2.

The operational interpretation of lazy assignment is more involved. A first attempt at generalizing the rule for assignment to be lazy might be simply to set $\boldsymbol{X} := \mathbf{t}, \mathbf{s} \Downarrow \mathbf{s}[\boldsymbol{X}/\mathbf{t}]$. This does not work because *terms must be evaluated in their original context.* For example,

$$\boldsymbol{X} := \boldsymbol{Y};\ \boldsymbol{Y} := \mathbf{1};\ \boldsymbol{X}$$

should be equivalent to $\boldsymbol{Y}$, *not* to $\mathbf{1}$. So stores must now map locations to *pairs* $(\mathbf{t}, \mathbf{s})$. Notice that now the definition of $\boldsymbol{Store}$ depends on itself! The solution is, however, much easier than it was when we ran into this problem in the denotational semantics (§3.1.1). First of all we allow stores to be *partial functions.* (Let $A \dashrightarrow B$ be the set of partial functions from $A$ to $B$.) Inductively define:

$$\boldsymbol{Store}_n = \bigcup_{\tau \in \boldsymbol{LType}} \boldsymbol{Loc}^\tau \dashrightarrow (\boldsymbol{Term}^\tau \times \bigcup_{i<n} \boldsymbol{Store}_i).$$

$\boldsymbol{Store}_0$ consists of exactly one store $\mathbf{s}_0$ (the nowhere defined function) and for all $n$, $\boldsymbol{Store}_n \subset \boldsymbol{Store}_{n+1}$. We can then let $\boldsymbol{Store} = \bigcup_i \boldsymbol{Store}_i$. Note that if $\mathbf{s} \in \boldsymbol{Store}_i$, then $\mathbf{s}[\boldsymbol{X}/(\mathbf{t}, \mathbf{s})] \in \boldsymbol{Store}_{i+1}$.

The operational semantics (Definition 2.8) is modified as follows:

$$\boldsymbol{X} := \mathbf{t},\ \mathbf{s} \Downarrow \mathbf{s}[\boldsymbol{X}/(\mathbf{t}, \mathbf{s})] \qquad \frac{\mathbf{s}(\boldsymbol{X}) = (\mathbf{t}, \mathbf{s}') \qquad \mathbf{t},\ \mathbf{s}' \Downarrow \mathbf{u}}{\boldsymbol{X},\ \mathbf{s} \Downarrow \mathbf{u}}$$

We do not prove that the interpretations given for ***AC*** with lazy assignment are equivalent, because the new model of store changes some of the earlier definitions significantly. We do, however, believe that such an equivalence result does hold.

### 5.2.2 Lazy sequence

The standard interpretation of the sequence operator '**;**' forces the left-hand side to terminate and return a state before the right-hand side is evaluated in this new state. But what if the right-hand side does not depend on the state at all, i.e., what if it is rigid? The lazy interpretation of '**;**' gives that, if **t** is rigid, then for *any* **u**,

$$\mathbf{u};\mathbf{t} \;=\; \mathbf{t}.$$

As mentioned above, in the denotational semantics the matter is handled simply by removing the restriction that $[\![\mathbf{t}]\!]$ is always strict. To handle lazy sequence operationally, we need to allow partial stores as we did for lazy assignment, but we do not require the store hierarchy. Therefore we define

$$\mathit{Store} = \bigcup_{\tau \in \mathit{LType}} \mathit{Loc}^{\tau} \dashrightarrow \mathit{Can}$$

and again set $\mathsf{s}_0$ to be the nowhere-defined store. The operational semantics does not need to be modified; we just add the rule

$$\frac{\mathbf{u},\ \mathsf{s}_0 \Downarrow \mathbf{d}}{\mathbf{t};\mathbf{u},\ \mathsf{s} \Downarrow \mathbf{d}}$$

to the others in Definition 2.8. Notice that this rule allows that there could be two distinct derivations for the same pair $\mathbf{t}, \mathsf{s}$, which invalidates Lemma 2.9. An example of such a term is $\boldsymbol{X} := \boldsymbol{X};\ \mathbf{0}$, which can be derived either by the usual operational rules from Definition 2.8 or by the rule above.

As far as rewriting goes, we need only add

$$\mathbf{t};\mathbf{mc} \;\Rrightarrow\; \mathbf{mc}$$

to support lazy sequence. As for lazy assignment, we will not prove that the equivalence of interpretations holds for this variant of ***AC***, but we do conjecture that such an equivalence does indeed hold.

To combine lazy assignment and lazy sequence, simply combine the modifications given above of the denotational, operational and rewriting interpretations of ***AC***.

## 5.3 Labelled State Backtracking

State backtracking is the ability to "rewind" the state to some previous configuration. This is useful when modeling *transactions* or other such actions, and is a fundamental component of logic programming languages such as Prolog [TN01]. There has been some work in including state backtracking in practical imperative languages, by K. Apt and colleagues, in the Alma-0 programming language [ABPS97].

We have already discussed the fact that state backtracking is an inherent feature of ***AC*** in §2.5. This is a limited form of state backtracking though: it only allows us to "localize" parts of the computation of a term. A more freewheeling kind of state backtracking can be added to ***AC*** simply by allowing locations to be of type **S**.

When locations of this type are allowed, we gain the ability roughly to *set* and *return to* state backtracking "markers" or labels. This is similar to the way state backtracking is done in Alma-0. Here is an example to show what is possible when we have a location $\boldsymbol{X}$ of type **S**.

$$\begin{aligned} & \boldsymbol{X} := (\boldsymbol{Y} := 1);\ \boldsymbol{Z} := 2;\ \boldsymbol{X} \\ = \ & \boldsymbol{Y} := 1 \end{aligned}$$

In the above example, the use of $\boldsymbol{X}$ at the end of the term backtracks the state to the point before $\boldsymbol{Z}$ was assigned, erasing its effect on the state. It actually backtracks the

state to before $\boldsymbol{X}$ *itself* was assigned, but it also executes the *alternate computation path* of assigning **1** to $\boldsymbol{Y}$.

All denotational and operational definitions are left unchanged. Rewriting is somewhat complicated by the fact that there are no rigid terms of type **S**; this is an interesting topic that merits further attention but must be left to future work. The way out seems, roughly, to be based on the observation that the term

$$\boldsymbol{X}\,;\mathbf{t}$$

depends *only on the contents of* $\boldsymbol{X}$. This complication is the main reason that we have not attempted to extend the equivalence proof to this version of ***AC***.

There might be interesting connections between the state backtracking we describe and the *delimited continuations* of Felleisen et al. [FFDM87]. [ABPS97] gives an implementation method for labelled state backtracking, by using *logs* to keep track of the state changes to reverse when state backtracking is performed.

## 5.4 *AC* with Procedure Composition

One feature that is missing in ***AC*** is the ability to build new procedures from constituent procedures. This weakness can be remedied by replacing the sequence operator '**;**' with a composition operator '**;;**' that takes procedure terms $\mathbf{t}^{\mathbf{S}\to\mathbf{S}}$ and $\mathbf{u}^{\mathbf{S}\to\tau}$ and forms their composition $\mathbf{t};;\mathbf{u}$. Executing this composed procedure is the same as executing **t** and then executing **u**; to wit,

$$!(\mathbf{t}^{\mathbf{S}\to\mathbf{S}};;\mathbf{u}^{\mathbf{S}\to\tau}) \;=\; !\mathbf{t};!\mathbf{u}$$

The denotational interpretation of this operator is straightforward.

$$[\![\mathbf{t};;\mathbf{u}]\!]\sigma \;=\; [\![\mathbf{u}]\!]\sigma \circ [\![\mathbf{t}]\!]\sigma \;=\; \lambda\sigma' \cdot [\![\mathbf{u}]\!]\sigma([\![\mathbf{t}]\!]\sigma\sigma')$$

Compare Definition 3.18-5: we could define sequence in terms of composition as follows

$$\mathbf{t};\mathbf{u} \;=\; !(\mathbf{i}\mathbf{t};;\mathbf{i}\mathbf{u}),$$

but the operational and rewriting rules for composition are simplified considerably if we keep the sequence operator as well. The operational semantics of procedure composition is then

$$\frac{\mathbf{t},\,\mathsf{s} \Downarrow \mathbf{i}\mathbf{v} \qquad \mathbf{u},\,\mathsf{s} \Downarrow \mathbf{i}\mathbf{w}}{\mathbf{t};;\mathbf{u},\,\mathsf{s} \Downarrow \mathbf{i}(\mathbf{v};\mathbf{w})}$$

Rewriting is augmented by the following rules:

$$\mathbf{i}\mathbf{t};;\mathbf{i}\mathbf{u} \Rrightarrow \mathbf{i}(\mathbf{t};\mathbf{u}) \tag{5}$$

$$\boldsymbol{X} := \mathbf{t};\ (\mathbf{u};;\mathbf{v}) \Rrightarrow (\boldsymbol{X} := \mathbf{t};\ \mathbf{u});;(\boldsymbol{X} := \mathbf{t};\ \mathbf{v}) \tag{6}$$

We can extend the proof of equivalence of interpretations easily to ***AC*** with composition. First extend the definition of access set (Definition 2.12 again:

If $\mathbf{t},\mathsf{s} \Downarrow \mathbf{i}\mathbf{v}$ and $\mathbf{u},\mathsf{s} \Downarrow \mathbf{i}\mathbf{w}$, then $\boldsymbol{acc}(\mathbf{t};;\mathbf{u},\mathsf{s}) = \boldsymbol{acc}(\mathbf{t},\mathsf{s}) \cup \boldsymbol{acc}(\mathbf{u},\mathsf{s})$

**Theorem 5.2.** *Theorem 4.16 holds for* ***AC*** *with the composition operator.*

*Proof.* As in §5.1, we need to extend Theorem 2.20, Theorem 3.33, Lemma 3.36, Theorem 4.6 and Theorem 4.11. Skipping as before the first and second due to their simplicity, Lemma 3.36 requires the following case for composition:

$$\begin{aligned}
& \mathsf{d} \sqsubseteq [\![\mathbf{t}^{\mathsf{s}\to\mathsf{s}};;\mathbf{u}^{\mathsf{s}\to\tau}]\!][\![\mathsf{s}]\!]^c \\
\Leftrightarrow\ & \mathsf{d} \sqsubseteq [\![\mathbf{u}]\!][\![\mathsf{s}]\!]^c \circ [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c \\
\Leftrightarrow\ & \exists \mathsf{d}_1,\mathsf{d}_2 \cdot \mathsf{d}_1 = [\![\mathbf{t}]\!][\![\mathsf{s}]\!]^c \wedge \mathsf{d}_2 = [\![\mathbf{u}]\!][\![\mathsf{s}]\!]^c \wedge \mathsf{d} \sqsubseteq \mathsf{d}_2 \circ \mathsf{d}_1 \\
\Rightarrow\ & \exists \mathbf{d}_1,\mathbf{d}_2 \cdot \mathbf{t},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_1 \wedge \mathbf{u},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_2 \wedge \mathsf{d} \sqsubseteq [\![\mathbf{i}\mathbf{d}_2]\!] \circ [\![\mathbf{i}\mathbf{d}_1]\!] \qquad \text{by IH} \\
\Leftrightarrow\ & \exists \mathbf{d}_1,\mathbf{d}_2 \cdot \mathbf{t},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_1 \wedge \mathbf{u},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_2 \wedge \mathsf{d} \sqsubseteq [\![\mathbf{d}_2]\!] \circ [\![\mathbf{d}_1]\!] \\
\Leftrightarrow\ & \exists \mathbf{d}_1,\mathbf{d}_2 \cdot \mathbf{t},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_1 \wedge \mathbf{u},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_2 \wedge \mathsf{d} \sqsubseteq [\![\mathbf{d}_1;\mathbf{d}_2]\!] \\
\Leftrightarrow\ & \exists \mathbf{d}_1,\mathbf{d}_2 \cdot \mathbf{t},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_1 \wedge \mathbf{u},\mathsf{s} \Downarrow \mathbf{i}\mathbf{d}_2 \wedge \mathsf{d} \sqsubseteq [\![\mathbf{i}(\mathbf{d}_1;\mathbf{d}_2)]\!]
\end{aligned}$$

Theorem 4.6 requires the following two cases:

$$[\![\mathsf{i}\mathsf{t};;\mathsf{i}\mathsf{u}]\!]\sigma = [\![\mathsf{i}\mathsf{t}]\!]\sigma \circ [\![\mathsf{i}\mathsf{u}]\!]\sigma = [\![\mathsf{t}]\!] \circ [\![\mathsf{u}]\!] = [\![\mathsf{t};\mathsf{u}]\!] = [\![\mathsf{i}(\mathsf{t};\mathsf{u})]\!]\sigma$$

$$\begin{aligned}
&[\![\boldsymbol{X} := \mathsf{t};\ (\mathsf{u};;\mathsf{v})]\!]\sigma \\
&= [\![\mathsf{u};;\mathsf{v}]\!]([\![\boldsymbol{X} := \mathsf{t}]\!]\sigma) \\
&= \begin{cases} [\![\mathsf{u};;\mathsf{v}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{t}]\!]\sigma] & \text{if } [\![\mathsf{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
&= \begin{cases} [\![\mathsf{v}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{t}]\!]\sigma] \circ [\![\mathsf{u}]\!]\sigma[\boldsymbol{X}/[\![\mathsf{t}]\!]\sigma] & \text{if } [\![\mathsf{t}]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
&= [\![\mathsf{v}]\!]([\![\boldsymbol{X} := \mathsf{t}]\!]\sigma) \circ [\![\mathsf{u}]\!]([\![\boldsymbol{X} := \mathsf{t}]\!]\sigma) \\
&= [\![\boldsymbol{X} := \mathsf{t};\ \mathsf{v}]\!]\sigma \circ [\![\boldsymbol{X} := \mathsf{t};\ \mathsf{u}]\!]\sigma \\
&= [\![(\boldsymbol{X} := \mathsf{t};\ \mathsf{u});;(\boldsymbol{X} := \mathsf{t};\ \mathsf{v})]\!]\sigma
\end{aligned}$$

Theorem 4.11 needs the following addition. If $\mathsf{t};;\mathsf{u},\mathsf{s}$ converges, then there are $\mathsf{v},\mathsf{w}$ s.t. $\mathsf{t},\mathsf{s} \Downarrow \mathsf{i}\mathsf{v}$ and $\mathsf{u},\mathsf{s} \Downarrow \mathsf{i}\mathsf{w}$. Let $C \supseteq \boldsymbol{acc}(\mathsf{t};;\mathsf{u},\mathsf{s}) = \boldsymbol{acc}(\mathsf{t},\mathsf{s}) \cup \boldsymbol{acc}(\mathsf{u},\mathsf{s})$.

$$\begin{aligned}
& \mathsf{s}\langle C\rangle;\ \mathsf{t};;\mathsf{u} && \\
\Rrightarrow\ & (\mathsf{s}\langle C\rangle;\mathsf{t});;(\mathsf{s}\langle C\rangle;\mathsf{u}) && \text{by (6) repeatedly} \\
\Rrightarrow\ & (\mathsf{i}\mathsf{v});;(\mathsf{i}\mathsf{w}) && \text{by IH twice} \\
\Rrightarrow\ & \mathsf{i}(\mathsf{v};\mathsf{u}) && \text{by (5)}
\end{aligned}$$

□

Introducing the composition operator has an interesting consequence: $\boldsymbol{AC}$ becomes a *self-contained Turing-complete language*, in the same sense as the pure (untyped) $\lambda$-calculus is such a language. As in the $\lambda$-calculus, we can encode numerals, arithmetic operations, etc. as $\boldsymbol{AC}$ terms.

We start by encoding numerals and some arithmetic operators. The encodings will depend on a specific location, $\boldsymbol{N}^{\mathsf{s}\to\mathsf{s}}$. The numeral $\mathbf{n}$ will be encoded as the $n$-fold iteration of $\boldsymbol{N}$, i.e., $\overline{\mathbf{n}} = \mathsf{i}(\underbrace{!\boldsymbol{N};\ldots;!\boldsymbol{N}}_{n \text{ times}})$. This method of encoding is inspired

by the Church encoding of numerals in $\lambda$-calculus, where $\mathbf{n}$ is encoded as $\lambda x \cdot \lambda f \cdot \underbrace{f(f(\cdots f}_{n \text{ times}}(x)\cdots))$. Operations are encoded as follows

$$\begin{aligned}
\overline{\mathtt{0}} &\equiv \mathsf{i}(X := X) \\
\overline{\mathtt{succ(n)}} &\equiv \overline{\mathtt{n}};; \mathsf{i}!N \\
\overline{\mathtt{n + m}} &\equiv \overline{\mathtt{n}};;\overline{\mathtt{m}} \\
\overline{\mathtt{n * m}} &\equiv X := \overline{\mathtt{0}};\ Y := \overline{\mathtt{m}};\ N := \mathsf{i}(X := (X;;Y));\ !\overline{\mathtt{n}};\ X
\end{aligned}$$

For multiplication, $N$ is set and repeatedly executed, resulting in the correct answer being stored in $X$, which is then returned. Calculating the predecessor relies on a similar trick:

$$\overline{\mathtt{pred(n)}} \equiv X := \overline{\mathtt{0}};\ N := \mathsf{i}(N := \mathsf{i}(X := (X;;\mathsf{i}!N)));\ !\overline{\mathtt{n}};\ X$$

Next we will encode a ***While***-loop based language, which is the language ***While*** that is defined (and shown to be Turing-complete) in [ZP93]. First there are memory locations $\mathtt{X}, \mathtt{Y}, \ldots$. ***While***-programs $\mathcal{P}$ are generated by:

$$\mathcal{P} ::= \mathtt{X := 0} \mid \mathtt{X := Y} \mid \mathtt{X{+}{+}} \mid \mathtt{X{-}{-}} \mid \mathcal{P}; \mathcal{Q} \mid \mathtt{while\ X > 0\ do}\ \mathcal{P}\ \mathtt{od}$$

The interpretation of a ***While***-program is as follows: given designated input and output variables, $\mathtt{I}$ and $\mathtt{O}$, the (partial) function computed by a program $\mathcal{P}$, $(\mathcal{P}) : \mathbb{N} \dashrightarrow \mathbb{N}$ is computed as follows. $(\mathcal{P})(n)$ is the result of setting $\mathtt{I}$ to $n$, then running $\mathcal{P}$, and then returning the value of $\mathtt{O}$ after the end of the program has been reached; if the end of the program is never reached then $(\mathcal{P})(n)$ diverges.

***While***-programs are encoded as follows: to each variable $\mathtt{X}$ is assigned a location $\overline{\mathtt{X}} \equiv X^{\mathtt{S}\to\mathtt{S}}$ such that $\mathtt{X} \equiv \mathtt{Y} \Leftrightarrow \overline{\mathtt{X}} \equiv \overline{\mathtt{Y}}$. There are also two auxiliary locations $W^{\mathtt{S}\to\mathtt{S}}$ and $V^{\mathtt{S}\to\mathtt{S}}$ that are distinct from those just mentioned. Translations of ***While***-

programs proceed inductively:

$$\overline{\mathtt{X := 0}} \equiv \overline{\mathtt{X}} := \overline{\mathtt{0}}$$

$$\overline{\mathtt{X := Y}} \equiv \overline{\mathtt{X}} := \overline{\mathtt{Y}}$$

$$\overline{\mathtt{X{+}{+}}} \equiv \overline{\mathtt{X}} := \overline{\mathtt{succ\,(X)}}$$

$$\overline{\mathtt{X{-}{-}}} \equiv \overline{\mathtt{X}} := \overline{\mathtt{pred\,(X)}}$$

$$\overline{\mathcal{P}; \mathcal{Q}} \equiv \overline{\mathcal{P}}; \overline{\mathcal{Q}}$$

$$\overline{\mathtt{while\ X>0\ do}\ \mathcal{P}\ \mathtt{od}} \equiv$$

$$W := \mathsf{i}!(V := \mathsf{i}(X := X);\ N := \mathsf{i}(V := \mathsf{i}(\overline{\mathcal{P}}; !W));\ !\overline{\mathtt{X}};\ V);\ !W$$

To complete the translation, we identify the output of the term: the numeral that results from executing $\mathcal{P}$ is given by $\overline{\mathcal{P}}\,;\overline{\mathtt{0}}$. Verification that the translations above provide the expected behaviour are left to the reader.

The ability to perform such a Turing-complete encoding adds weight to our claim that ***AC*** can be seen as an imperative cousin to the $\lambda$-calculus, and strengthens our claim that ***AC*** is a true *imperative reasoning language.*

## 5.5 Combining *AC* with Typed $\lambda$-Calculus

One of the most interesting extensions that we can make to ***AC*** involves adding *functional features* to its core operators. The most natural choice here is a simply typed $\lambda$-calculus, for multiple reasons: first, ***AC*** itself is simply typed; second, Janssen's and Hung's systems contain simply typed $\lambda$-calculus; third, we have already identified $\lambda$-calculus as our favoured functional reasoning language.

Bringing in a typed $\lambda$-calculus allows the treatment of quite interesting features. Hung uses abstraction and application to model *procedure parameters* passed by value, by reference and by name [Hun90]. The ability to abstract over *intensions*, particularly those of type $\mathtt{S} \to \mathtt{S}$, allows for the easy handling of difficult control

constructs like continuations [SW74], [dB80, Chapter 10].

Incorporating $\lambda$-calculus also brings ***AC*** closer to practical programming languages, which invariably contain at least a few functional features. Higher-order imperative languages, such as ALGOL 60 [BBG+97] and derivatives, combine powerful imperative and functional features. Janssen and Hung have already studied aspects of such languages using ***DIL***; it will be interesting to see what new features can be handled using ***AC***.

But the most interesting aspect of the combination is this: because we have endeavoured to make ***AC*** as *pure* an imperative language as possible, the study of its combination with pure functional features could provide a good perspective on the precise points of troublesome interaction between functional and imperative constructs. Given the oft-observed tension between these two paradigms [Bac78, HHJW07], such an orthogonalization might give a new approach to an old problem.

Now to business. We will call our combined language $\boldsymbol{\lambda AC}$, the *$\lambda$/assignment calculus*. The first part of the system that needs to be changed is the set of types, which now allow *any type* as a function domain. (In this section, we leave out the Booleans and numbers for simplicity.)

$$\tau ::= \mathbf{S} \mid \tau \to \tau,$$

We now introduce a new set of syntactic atoms: ***Var***, the set of *variables*, ranged over by $x, y, \ldots$. As with locations, these are individually typed and cannot be of type $\mathbf{S}$.[1] The sets ***Var*** and ***Loc*** are disjoint. ***Loc*** becomes more general as it can now range over other function types, but it is still restricted to a finite number of types. ***Var*** does not require such a restriction.

The syntax of $\boldsymbol{\lambda AC}$ is:

[1] An interesting system arises when we allow variables of type $\mathbf{S}$, such as in [Gal75, §8], but it raises more questions that we do not have time to delve into now. It is certainly an interesting subject for future research.

**Definition 5.3** (Syntax of $\boldsymbol{\lambda AC}$). The typed terms of $\boldsymbol{\lambda AC}$ are given by

| | | | |
|---|---|---|---|
| 1. | $\boldsymbol{X}^{\tau}$ | $\in$ | $\boldsymbol{Term}^{\tau}$ |
| 2. | $\boldsymbol{x}^{\tau}$ | $\in$ | $\boldsymbol{Term}^{\tau}$ |
| 3. | $\mathbf{i}\mathbf{t}^{\tau}$ | $\in$ | $\boldsymbol{Term}^{\mathbf{s}\to\tau}$ |
| 4. | $\mathbf{!t}^{\mathbf{s}\to\tau}$ | $\in$ | $\boldsymbol{Term}^{\tau}$ |
| 5. | $\boldsymbol{X}^{\tau} := \mathbf{t}^{\tau}$ | $\in$ | $\boldsymbol{Term}^{\mathbf{s}}$ |
| 6. | $\mathbf{t}^{\mathbf{s}};\mathbf{u}^{\tau}$ | $\in$ | $\boldsymbol{Term}^{\tau}$ |
| 7. | $\boldsymbol{\lambda x}^{\tau_1}\cdot\mathbf{t}^{\tau_2}$ | $\in$ | $\boldsymbol{Term}^{\tau_1\to\tau_2}$ |
| 8. | $\mathbf{t}^{\tau_1\to\tau_2}\mathbf{u}^{\tau_1}$ | $\in$ | $\boldsymbol{Term}^{\tau_2}$ |

We skip the operational semantics due to the fact that things become more complex in the presence of functional features. The main reason for this is that intension can no longer be seen as representing the pure text of the term it encloses, due to the fact that it only binds the *modal* component of the term and not the *functional* part. Conversely, the usual operation treatment of the $\lambda$-calculus does not work because of the fact that functional binding does not affect the modal component of a term. This explanation may seem a bit vague, but a full treatment must be left to future work.

The next step, then, is then to give the denotational semantics of $\boldsymbol{\lambda AC}$. We do not present in detail the definitions of ***State*** and the semantic domains, as they proceed basically along the same lines of Definitions 3.12–3.15. The only tool that we are lacking is a device for assigning values to variables; we call this a *valuation*, and it performs the same function as a state does for locations. The set of valuations ***Val***, ranged over by $\rho$, consists of (total) functions from ***Var*** into $\mathbb{D}$ respecting types.

The meaning function now requires *two arguments*: a valuation and a state:

$$[\![\cdot]\!] \;:\; \boldsymbol{Term}_{\lambda AC} \to \boldsymbol{Val} \to \boldsymbol{State} \to \mathbb{D}.$$

We will give a semantics that is both imperatively and functionally lazy, mainly because of its elegance:

**Definition 5.4** (Semantics of $\boldsymbol{\lambda AC}$).

1. $[\![\boldsymbol{X}]\!]\rho\sigma = \sigma(\boldsymbol{X})$
2. $[\![\boldsymbol{x}]\!]\rho\sigma = \rho(\boldsymbol{x})$
3. $[\![\mathbf{i}\,\mathbf{t}]\!]\rho\sigma = [\![\mathbf{t}]\!]\rho$
4. $[\![\mathbf{!t}]\!]\rho\sigma = [\![\mathbf{t}]\!]\rho\sigma\sigma$
5. $[\![\boldsymbol{X} := \mathbf{t}]\!]\rho\sigma = \sigma[\boldsymbol{X}/[\![\mathbf{t}]\!]\rho\sigma]$
6. $[\![\mathbf{t}\,;\mathbf{u}]\!]\rho\sigma = [\![\mathbf{u}]\!]\rho\,([\![\mathbf{t}]\!]\rho\sigma)$
7. $[\![\boldsymbol{\lambda x}\cdot\mathbf{t}]\!]\rho\sigma = \lambda x\cdot[\![\mathbf{t}]\!]\rho[\boldsymbol{x}/x]\sigma$
8. $[\![\mathbf{tu}]\!]\rho\sigma = [\![\mathbf{t}]\!]\rho\sigma([\![\mathbf{u}]\!]\rho\sigma)$

There should be nothing surprising about the above definitions, other than the fact that they are strikingly straightforward. They strongly resemble the definitions of Hung [Hun90]; it would be interesting to incorporate our imperative approach to recursion into Hung's system which does not allow state-typed terms.

Finally, we discuss the issue of term rewriting for $\boldsymbol{\lambda AC}$. Without going into too much detail, we can say, first of all, that all of the rules in Chapter 4 hold (modulo the modifications indicated in §5.2 if we are working with lazy imperative features) if we treat abstraction and application as transparent operators, and variables as constants. All we need, then is a way to rewrite $\lambda$-abstraction and application. We use the usual tool, $\beta$-conversion. As Montague points out, however, *$\beta$-conversion in this setting cannot always be carried out.* Because of the presence of modal operators, and thus of referentially opaque contexts in terms, only a restricted form of $\beta$-conversion holds. We follow Janssen's definition:

**Definition 5.5** ($\beta$-conversion for $\boldsymbol{\lambda AC}$).

$$(\boldsymbol{\lambda x}\cdot\mathbf{t})\mathbf{u} \;\Rrightarrow_\beta\; \mathbf{t}[\boldsymbol{x}/\mathbf{u}]$$

*only if* $\mathbf{u}$ is rigid *or* $\boldsymbol{x}$ is not in any opaque context in $\mathbf{t}$. Usually a syntactic approximation is taken to these conditions, i.e., that $\mathbf{u}$ must be *modally closed* and that $\boldsymbol{x}$

does not lie within the scope of an '$\mathbf{i}$' operator, or on the right hand side of a sequence operator.

As noticed in [FW80], this restricted form of $\beta$-conversion is *not confluent*, but the author has discovered some interesting variations of the rule that *are* confluent. These will be presented in future work.

Note that Rule 8 (in Definition 4.2) extends to $\lambda$ and application which are transparent, and that Rule 2 extends to variables which are modally closed.

Here is an example of an $\lambda AC$ term:

$$F := \mathbf{i}(\lambda x \cdot \textbf{if } x \leq 1 \textbf{ then } 1 \textbf{ else } x \times !F(x-1));\ !F$$

Letting $\mathbf{f} \equiv \mathbf{i}(\lambda x \cdot \textbf{if } x \leq 1 \textbf{ then } 1 \textbf{ else } x \times !F(x-1))$, notice that

$$\begin{aligned}
& F := \mathbf{f};\ !F \\
\Rrightarrow\ & \lambda x \cdot \textbf{if } x \leq 1 \\
& \qquad \textbf{then } 1 \\
& \qquad \textbf{else } x \times (F := \mathbf{f};\ !F(x-1)) \\
\Rrightarrow\ & \lambda x \cdot \textbf{if } x \leq 1 \\
& \qquad \textbf{then } 1 \\
& \qquad \textbf{else } x \times \textbf{if } x \leq 2 \\
& \qquad\qquad \textbf{then } 1 \\
& \qquad\qquad \textbf{else } (x-1) \times (F := \mathbf{f};\ !F(x-2)) \\
\Rrightarrow\ & \lambda x \cdot \textbf{if } x \leq 1 \\
& \qquad \textbf{then } 1 \\
& \qquad \textbf{else } x \times \textbf{if } x \leq 2 \\
& \qquad\qquad \textbf{then } 1 \\
& \qquad\qquad \textbf{else } (x-1) \times \textbf{if } x \leq 3 \\
& \qquad\qquad\qquad \textbf{then } 1 \\
& \qquad\qquad\qquad \textbf{else } (x-2) \times (F := \mathbf{f};\ !F(x-3))
\end{aligned}$$

The example uses imperative recursion to "build" a recursive function, erasing all traces of the imperative computation as it goes.

# Chapter 6

# Conclusion

## 6.1 Goals and Contributions

I hope to have convinced the reader, in the last four chapters, that ***AC*** does indeed possess the desirable properties listed in §1.1:

- It is *small*, *elegant* and (hopefully) *intuitive.*

  As discussed in Chapter 2, ***AC***'s set of four basic operators is simple and understandable. Furthermore, §5.4 demonstrates that, with a small change, ***AC***'s meager set of operators can function as a self-contained Turing complete language.

- Its operators (faithfully) represent well-understood, fundamental concepts.

  By taking only assignment, sequence, procedure formation and procedure invocation as basic, we remain close to practical imperative languages. On the other hand, the intension and extension operators of Montague are also by now familiar and well-understood by logicians.

- It is *well grounded*, having operational and denotational semantics that are *equivalent.*

Chapters 2 and 3 show that ***AC*** has relatively uncomplicated operational and denotational semantics, and Theorem 3.37 tells us that they are in fact equivalent.

- We can *rewrite* its terms using simple (provably valid) rules.

  Chapter 4 is devoted to this rewriting system. The rules are not only valid; Theorem 4.16 shows their completeness in that they can be used to arrive at any result that can be computed operationally

- It has interesting properties.

  Section 2.5 gives an example of this: ***AC***'s natural incorporation of (limited) state backtracking allows for the rewriting system of Chapter 4.

- It is easy to modify and extend.

  This is demonstrated in Chapter 5.

I hope that the above points show that Assignment Calculus is a realization of our goal to develop an *imperative reasoning language.* We also contribute the following:

1. A philosophical examination of the concept of *pure imperative reasoning* (Chapter 1).
2. A generalization of the concept of *procedure* using the *intension operator* of Montague, continuing the line of research of Janssen [JvEB77, Jan86] and Hung and Zucker [Hun90, HZ91].
3. An original operational interpretation of Montague's intension and extension operators (Chapter 2).
4. An analysis of the concept of *state backtracking* as a fundamental imperative action (§2.5, §5.3).
5. The use of a *reflexive domain* as a model of *possible worlds.* (Chapter 3)

6. The first systematic treatment of *imperative laziness* (§5.2).

## 6.2 Related Work

The final step in our presentation is to explore related and otherwise relevant work. First note that there has been no other attempt, as far as the author knows, to define a core imperative reasoning language as we have done. Therefore all of the other work that we will examine is only indirectly related to our aims; nevertheless there are certainly interesting connections to be explored.

The first and perhaps most striking language of interest that can be found in the literature is (unfortunately!) nameless; it is defined in the seminal report of Strachey and Scott [SS71, §5]. Here we find a language that has features that closely resemble those of ***AC***: there are operators for referencing and dereferencing, and operators for treating a procedure as an expression and an expression as a procedure. These features seem close in spirit and in meaning to intension and extension, despite some differences. It is quite interesting that their aim seems to be to reduce more complex imperative language features to simple and powerful ones, which is what we have also attempted to do with ***AC***.

Next we consider *Floyd-Hoare-style logics.* These are common when studying imperative languages [dB80, Win93] and there are several modern variations which we will mention. The important point of note here is that such logics are designed to reason *about* imperative languages rather than *within* them (by using, say, a rewriting system as we have done for ***AC***). Programs and assertions about programs are (usually) seen as forming *two distinct syntactic classes.* Such a method of reasoning about programs, though useful, is not in the spirit of our conception of "reasoning language" as developed in Chapter 1. Nevertheless, much interesting work has been done in this area that is relevant to imperative reasoning. It will be interesting to study, in future work, connections with this line of research.

The other main difference between Floyd-Hoare-style systems and ours is that they are *logics*. They deal in preconditions and postconditions—assertions about the state before and after program execution. In this respect they have more in common with Janssen's and Hung's systems which are based on intensional logic and deal with weakest preconditions. Although there are commonalities with our approach of defining a rewriting system, they are too distant to allow for an immediate comparison.

Notable systems of Floyd-Hoare-style logic are Algorithmic Logic [MS87], Dynamic Logic [HKT00], and Separation Logic [Rey02]. A careful comparison of their systems with ours is better suited to future work, because we do not actually define a logic as they do. Of particular interest, however, is the *store model* used in Separation Logic, which facilitates *local reasoning* and reasoning about *pointers*. We would like to compare our treatment of pointers (§5.1) with theirs, and to see if our treatment of pointers as special kinds of procedures could be combined with their approach. The connection is strengthened by recent work [Kri] that treats *higher-order procedures* in Separation Logic; we find, in his use of quotation and evaluation for procedure declaration, a kindred approach to ours and a likely fruitful collaboration in the future.

Insofar as *rewriting systems for imperative languages*, the prime example is the work of Felleisen [FH92]. He adds facilities for handling state and control operators to Plotkin's call-by-value $\lambda$-calculus, which results in a quite elegant system. There could be a good deal of interesting work in comparing our work with Felleisen's; the main problem that we immediately encounter is that his work depends fundamentally on the $\lambda$-calculus, which is precisely what we have tried to *avoid* using in (the usual version of) $\boldsymbol{AC}$. It is impossible to extract a pure imperative (by our definition) part of his system to compare with $\boldsymbol{AC}$. The best starting point for a comparison would then be the language $\boldsymbol{\lambda AC}$ of §5.5. We intend to undertake such an analysis in forthcoming work.

Another example of a rewriting system for imperative languages is the language ***iRho*** of Liquori and Serpette [LS08]. The same essential differences mentioned above also apply here: because it is an extension of the Rewriting Calculus [CKL02] which is a *functional* calculus, ***iRho*** is really a functional language enriched with imperative features. It seems impossible (to the author) to extract a workable, pure imperative system from their work.

Another line of research that is related to ours is based on *abstract machines.* We identify in particular Random Access Stored Program machines and Pointer Machines [vEB90]. These are different in spirit from our work: their intention is to provide a model of computation that is adequate for studying computability and complexity of algorithms that use stored procedures and pointers. We mention them mainly because they indicate a *need* for modeling features like procedures and pointers. The main deficiency, in our view, of machine-based models such as these is that they are "too concrete": they are mired in implementation details and are therefore difficult to use as *reasoning tools.*

## 6.3 Future Work

We now discuss interesting avenues of further research.

1. Exploring, expanding, and improving the rewriting system of Chapter 4. As we stated there, the goal was to arrive at a small set of rules that was sufficient to achieve our equivalence proof; however, it would be better to develop more powerful rules that might be more intuitive in terms of real-world use. There is also the matter that we have not proved the confluence conjecture for our rules—certainly this is a question that we would like to answer.[1]
2. Clarifying the philosophical issues related to imperative reasoning; looking at

[1]The conjecture has since been proved by the author.

programming languages, both imperative and functional, in order to solidify and explicate our position on referential opacity vs. transparency.

3. Looking at other basic constructs for ***AC***, such as, say, stacks or arrays (as in Hung [Hun90]) as well as unordered memory locations.
4. Examining more carefully the concept of *state backtracking* in ***AC***. As mentioned in §2.5, we believe that state backtracking is a fundamental part of imperative reasoning; therefore, we would like to provide an improved analysis of what it is and how it takes part in and affects imperative reasoning.
5. Developing the extensions to ***AC*** that are presented in Chapter 5, particularly the combination with the $\lambda$-calculus, as we believe that this direction can be particularly fruitful because of its immediate relevance to the foundations of hybrid imperative/functional programming languages.
6. Exploring connections with Separation Logic, particularly its interesting store model, and with Felleisen's work as mentioned above.

The aim of this dissertation was to provide an analysis of imperative reasoning. We have done this on multiple levels: from a philosophical dissection of the concept of imperative reasoning in Chapter 1, to an actual language ***AC*** and suggested extensions, to mathematical, operational and rewriting-based meanings, to a real implementation in Appendix A. We believe that this broad approach leaves ***AC*** well-prepared for further investigations, and we hope that it will stimulate future work in what we consider to be an exciting new approach to an old paradigm.

# Appendix A

# Implementation

This appendix is devoted to a presentation of an implementation of the rewriting system of ***AC***. All of the rules given in Chapter 4, along with all of those for extensions mentioned in Chapter 5, are included. The program is written in the Haskell programming language [P+03], and is freely available on the author's website at (`www.cas.mcmaster.ca/~bendermm`). Up-to-date instructions are also available there.

## A.1 Introduction and Usage Instructions

In order to allow users to interactively work through examples given in this thesis, we have developed an implementation of the rewriting system for ***AC*** and its extensions. The program works by allowing the user to enter ***AC*** terms in an ASCII version of ***AC*** syntax, and then to interact with these terms by selectively rewriting parts of those terms in whatever way that they wish. Users can also load terms from a text file; several examples are included (see §A.5).

The program can be broken down into three parts:

1. Parsing and printing: this part of the program parses ASCII representations of ***AC*** terms, and prints formatted (indented) program text to the screen;
2. Properties and manipulation: this component deals with the computation of properties of ***AC*** terms and the application of the rewrite rules to parts of terms.
3. Interaction: this component of the program allows the user to move freely about a term, select parts to rewrite, and apply rewriting.

We give descriptions and instructions for each part, but we have only included code for part 2. The reason for this is that most of the code for parts 1 and 3 is not directly relevant to the implementation of terms and rewrite rules presented in the thesis.

The program was designed to allow the user to interact with ***AC*** terms using a Haskell interpreter such as GHCI (`www.haskell.org/ghc/`), and requires an `xterm`-compatible terminal. Here is a simple example to get started:

1. Start by downloading and unpacking the Haskell program into a directory of your choice. Open a terminal in this directory.
2. Start the Haskell interpreter; if you are using GHCI, the command is

   ```
   ghci AC
   ```

   If another interpreter is being used, please consult its documentation for usage instructions, or check the author's website for directions.
3. Load a term by entering, say,

   ```
   load "simple1"
   ```

   You should see the listing of the example appear. Since it is now the last returned item, it can be referred to in the Haskell interpreter as "`it`".

4. Begin interacting with the example by entering

```
act it
```

   which will clear the screen, draw the term in the upper-left corner, and allow you to move the cursor around using the arrow keys. You can select an assignment statement by positioning the cursor over the ':=' and pressing the space bar; deselecting is done in the same way. To rewrite the selected assignment(s), press Enter or the 'r' key.

5. Arithmetic operators can also be selected in a similar way. Notice that rewriting these operators will not have any effect until both sides are numbers.

6. To highlight all of the selectable points at once, press the 'a' key. (Make sure that Caps Lock is not on, because typing 'A' will have the opposite effect and deselect everything.) To maneuver more quickly through terms, you can hold Shift while pressing the arrow keys, or use the Page Up and Page Down keys. Alternatively, you can cycle through all of the selectable points in the term by pressing 'n' and 'p'. A full list of all of the control keys is provided in §A.4.

7. When you have finished rewriting the term, you can exit interactive mode by pressing 'q'.

Terms can be assigned to variables by using the `let` operator, as in: `let x = load "simple1"`. For more on the interactive command prompt and what can be done there, please refer to the relevant documentation. Instructions for GHCI can be found online at `www.haskell.org/ghc/`.

## A.2 Parsing and Printing

The syntax that is used by the program is by necessity different than that given in the thesis, not only because of the lack of a 'ı' key on most keyboards. Nevertheless, we

have tried to keep approximations as close as possible. Here is a table of ***AC*** operators and their corresponding ASCII equivalents, in decreasing order of precedence:

| | ***AC* syntax** | **ASCII syntax** | **Selectable** |
|---|---|---|---|
| Booleans | **b** | `True, False` | no |
| Numerals | **n** | `0, 1, ...` | no |
| Locations | $X$ | Identifier starting with capital | no |
| Variables | $x$ | Identifier starting with lowercase | no |
| Parentheses | () | `()` | no |
| Intension | **i** | `^` | no |
| Extension | **!** | `!` | yes |
| Application | juxtaposition | juxtaposition | no |
| Arithmetic, etc. | +, −, etc. | +, -, etc. | no |
| Composition | ;; | `;;` | yes |
| Assignment | := | `:=` | yes |
| Lazy assignment | := | `:=~` | yes |
| L-Value assignment | ← | `<-` | yes |
| Sequence | ; | `;` | no |
| Lazy sequence | ; | `;~` | yes |
| Abstraction | $\lambda x \cdot$ | `\x.` | yes |
| Conditional | **if then else** | `if then else` | yes |

To parse a term from the command prompt, use the `parse` function, for example,

```
parse "\x . X:=x; Y:=^X".
```

When a term is being displayed, whether at the command prompt or in interactive mode, it is indented and spaced for the user's convenience and ease of readability.

The program does not currently include an implementation of types. This allows the user to experiment with terms that do not correspond to well typed ***AC*** terms; if however a term does correspond to a typed ***AC*** term, the implementation will behave as expected.

## A.3 Properties and Manipulation

We begin by giving the Haskell definitions that implement the set of operators listed above. First, we employ these useful type synonyms.

```
type Var    = String
type Loc    = String
type Active = Bool
type Lazy   = Bool
```

*Active* is used to indicate whether a particular operator is selected or not; *Lazy* will designate an assignment or sequence operator to be *lazy* as discussed in §5.2. Terms are generated by

```
data Term = TInt Int | TBool Bool | TVar Var | TLoc Loc
          | TIn Term | TEx Active Term
          | TAss Active Lazy Term Term | TSeq Active Lazy Term Term
          | TLam Active Var Term | TApp Term Term
          | TCond Active Term Term Term
          | TOp Active String Term Term
            deriving Eq
```

Next, we collect various important properties and basic operations on terms. First, the `freeVars` function returns a list of all of the free variables in a term. The `closed` function tells us whether or not a term is (functionally) closed.

`sub` performs substitution of a term for a variable. We have been careful to provide a correct substitution mechanism that chooses a fresh variable name (when needed) in order to avoid the variable capture problem. The `newVar` function implements this.

```
freeVars :: Term → [Var]
freeVars (TVar x)        = [x]
```

```
freeVars (TAss _ _ t u)  = freeVars t `U` freeVars u
freeVars (TSeq _ _ t u)  = freeVars t `U` freeVars u
freeVars (TLam _ x t)    = delete x $ freeVars t
freeVars (TApp t u)      = freeVars t `U` freeVars u
freeVars (TCond _ t u v) = freeVars t `U` freeVars u `U` freeVars v
freeVars (TOp _ _ t u)   = freeVars t `U` freeVars u
freeVars _               = []

closed :: Term → Bool
closed = null·freeVars

sub :: Term → Var → Term → Term
sub a s = sub'
 where
  sub' t@(TVar x)      = if x ≡ s then a else t
  sub' (TIn t)         = TIn $ sub' t
  sub' (TEx b t)       = TEx b $ sub' t
  sub' (TAss b l t u)  = TAss b l (sub' t) (sub' u)
  sub' (TSeq b l t u)  = TSeq b l (sub' t) (sub' u)
  sub' t@(TLam b x u)  | x ≡ s               = t
                       | x `elem` freeVars a
                         = let z = newVar (x++"'") (freeVars a)
                            in TLam b z $ sub' $ sub (TVar z) x u
                       | otherwise           = TLam b x $ sub' u
  sub' (TApp t u)      = TApp (sub' t) (sub' u)
  sub' (TCond b t u v) = TCond b (sub' t) (sub' u) (sub' v)
  sub' (TOp b s' t u)  = TOp b s' (sub' t) (sub' u)
  sub' t               = t

newVar v vs | v `elem` vs = newVar (v++"'") vs
```

```
  | otherwise    = v
```

Next, we identify two modal properties of terms. The first is talked about in the thesis; it is *modal closedness* and extends Definition 2.15. It is implemented by the `mClosed` function. The second is experimental and allows more freedom in certain contexts; it is meant to identify when a term is *guaranteed to terminate* and is implemented by the `terminate` function.

```
mClosed :: Term → Bool
mClosed (TLoc _)          = False
mClosed (TEx _ _)         = False
mClosed (TAss _ _ _ _)    = False
mClosed (TSeq _ _ _ _)    = False
mClosed (TLam _ x t)      = mClosed t
mClosed (TApp t u)        = mClosed t ∧ mClosed u
mClosed (TCond _ t u v)   = mClosed t ∧ mClosed u ∧ mClosed v
mClosed (TOp _ _ t u)     = mClosed t ∧ mClosed u
mClosed _                 = True
terminate :: Term → Bool
terminate (TLoc _)          = False
terminate (TEx _ t)         = False
terminate (TAss _ _ t u)    = terminate t ∧ terminate u
terminate (TSeq _ _ t u)    = terminate t ∧ terminate u
terminate (TLam _ x t)      = terminate t
terminate (TApp t u)        = terminate t ∧ terminate u
terminate (TCond _ t u v)   = terminate t ∧ terminate u ∧ terminate v
terminate (TOp _ _ t u)     = terminate t ∧ terminate u
terminate _                 = True
```

Now, we look at rewriting. The first point we would like to make here is that rewriting is implemented in two ways: shallow and deep. Shallow rewriting of (say) assignment consists of pushing an assignment inward by one step (as in Definition 4.2); deep rewriting pushes it in as far as possible.

Rewriting is implemented by the `rewrite` function, which handles parenthesization and other organizational aspects. To handle the rewriting of specific kinds of operators, `rewrite` calls one of `rewriteCond` (for conditionals), `rewriteOp` (for operators), `rewriteLam` (for $\beta$-conversion), `rewriteEx` (for extension operators), or `rewriteAss` (for assignment statements).

```
rewrite :: Bool → Term → Term
rewrite d = rW
 where
  rW (TEx True t)              = rewriteEx d $ rW t
  rW (TEx _    t)              = TEx False $ rW t
  rW (TSeq b l (TAss True l' t u) v)
                               = rewriteAss d l' b l (rW t) (rW u) (rW v)
  rW (TSeq b l (TSeq b' l' t u) v)
                               = rW $ TSeq b' l' t $ TSeq b l u v
  rW (TSeq True True t u)      | mClosed u = rW u
  rW (TSeq b l t u)            = TSeq b l (rW t) (rW u)
  rW (TAss b l t u)            = TAss b l (rW t) (rW u)
  rW (TIn t)                   = TIn $ rW t
  rW (TApp (TLam True x u) v) = rewriteLam d x (rW u) (rW v)
  rW (TApp t u)                = TApp (rW t) (rW u)
  rW (TLam b x t)              = TLam b x $ rW t
  rW (TCond True t u v)        = rewriteCond (rW t) (rW u) (rW v)
  rW (TCond b t u v)           = TCond b (rW t) (rW u) (rW v)
  rW (TOp True s t u)          = rewriteOp s (rW t) (rW u)
  rW (TOp b s t u)             = TOp b s (rW t) (rW u)
  rW t                         = t
```

```
rewriteCond (TBool True)  = const
rewriteCond (TBool False) = flip const
rewriteCond t             = TCond True t

rewriteOp "+"   (TInt n) (TInt m)   = TInt $ n + m
rewriteOp "-"   (TInt n) (TInt m)   = TInt $ n - m
rewriteOp "*"   (TInt n) (TInt m)   = TInt $ n * m
rewriteOp "<"   (TInt n) (TInt m)   = TBool $ n < m
rewriteOp ">"   (TInt n) (TInt m)   = TBool $ n > m
rewriteOp "="   t        u          = TBool $ t ≡ u
rewriteOp "=="  t        u          = TBool $ t ≡ u
rewriteOp "/="  t        u          = TBool $ t ≠ u
rewriteOp "<="  (TInt n) (TInt m)   = TBool $ n ≤ m
rewriteOp ">="  (TInt n) (TInt m)   = TBool $ n ≥ m
rewriteOp "&&"  (TBool b) (TBool b') = TBool $ b ∧ b'
rewriteOp "||"  (TBool b) (TBool b') = TBool $ b ∨ b'
rewriteOp ";;"  (TIn t)   (TIn u)    = TIn $ TSeq False False t u
rewriteOp ";;;" (TIn t)   (TIn u)    = TIn $ TOp False ";;" t u
rewriteOp s t u                      = TOp True s t u

rewriteEx d (TIn t)                = t
rewriteEx d (TApp (TLam b x t) u) = TApp (TLam b x ((if d then
                                        rewriteEx d else TEx True) t)) u
rewriteEx d t                      = TEx True t
```

The heart of the rewriting function is the `rewriteAss` function that deals with rewriting assignment statements; it is the implementation of the set of rules in Definition 4.2 along with those for all extensions in Chapter 5.

```
rewriteAss deep lazy seqb seql (TIn (TLoc x)) t = rAX
 where
  end        = TAss True lazy (TIn (TLoc x)) t
  stop       = TSeq seqb seql end
  next       = if deep then rAX else stop
  safe s     = mClosed t
  apply u    = if lazy ∨ terminate t then u else stop u
  fV         = freeVars t

  appendSeq sb sl t (TSeq b l u v) = TSeq b l u $ appendSeq sb sl t v
  appendSeq sb sl t u              = TSeq sb sl u t

  rAX u@(TLoc y)       | y ≡ x     = t
                       | otherwise = apply u
  rAX u@(TEx b v)      | safe x    = stop $ TEx b $ next v
                       | otherwise = stop u
  rAX u@(TAss b l (TIn (TLoc y)) v)
                       | y ≡ x     = TAss b l (TIn (TLoc y)) (next v)
                       | safe y    = TSeq seqb seql (TAss b l
                                          (TIn (TLoc y)) (next v)) end
                       | otherwise = stop u
  rAX u@(TAss b l v w) | safe x    = stop $ TAss b l (next v) w
                       | otherwise = stop u
  rAX (TSeq b l (TSeq b' l' u v) w)
                      = rAX $ TSeq b' l' u $ TSeq b l v w
  rAX (TSeq b l (TAss b' l' (TIn (TLoc y)) u) w)
                       | y ≡ x     = TSeq b l (TAss b' l'
                                          (TIn (TLoc y)) (next u)) w
                       | safe y    = TSeq b l (TAss b' l'
                                          (TIn (TLoc y)) (next u)) (next w)
```

```
  rAX (TSeq b l u v)   = appendSeq b l v $ rAX u
  rAX (TLam b y u)     | y`elem`fV = let z = newVar (y++"'") fV in TLam b z
                                       $ next $ rewriteLam True y u $ TVar z
                       | otherwise = TLam b y $ next u
  rAX (TApp u v)       = TApp (next u) (next v)
  rAX (TCond b u v w)  = TCond b (next u) (next v) (next w)
  rAX (TOp b s u v)    = TOp b s (next u) (next v)
  rAX u                = apply u

rewriteAss _ lazy seqb seql t u = TSeq seqb seql $ TAss True lazy t u
```

The `rewriteLam` function deals with $\beta$-conversion of the functional ($\lambda$-calculus) features in our implementation. Recall (§5.5) that $\beta$-conversion cannot always be carried out.

```
rewriteLam deep x t' t = rLX t'
 where
  next t' | deep      = rLX t'
          | otherwise = TApp (TLam True x t') t

  rLX (TVar y)       | x ≡ y             = t
  rLX (TEx b u)                          = TEx b $ next u
  rLX v@(TIn u)      | mClosed t         = TIn $ next u
                     | x`elem`freeVars u = TApp (TLam True x v) t
                     | otherwise         = TIn u
  rLX v@(TLam b y u) | x ≡ y             = v
                     | y`elem`freeVars t
                       = let z = newVar (y++"'") (freeVars t)
                          in TLam b z $ next $ rewriteLam True y u (TVar z)
                     | otherwise         = TLam b y $ next u
```

```
rLX (TAss b l u v)                    = TAss b l (next u) (next v)
rLX (TSeq b l u v) | mClosed t        = TSeq b l (next u) (next v)
                   | otherwise        = TApp (TLam True x $
                                            TSeq b l (next u) v) t
rLX (TApp u v)                        = TApp (next u) (next v)
rLX (TCond b u v w)                   = TCond b (next u)
                                          (next v) (next w)
rLX (TOp b s u v)                     = TOp b s (next u) (next v)
rLX u                                 = u
```

## A.4 Interaction

The most important part of the implementation is the interactive mode, where a user can experiment with the rewriting rules of ***AC***. Interaction with a term `t` is initiated by entering

```
act t
```

The following table lists all of the keys that can be used in interactive mode along with their effects. Notice that each key has a normal effect and a "shifted" effect, which is its effect if it is pressed while holding Shift. It is therefore important to make sure that Caps Lock is off when in interactive mode.

| Control key | Action | Shifted action |
|---|---|---|
| Arrow key | Move cursor by 1 | Move cursor and screen by 5 |
| Page Up/Down | Move cursor and screen by 5 | Move cursor and screen by 5 |
| Space | Toggle selection at cursor | Toggle selection at cursor |
| a | Select all | Deselect all |
| r, Enter, d | Deep rewrite | Deep rewrite |
| s | Shallow rewrite | Shallow rewrite |
| n | Next selection point | Previous selection point |
| p | Previous selection point | Next selection point |
| m | Select all modal operators | Deselect all modal operators |
| f | Select all functional operators | Deselect all functional operators |
| o | Select all other operators | Deselect all other operators |
| q | Quit | Quit |

## A.5 Examples

The following example terms are included with the implementation.

1. `simple1.ac`. This example is just a more involved version of Example 1 from §2.1.

```
X := 1;
Y := 2;
Z := 3;
X := Y + X;
Y := Y + 1;
Z := X * 2;
X := 5;
Y := Y - Z
```

2. `simple2.ac`. This example is similar to Example 2 from §2.1.

```
X := 1;
P := ^X;
X := 3;
!P
```

3. `simple3.ac`. Here we have an example of abstraction and application.

```
X := 1; (if Y := 4; X < Y then \x . x + 3
                          else \x . x + 5) 4)
```

4. `factorial1.ac`. See Example 3, §2.1.

```
P := ^( if Y > 1
        then Y * (Y := Y - 1; !P)
        else 1
      );
Y := 6;
!P
```

5. `factorial2.ac`. An alternative approach to factorial, this one looks more like a traditional imperative program. It does not make use of state backtracking.

```
X := ^(!Y; !X);
Y := ^(if N > 0 then M := M * N;
                     N := N - 1
                else X := ^(X := X));
N := 6;
M := 1;
!X;
M
```

6. `factorial3.ac`. This form of factorial follows the example given in §5.5.

```
Fact := ^(F := ^(\x . if x<=1 then 1
                                  else x * !F (x - 1)); !F);
!Fact 6
```

7. `while1.ac`. This example gives an ***AC*** encoding of a `while`-loop.

```
X := 1;
Y := 6;
While := ^(if !WCond
           then !WLoop;
                !While
           else (X := X)
          );
WCond := ^(Y > 1);
WLoop := ^(X := Y * X;
           Y := Y - 1
          );
!While;
X
```

8. `while2.ac`. Another `while`-loop example, this one is passed the test and body as parameters.

```
While := ^(\wcond. \wloop.
              if !wcond then !wloop;
                             !While wcond wloop
                        else (X:=X)
          );
Y:=5;
X:=1;
```

```
!While ^(Y > 1) ^(X := X * Y; Y := Y - 1);
X
```

9. `lvalue1.ac`. A small demonstration of L-values.

```
Y := True; X := False; (if X then ^X else ^Y) <- False; Y
```

10. `lazy1.ac`. A demonstration of the usefulness of lazy assignment. We recommend trying this example with a strict assignment to see what happens.

```
Fact :=~ (F := ^(\x . if x <= 1 then 1
                                else x * !F (x - 1));
          !F);
Fact 5
```

11. `stack1.ac`. An interesting demonstration of the power of labelled state backtracking, this gives an implementation of a stack using one state-typed location.

```
Top := ^(S; X);
Push := ^(\n . S := (X := n));
Pop := ^(S:= (S; S));
!Push 1;
!Push 2;
!Push 3;
!Pop;
!Top
```

12. `composition.ac`. A small demonstration of the composition operator.

```
!(X:= ^(Y := 1);
    (X ;; ^Y)
 )
```

# Bibliography


[ABPS97] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-0: An imperative language that supports declarative programming. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1997.

[AJ94] Samson Abramsky and Achim Jung. Domain Theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.

[Bac78] John Backus. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[BBG+97] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language algol 60. *ALGOL-like Languages, Volume 1*, pages 19–49, 1997.

[BT83] Andrzej Blikle and Andrzej Tarlecki. Naive denotational semantics. In *IFIP Congress*, pages 345–355, 1983.

[BW88] Richard Bird and Philip Wadler. *An introduction to functional programming.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[Car47] Rudolf Carnap. *Meaning and Necessity.* University of Chicago Press, Chicago, 1947.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[Chu51] Alonzo Church. A formulation of the logic of sense and denotation. In Paul Henle, editor, *Structure, Method and Meaning: Essays in Honor of Henry M. Sheffer*, pages 3–24. Liberal Arts Press, 1951.

[CKL02] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Rewriting calculus with(out) types. In Fabio Gadducci and Ugo Montanari, editors, *Proceedings of the fourth workshop on rewriting logic and applications*, Pisa (Italy), 2002. Electronic Notes in Theoretical Computer Science.

[dB80] Jaco de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, 1980.

[Dij68] Edsger W. Dijkstra. Go To statement considered harmful. *Comm. ACM*, 11(3):147–148, 1968. letter to the Editor.

[FFDM87] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Computer Science Dept. Technical Report 216, Indiana University, Bloomington, Indiana, February 1987.

[FH88] A. J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, July 1988.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[Fre92] Gottlob Frege. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, 100:25–50, 1892. Translated as "On Sense and Reference" in [GB52].

[FW80] Joyce Friedman and David S. Warren. $\lambda$-normal forms in an intensional logic for English. *Studia Logica*, 39:311–324, 1980.

[Gal75] Daniel Gallin. *Intensional and Higher-Order Modal Logic.* North-Holland, Amsterdam, 1975.

[GB52] Peter Geach and Max Black, editors. *Translations from the Philosophical Writings of Gottlob Frege.* Blackwell, 1952.

[GS90] Jeroen Groenendijk and Martin Stokhof. Dynamic Montague Grammar. In *Papers from the Second Symposium on Logic and Language*, pages 3–48. Akademiai Kiadoo, 1990.

[Hen63] Leon Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52(323-344):2, 1963.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, Cambridge, MA, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HR77] Jerry R. Hobbs and Stanley J. Rosenschein. Making computational sense of Montague's intensional logic. *Artif. Intell.*, 9(3):287–306, 1977.

[Hun90] Hing-Kai Hung. *Compositional Semantics and Program Correctness for Procedures with Parameters.* PhD thesis, SUNY-Buffalo, 1990. Available as Computer Science Dept. Technical Report 90-18.

[HZ91] Hing-Kai Hung and Jeffery Zucker. Semantics of pointers, referencing and dereferencing with intensional logic. *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 127–136, 1991.

[Jan86] Theo M.V. Janssen. *Foundations and Applications of Montague Grammar: Part 1: Philosophy, Framework, Computer Science.* Centrum voor Wiskunde en Informatica, 1986.

[JvEB77] Theo M. V. Janssen and Peter van Emde Boas. On the proper treatment or referencing, dereferencing and assignment. In Arto Salomaa and Magnus Steinby, editors, *ICALP*, volume 52 of *Lecture Notes in Computer Science*, pages 282–300. Springer, 1977.

[Kah87] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[Kri] Neel Krishnaswami. Separation logic for a higher-order typed language. Draft presented at SPACE 2006 workshop.

[Kri59] Saul A. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

[Lau93] John Launchbury. Lazy Imperative Programming. In *ACM SigPlan Workshop on State in Prog. Langs.*, June 1993.

[LS08] Luigi Liquori and Bernard Paul Serpette. IRho: An imperative rewriting calculus. *Mathematical. Structures in Comp. Sci.*, 18(3):467–500, 2008.

[MHGG79] J. Marti, A. C. Hearn, M. L. Griss, and C. Griss. Standard LISP report. *SIGPLAN Not.*, 14(10):48–68, 1979.

[Mon70] Richard Montague. Universal grammar. *Theoria*, 36:373–398, 1970. Reprinted in [Tho74].

[Mon73] Richard Montague. The proper treatment of quantification in ordinary English. In K.J.J. Hintikka, J.M.E. Moravcsik, and P. Suppes, editors, *Approaches to Natural Language*, pages 221–242. Reidel, 1973. Reprinted in [Tho74].

[MS87] C. Mirkowska and Andrzej Salwicki. *Algorithmic Logic.* Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[P+03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Qui60] W. V. Quine. *Word and Object.* MIT Press, Cambridge, MA, 1960.

[Rea89] Chris Reade. *Elements of functional programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[Sch86] David A. Schmidt. *Denotational semantics: a methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG–2, Oxford University Computing Laboratory, Oxford, England, November 1970.

[SHLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical Theory of Domains.* Cambridge University Press, New York, NY, USA, 1994.

[SS71] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., April 1971. Polytechnic Press.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[Str73] Christopher Strachey. *The varieties of programming language.* Oxford University Computing Laboratory, Programming Research Group, Oxford,, 1973.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Programming Research Group Technical Monograph PRG-11, Oxford Univ. Computing Lab., 1974. Reprinted in *Higher-Order and Symbolic Computation*, vol. 13 (2000), pp. 135–152.

[Tho74] Richmond H. Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague.* Yale University Press, 1974.

[TN01] Allen B. Tucker, Jr. and Robert E. Noonan. *Programming Languages: Principles and Paradigms.* McGraw-Hill Higher Education, 2001.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[vEB90] Peter van Emde Boas. Machine models and simulations. *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 1–66, 1990.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[ZP93] Jeffery Zucker and Laurette Pretorius. Introduction to computability theory. *South African Computer Journal*, 9:3–30, 1993.