Programming Research Group

A GAMES-BASED FOUNDATION FOR COMPOSITIONAL SOFTWARE MODEL CHECKING

Dan R. Ghica

PRG-RR-02-13

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

A thesis submitted to the
School of Computing, Queen's University
Kingston, Ontario, Canada,
in conformity with the requirements for
the degree of Doctor of Philosophy

submitted November, 2002

onorabilei Bombonilă

Abstract

We present a program specification language for Idealized Algol that is compatible both with inferential reasoning and model checking. Model-checking is made possible by the use of an algorithmic, regular-language semantics, which is a representation of the fully-abstract game semantic model of the programming language. Inferential reasoning is carried out using rules based on Hoare's logic of imperative programming, extended to handle procedures and computational side effects. The main logical innovation of this approach is the use of generalized universal quantifiers to specify properties of non-local objects. Together, the regular-language semantics of the programming language and its specification language on the one hand, and the inferential properties of the specification language on the other provide a foundation for compositional software model checking.

Co-authorship

Chapter 4 (*Regular-language semantics*) is based on research work done jointly with Guy McCusker, School of Cognitive and Computing Science, University of Sussex at Brighton, Falmer, Brighton, UK [GM00, GM].

Acknowledgements

I owe much gratitude to my supervisor, Bob Tennent. His generosity and open-mindedness gave me the chance to explore then pursue a research topic that I found interesting, challenging and fun, while his keen research sense steered me away from many dead ends.

Another great debt I owe to Guy McCusker, for explaining games semantics to me and for patiently answering my silly questions. Without his help this research would have been impossible.

I thank the external examiner, Steve Brookes, for reading this work with great care and giving many useful suggestions.

The support and encouragement I received from Samson Abramsky and Luke Ong were important motivators for my research. I thank them for finding interest in my work.

An invaluable source of ideas and motivation were the people I have met at various conferences, especially MFPS, and on my trips to University of Sussex and University of Edinburgh: Cristiano Calcagno, Paul Levy, Peter O'Hearn, John Power, Uday Reddy, John Reynolds, Hayo Thielecke, Hongseok Yang and many more.

For the great times I had in Kingston I thank Art, Dan and the rest of my friends, and especially Sebi and my soccer buddies.

The moral and financial support I have received from my parents has been tremendous and essential. I will never be able to repay your generosity.

Last, but not least, I want to thank my wife Georgi for her love, her indomitable spirit and especially her courage. To her I dedicate this work.

Contents

List of Figures

Chapter 1

Introduction

In 1997 David Schmidt wrote a brief but influential article for the *ACM Conference on Strategic Directions in Computing Research* titled "On the need for a popular semantics," critically evaluating the progress in and the direction of research work in programming language semantics [Sch97]. Although the author identifies some important successes, most notably advances in type systems and providing guidance for the increasingly popular object-oriented paradigm, the tone of the article is, by and large, a critical one. Dis-satisfaction is expressed with regard to the plethora of specialized algebraic techniques, logics and calculi which keep modern semanticists busy but require too much technical background to become a part of practical programming or the undergraduate curricula. As a result, claims the author, semantic research has had disappointingly little impact on language design and implementation and on program verification. A rather stark warning is delivered: semantics runs the risk of "specializing out of the consciousness of the public" and getting stuck in the rather infertile pursuit of dissecting programming features.

In his paper, Schmidt raises a challenge for semantic research:

> A challenge for semantics writers is the following: design a calculational semantics for a significant subset of, say, JAVA, that can be learned and applied by first-year university students to debug their programs.

The semantics presented in this dissertation represents, in the author's opinion, the best, and perhaps the only, effort to date that meets to a significant extent Schmidt's challenge.

The most natural way in which semantic techniques can be presented is by using a concrete programming language. The language used throughout this dissertation is not JAVA, as suggested by Schmidt, but a variant of IDEALIZED ALGOL [Rey81a] (IA). IA is a language expressive enough to support many common programming idioms but compact and uniform enough to allow an elegant semantics. This is the principal reason why ALGOL is chosen as the main presentation vehicle; the concepts and techniques that will be introduced need a concrete programming language to which they can be applied. But there seems to be little reason to believe that these concepts and techniques cannot be extended to other programming language fragments [Ghi01a, Ong02].

That IA has such an elementary semantics is in itself quite remarkable. The language has attracted much attention from semantic researchers because it neatly and uniformly combines a procedural mechanism based on the lambda calculus with the simple imperative language of assignments, branching, iteration and assignable local variables. The two facets of IA, functional and imperative, have been quite well understood, in isolation. Therefore, it seems that from the beginning there was an implicit expectation that the semantic models of the apparently orthogonal functional and imperative features would neatly combine into an adequate model for IA. However, that was not the case at all; the two-volume collection [OT97] amasses almost two decades' worth of research trying to pin down the subtle and complex interactions of IA's *abstract store*. The historical survey [TG00] traces even farther back in time research efforts along the same lines, to the pioneering work of Christopher Strachey [Str64].

Although the research work mentioned above did not lead to a technically perfect denotational semantic model of IA, it provided key insights, both intuitive and technical, into the *local* and *parametric* nature of store. The first concept embodies the principle that local variables cannot be affected by actions of non-local procedures. The pioneering work of Oles and Reynolds [Ole82], using *functor categories*, had a substantial and lasting impact. For example, current work in *pointer logic* [ORY01, *et.al.*] is based on ideas stemming from this early work. The second concept, *parametricity*, reflects a fundamental intuition of *representation independence*, and led to significant further foundational work [FJ+96]. To date, the most advanced denotational model of IA based on traditional techniques, fully abstract for the second order fragment, is Reynolds and O'Hearn's translation of IA into the *polymorphic linear lambda calculus* [OR00].

§

In the 1960s, around the same time Scott and Strachey were laying the foundations of the mathematical theory of semantics, Lorenzen was introducing a radical new alternative to semantics of logic, through game theory [Lor60]. Lorenzen saw the process of proving a logical formula as a game between a verifier and a falsifier; the truth, respectively falsity, of a proposition amounts to the existence of a *winning strategy* for the verifier, respectively the falsifier, in the game of logic. Lorenzen was influenced by earlier work of Zermelo [Zer13], von Neumann [vNM44] and Nash [Nas50], who created a theory of economics based on parlor games. These revolutionary ideas proved to have a profound impact on the foundations of mathematics [Con76] and mathematical logic [Hin96] but they only began to be used by computer scientists in the 1990s, to give a semantic interpretation to *linear logic* [Bla92]. Independently, Hyland and Ong [HO00], Abramsky *et. al.* [AMJ94] and Nickau [Nic94] used game theory to give the first syntax-independent fully-abstract semantics to the language PCF; shortly afterwards, Abramsky and McCusker developed the first syntax-independent fully abstract semantic model of IA [AM96]. A plethora of fully abstract games-based models for programming languages followed in rapid succession in that period, establishing game theory as a powerful new tool in programming semantics: recursive types [McC98], call-by-value [AM98], generalized references [AHM98], nondeterminism [HM99], control [Lai97], CSP [Lai01], *etc.*

It is rather striking that game theory, developed for economics, an area that has little apparent in common with computer science, gave the first technically complete answer to the questions formulated by Strachey. It is also striking how different the two models are. The traditional Scott-Strachey models rely on set-theoretic foundations and take an *extensional* and *static* view of computation. By contrast, games models have combinatorial foundations and take an *intensional* and *dynamic* view of computation. Maybe the example that best illustrates this difference is the way in which the two models deal with store. The traditional models are stateful, and store, i.e. the collection of "stack" variables in use at any moment, is key, parameterizing the meaning of all phrases. By contrast, in the stateless, behavioural, games model, store is simply ignored, variables being represented by abstract traces of basic *read* and *write* actions. This style of semantics, based on *action traces*, has been used before to model interference-controlled [Red96] and parallel IA [Bro93].

However, although games-based models of computation proved to be technically satisfactory, they lacked other important qualities. They often seem opaque, somehow unable to reflect basic computational intuitions; they used complex and unwieldy notations which made many researchers think that their only possible use was to prove meta-theoretical results and that they were unlikely to have any impact on practice. In fact, only two years ago I was writing the following [TG00]:

> The stateful view is conceptually familiar and allows particular equivalences to be validated quite easily; the behavioural view is conceptually more sophisticated but allows proofs of general properties such as full abstraction.

But subsequent developments showed, surprisingly, this criticism of games to be unfair. Under certain restrictions and subject to certain simplifications, it is the games model which gives a truly elementary tool for reasoning about particular equivalences [GM00].

In joint work with Guy McCusker, one of the authors of the games model for IA, we showed how in the absence of higher order procedures and recursion much of the games technical apparatus can be discarded. What is left is simple enough to be described using only a meta-language of extended regular expressions. Using this meta-language we were able to give elementary and calculational proofs to equivalences that required rather sophisticated techniques in the traditional model. This semantics is very appealing indeed, as it is simple and calculational. It can be convincingly argued that this semantics meets Schmidt's challenge.

In the much simplified regular-language formulation, it is easy to see that game theoretic semantics can also give a good intuitive understanding of computation. In fact, except for the interpretation of procedures, the regular-language semantics is virtually identical to the abstract trace semantics used by David Harel in the 1970s to interpret dynamic logic [Har79]. So, albeit via a circuitous route, game semantics seems to vindicate very old semantic ideas.

§

According to Schmidt, one of the virtues of a popular semantics should be to offer good support for software verification. The regular-language semantics, in addition to being conceptually neat, has an immediate important practical application to program verification through *model checking*. Hankin and Malacaria first applied games-based techniques to program analysis of programs but to a rather different end, flow analysis [HM98, *et.al.*].

Model checking is a system verification technique based on semantics: the verifier must check whether a given system is a *model* for some formula [CGP99, *et.al.*]; this method is to be contrasted with *theorem proving*, which is based on logical inference: the verifier must find a proof in the logic for some formula. Model checking is currently enjoying increased popularity because it is more easily automated than theorem proving. The main impediment to model checking is the fact that the verification problem is computationally demanding in the extreme; however, constant advances in algorithms and data representations on the one hand and hardware on the other make this problem seem less and less daunting, to the extent that model checking of hardware systems is becoming common practice in industry [CW+96].

Software model checking (SMC), applying model checking to software verification, is a newer but active area of academic [CD+00] and industrial [BR01, Hol97] research, with an eye on immediate applicability. However, software systems do not seem to be as naturally suited to model checking as hardware system. The complexity of software is generally higher than that of hardware; and software has a more dynamic nature than hardware. Current SMC techniques are best adapted to deal with small, compact, monolithic programs, such as device drivers or network protocols.

It is generally acknowledged in the software model-checking community that the source of most problems specific to SMC is the so called *semantic gap*, i.e. the absence of mechanically checkable models of software. Indeed, traditional semantic models of programming languages are abstract and not calculational, therefore unsuitable for automatic verification. In the absence of suitable models, SMC must use *ad hoc* models based on finite state machines; such models are not always sound and complete and, because of that, are difficult to use. The greater difficulties are raised by the presence of non-local objects in the program to be verified, for example library function calls; they are often simply ignored. Such models may report non-existent errors or ignore real ones. One of the main theses of this work is that the regular-language semantics can successfully bridge this gap. Regular languages are an elementary formalism for which virtually all interesting properties have been showed to be decidable [Sto74, et.al.].

Bridging the semantic gap not only gives a solid theoretical foundation to SMC but also leads to natural solutions for several of its major documented problems [Kur97]:

- *Local versus global verification.* Current SMC techniques cannot handle program fragments, that is programs with non-local identifiers, including functions. Consequently, only whole programs can be checked. But since industrial programs are large, and SMC is computationally demanding, the verification of large scale programs using this method is not feasible. In industrial jargon this problem is sometimes referred to as a *scalability problem* for SMC.

- *Source-level versus model-level verification.* Many SMC techniques are adaptations from hardware verification. The program needs to be mapped into an automaton-like system, which is subject to verification. However, the mapping itself is not perfect and may introduce errors.

- *Unity of programming and specification languages.* SMC techniques derived from hardware verification apply to specification languages based on temporal logics. The natural properties to check in these languages are safety (that an undesirable event does not occur) and liveness (that a desirable event occurs). In the verification of software systems, however, one is concerned not only with events but also with values of program variables and relations among them. Expressing the latter in terms of the former is possible but artificial and too sophisticated for the programmer.

The first proposal to applying the regular-language semantics to deal with these issues is [Ghi01b].

But, perhaps, the greatest challenge of SMC is *compositionality*, that is, local verification combined with the ability to establish properties of a whole program from the properties of its individually-checked components without requiring checking the whole program all over again. In fact, local verification is of little use in the absence of compositional reasoning.

What compositional model checking really demands is a programming logic with a formal model that can be checked automatically. Such a logic, which can support both model checking and inferential reasoning, is the main focus of the second part of this presentation.

It is worth mentioning that the variant of IA used in this presentation was long considered totally unsuitable for any kind of formal program verification because it uses expressions with side-effects, i.e. expressions that can assign to non-local variables in the course of computing a value. As early as 1968 David Park had noted [Par68] that it is virtually impossible to reason about programs in procedural languages without assuming that expressions have no side-effects. A similar conviction determined John Reynolds to remove expressions with side-effects from the definition of IA [Rey81a] and to refer to the presence of such expressions in ALGOL 60 as a "bug."[1]

But expressions with side effects are a staple of programming. Trying to reconcile side-effect free expressions with natural programming idioms through constructions such as *block expressions* raises difficult syntactic, type-theoretic and semantic issues [TT91]. Moreover, even if assignments are banned from expressions, some computational effects cannot be eliminated by fiat, for example division by zero, overflow and non-termination. In Reynolds's influential *Craft of Programming* [Rey81b], non-termination in expressions is ignored, along with side-effects, but the author acknowledges that this is a serious limitation:

> [...] With much regret, we will avoid the use of recursive function procedures in this book. The reason is similar to that for avoiding expressions with

[1] The original specification of ALGOL 60 seems inconsistent in its semantic definitions of expressions (Section 3.3.3) and functions (Section 5.4.4), so there is indeed a "bug" in the language [NB$^+$63].

> side-effects. [...] Unfortunately, the possibility that such expressions might occur in assertions cannot be accommodated by the logic we are using for program specification.

Fortunately, the regular-language semantics of IA naturally gives rise to a logic that can deal with side-effects properly.

The logic presented in this dissertation is superficially similar to Reynolds's but it starts from fundamentally different semantic intuitions. Reynolds's key concept is that of *non-interference*; two phrases do not interfere if they do not share any store. The logic is concerned with *localizing* the effects of computation. By contrast, the key concept of the logic presented here is that of *stability*; a phrase is stable if it behaves in a certain way throughout the computation. The logic is therefore concerned with *globalizing* constraints on the behaviour of language objects. This notion of stability is reflected in the logic by a generalized quantifier. Generalized quantifiers have always attracted game theorists [Hin96] and they came about naturally in the logic presented here as a way to introduce non-local objects whose behaviour is subject to certain restrictions.

Another application of generalized quantifiers is to provide a model-checking-friendly specification and verification framework which formalizes a certain style of specifications called *abstract modeling*, and is semantically similar to de Alfaro and Henzinger's concept of *interface automata* [dAH01].

1.1 Outline

We will proceed by giving an overview of the language IA and the state-of-the-art operational model and programming logic. Some technical operational properties are needed later on, but the programming logic is provided only for comparison. In Chapter 3 we will describe the full games model of IA, then in the following chapter we will see how under certain language restrictions the model can be represented using regular languages only. This chapter will also illustrate the model by showing how several important equivalences of IA can be proved.

In Chapter 5 we discuss the problems raised by specifying and verifying program fragments of IA, and give the syntax and the semantics of a specification language. In the following chapter we explore the logical properties of the specification language and we prove the soundness of a system of inference rules. Finally, we extend the specification language to handle procedure and function specifications.

1.2 Thesis

This dissertation will defend the following thesis:

> *The game semantics of interesting programming language fragments can be represented by regular languages and manipulated by an elementary calculus of extended regular expressions; the regular language semantics is simple yet powerful enough to bridge the semantic gap of software model checking and,*

consequently, to support local verification, source-level verification and compositional reasoning using a specification language that is semantically similar to the programming language.

Chapter 2

First order Idealized ALGOL

ALGOL 60 *is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors.* Tony Hoare

Reynolds's Idealized ALGOL (IA) is a compact language which combines the fundamental features of procedural languages with a full higher-order procedure mechanism. This combination makes the language surprisingly expressive. For example, simple forms of classes and objects may be encoded in IA [Red98]. For these reasons, IA has attracted a great deal of attention from theoreticians; some 20 papers spanning almost 20 years of research were recently collected in book form [OT97].

IA is a streamlined version of ALGOL 60 [NB$^+$63]. It has been defined by Reynolds as a programming language that satisfies the following properties:

Principle 1. IA is obtained from the simple imperative language by imposing a procedure mechanism based on a fully typed, call-by-name lambda calculus.

Principle 2. There are two fundamentally different kinds of type: *data types*, each of which denotes a set of values appropriate for certain variables and expressions, and *phrase types*, each of which denotes a set of meanings appropriate for certain identifiers and phrases.

Principle 3. The order of evaluation for parts of expressions, and of implicit conversions between data or phrase types, should be indeterminate, but the meaning of the language, at an appropriate level of abstraction, should be independent of this indeterminacy.

Principle 4. Facilities such as procedure definition, recursion, and conditional and case constructions should be uniformly applicable to all phrase types.

Principle 5. The language should obey a stack discipline, and its definition should make this discipline obvious.

We conform to these principles, except for *Principle 3.* This variant of IA is known as IA with *active expressions* and has been analyzed extensively [Sie94, AM96, OR00]. We consider only the recursion-free second order fragment of this language, and only finite data sets.

2.1 Syntax

The data types τ of the language (i.e. types of data assignable to variables) are finite subsets of the integers and the booleans. The phrase types θ of the language are those of commands, variables and expressions, plus first-order function types.

$$\begin{aligned} \tau &::= \mathbf{int} \mid \mathbf{bool}, \\ \sigma &::= \mathbf{comm} \mid \mathbf{var}\tau \mid \mathbf{exp}\tau, \\ \theta &::= \sigma \mid \sigma \to \theta. \end{aligned}$$

Terms are introduced using type judgements of the form $\Gamma \vdash P : \theta$, where Γ is a finite function from identifiers to phrase types: $\Gamma = \{x_1 : \theta_1, \ldots, x_k : \theta_k\}$.

Let $\mathrm{dom}(\Gamma)$ and $\mathrm{rng}(\Gamma)$ be the domain, respectively the range of Γ. We use the following notation:

$$\Gamma \mid \Gamma' : \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma') \to \mathrm{rng}(\Gamma) \cup \mathrm{rng}(\Gamma'), \qquad (\Gamma \mid \Gamma')(x) \stackrel{\mathrm{def}}{=} \begin{cases} \Gamma(x) & \text{if } x \notin \mathrm{dom}(\Gamma') \\ \Gamma'(x) & \text{otherwise.} \end{cases}$$

The terms of the language and their typing rules are presented in Figure 2.1 on the facing page.

The data types of the language, i.e. the types of values assignable to variables, are bounded integers (**int**) and booleans (**bool**). The phrase-types, i.e. the types of terms, are commands (**comm**), boolean and integer variables (**varint**, **varbool**) and expressions (**expint**, **expbool**), as well as first-order functions. The usual operators of arithmetic and logic are employed.

The imperative constructs are the common ones: assignment (:=), command sequencing (;), iteration (**while**) and branching (**if**). Other common branching (**case**) and iterative constructs (**for**, **do-until**) are not included because they can be easily expressed in terms of the existing ones. They do not contribute semantically, being only what is called *syntactic sugar*. Branching is imposed uniformly on ground types, so we have branching for expressions (similar to the -?-:- operator in C), and variable-typed terms.

The behaviour of variables in imperative languages is dual, depending on whether they occur on the left-hand side (*l-values*) or right-hand side (*r-values*) of assignment statements. The proper behaviour is usually automatically resolved by compilers using type-coercion rules, from variable types to expression types, when a variable is used on the right-hand side. For clarity of presentation we will not introduce such coercion rules, but we will use instead an explicit de-referencing operator (!) in the language.

The main difference between the IA variant presented here and Reynolds's is that commands can be sequenced not only with commands but also with expressions or variables. The result is what is called an *active expression* (*active variable*, respectively). The informal semantics of an active expression is that it calculates a value while possibly

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{comm}} \qquad \frac{}{\Gamma \vdash \mathbf{diverge} : \mathbf{comm}}$$

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{expbool}} \qquad \frac{}{\Gamma \vdash n : \mathbf{expint}}$$

$$\frac{}{\Gamma \mid x : \theta \vdash x : \theta} \qquad \frac{\Gamma \vdash V : \mathbf{var}\tau}{\Gamma \vdash !V : \mathbf{exp}\tau}$$

$$\frac{\Gamma \vdash E_1 : \mathbf{expint} \quad \Gamma \vdash E_2 : \mathbf{expint}}{\Gamma \vdash E_1 + E_2 : \mathbf{expint}} \qquad \frac{\Gamma \vdash E_1 : \mathbf{expint} \quad \Gamma \vdash E_2 : \mathbf{expint}}{\Gamma \vdash E_1 = E_2 : \mathbf{expbool}}$$

$$\frac{\Gamma \vdash B_1 : \mathbf{expbool} \quad \Gamma \vdash B_2 : \mathbf{expbool}}{\Gamma \vdash B_1 \ \mathbf{and}\ B_2 : \mathbf{expbool}} \qquad \frac{\Gamma \vdash B : \mathbf{expbool}}{\Gamma \vdash \mathbf{not}\ B : \mathbf{expbool}}$$

$$\frac{\Gamma \vdash V : \mathbf{var}\tau \quad \Gamma \vdash E : \mathbf{exp}\tau}{\Gamma \vdash V := E : \mathbf{comm}} \qquad \frac{\Gamma \vdash C : \mathbf{comm} \quad \Gamma \vdash M : \sigma}{\Gamma \vdash C; M : \sigma}$$

$$\frac{\Gamma \vdash B : \mathbf{expbool} \quad \Gamma \vdash M_1 : \sigma \quad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash \mathbf{if}\ B\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \sigma} \qquad \frac{\Gamma \vdash B : \mathbf{expbool} \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{while}\ B\ \mathbf{do}\ C : \mathbf{comm}}$$

$$\frac{\Gamma \vdash F : \sigma \to \theta \quad \Gamma \vdash M : \sigma}{\Gamma \vdash FM : \theta} \qquad \frac{\Gamma \mid v : \mathbf{var}\tau \vdash M : \sigma}{\Gamma \vdash \mathbf{new}\tau\ v\ \mathbf{in}\ M : \sigma}$$

Figure 2.1: Terms and typing rules of IA

writing to non-local variables. This is a common feature of most imperative languages. One special command of the language is **diverge**. It causes the execution of the program to enter a state similar to that caused by an infinite loop. The command that performs no operation, similar to the empty command in C or PASCAL, is **skip**.

We also use first-order lambda-abstraction and a *let* constructor for declarations:

TYPING RULES

$$\frac{\Gamma \mid m : \sigma \vdash P : \theta}{\Gamma \vdash \lambda m{:}\sigma.P : \sigma \to \theta} \qquad \frac{\Gamma \vdash P : \theta \qquad \Gamma \mid x : \theta \vdash P' : \theta'}{\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ P\ \mathbf{in}\ P' : \theta'}$$

We call this self-contained programming language *first-order IA*.

Throughout the dissertation we will use meta-variables consistently so as to minimize the need to decorate them with their types, whenever possible. The complete list of meta-variables is in Appendix A on page 171.

2.2 Operational semantics

Operational semantics, in general, is a clear and convenient way to specify a programming language, so it is common to use it as a benchmark by which to measure a denotational semantics. However, it is usually quite cumbersome to prove term equivalences or properties of programs using the operational semantics because proofs require induction on the definitions. For purposes other than specification a denotational semantics is usually more practicable.

But Pitts's [Pit96] operational semantics of IA is remarkable in that it exploits structural properties of the language discovered first in a *denotational* setting to greatly simplify equational reasoning. The insight that this operational semantics borrowed from the denotational semantics of IA is that terms of IA are *parametric*, that is they preserve logical relations on state sets. This property was first noticed by O'Hearn and Tennent [OT93a] and, independently, by Sieber [Sie94] and it helps establish other general properties of contextual equivalence, such as extensionality of functions

$$F \equiv_{\sigma\to\theta} F' \text{ if and only if } \forall M{:}\sigma. FM \equiv_\theta F'M,$$

reducing contextual equivalence at function type to contextual equivalence at ground types. But extensionality is not a general property of programming languages, failing in languages such as ML or SCHEME because of the interactions between call-by-value functions and pointers, respectively generalized control (**call/cc**).

In addition to providing a concise and handy definition of IA, and tools for reasoning about contextual equivalence, some of the technical results of this work are needed in the proof of full abstraction for the regular-language semantics. The main result that interests us is that of *operational extensionality*, Theorem 2.1 on page 17.

Pitts's treatment is based on IA with side-effect free expressions and higher-order procedures and recursion. Eliminating the higher-order and recursive procedures from the language is straightforward and does not change the proofs or the technical results. But introducing expressions with side-effects actually requires modifying some of the definitions and the proofs. In Pitts's paper, the side-effect free nature of expressions plays a certain technical role in proving the parametric properties of IA, but we will see that these properties also hold in the presence of side-effects. That IA with side-effects is parametric has been shown before, but in a denotational setting, by O'Hearn and Reynolds [OR00].

The operational semantics of IA is specified using an inductively defined evaluation relation of the form:

$$\Omega \vdash s, P \ \Downarrow_\theta \ s', P',$$

where Ω is a finite set of global variables, called a *world*, $\Omega = \{x_1 : \mathbf{var}\tau_1, \ldots, x_n : \mathbf{var}\tau_n\}$, P and P' are terms of IA such that $\Omega \vdash P : \theta$ and $\Omega \vdash P' : \theta$ and s, s' are states, that is functions mapping identifiers to values:

$$\begin{aligned} s, s' \in \mathit{States}(\Omega) = \big\{x \mapsto v \mid x \in \mathrm{dom}\ \Omega,\ \Omega(x) = \mathbf{varint},\ v \in \mathbb{Z}\big\} \\ \cup \big\{x \mapsto v \mid x \in \mathrm{dom}\ \Omega,\ \Omega(x) = \mathbf{varbool},\ v \in \{\mathbf{true}, \mathbf{false}\}\big\} \end{aligned}$$

If $\Omega \vdash s, P \Downarrow_\theta s', P'$ does not hold for any s' and P' then, by definition,

$$\Omega \vdash s, P \Uparrow_\theta .$$

Terms with free variables only of $\mathbf{var}\tau$ type will be called *semi-closed terms*.

The evaluation rules are given in Figure 2.2 on the next page, using big-step (natural) operational rules. We use $\star$ to denote the IA arithmetical and logical operators as well as their obvious interpretation. The rules have some differences when compared to Pitts's presentation. Some of the differences are minor and do not lead to any technical problem: the rules for application are restricted to first-order functions, and instead of lambda abstraction for higher order terms, only the more restricted **let** binder is allowed; general recursion is replaced by new rules for a **while** command; booleans are allowed as a data type.

In the rule for function application, by $P[-/-]$ we denote substitution, with renaming of bound identifiers to prevent capture.

We will be concerned with equations of the form:

$$\Gamma \vdash P \equiv_\theta P', \quad \text{where } \Gamma \vdash P : \theta, \Gamma \vdash P' : \theta.$$

In order to give a definition for contextual equivalence, we need two auxiliary definitions:

Definition 2.1 (Context) *A context $C[-_\theta]$ is a term in which a sub-term of type θ has been replaced by a "hole" $-_\theta$. The term resulting from replacing it with a term $P : \theta$ is denoted by $C[P]$.*

Definition 2.2 (Traps) *$Traps(C[-_\theta])$ is the set of identifiers that occur in $C[-_\theta]$ associated to binders containing $-_\theta$ within their scope.*

Definition 2.3 (Contextual equivalence) *If P_1, P_2 are terms of IA of type θ with global variables contained in $dom(\Omega)$ and the rest of its free identifiers contained in $dom(\Gamma)$, we write*

$$\Omega \mid \Gamma \vdash P_1 \equiv_\theta P_2$$

to indicate that they are contextually equivalent.

This means that for all worlds $\Omega' \supseteq \Omega$ and for all semi-closed contexts $C[-_\theta] : \mathbf{comm}$ with $Free(C[-_\theta]) \subseteq \Omega'$ so that $\Gamma \subseteq Traps(C[-_\theta])$, and for all states $s, s' \in States(\Omega')$,

$$\Omega' \vdash C[P_1], s \Downarrow s', \mathbf{skip} \textit{ if and only if } \Omega' \vdash C[P_2], s \Downarrow s', \mathbf{skip}.$$

For practical purposes the definition above is awkward because of the quantification over all contexts. The following, much simpler, kind of equivalence will be shown to be actually the same as the one above.

Definition 2.4 (Extensional equivalence of semi-closed terms) *If $\Omega \vdash M_i : \theta$, $i = 1, 2$ we write $\Omega \vdash M_1 \cong_\theta M_2$ to indicate that the terms are extensionally equivalent, defined inductively on the structure of θ as follows:*

$$\Omega \vdash s, \mathbf{true} \Downarrow_{\mathbf{expbool}} s, \mathbf{true}, \quad \Omega \vdash s, \mathbf{false} \Downarrow_{\mathbf{expbool}} s, \mathbf{false},$$

$$\Omega \vdash s, n \Downarrow_{\mathbf{expint}} s, n, \quad \Omega \vdash s, \mathbf{skip} \Downarrow_{\mathbf{comm}} s, \mathbf{skip}$$

$$\frac{\Omega \vdash s, B \Downarrow_{\mathbf{expbool}} s', \mathbf{b} \qquad \Omega \vdash s', M_{\mathbf{b}} \Downarrow_{\sigma} s'', M}{\Omega \vdash s, \mathbf{if}\ B\ \mathbf{then}\ M_{\mathbf{true}}\ \mathbf{else}\ M_{\mathbf{false}} \Downarrow_{\sigma} s'', M}$$

$$\frac{\Omega \vdash s, E_1 \Downarrow_{\mathbf{expint}} s', n_1 \qquad \Omega \vdash s', E_2 \Downarrow_{\mathbf{expint}} s'', n_2}{\Omega \vdash s, E_1 \star E_2 \Downarrow_{\mathbf{expint}} s'', n} \text{ if } n = n_1 \star n_2$$

$$\frac{\Omega \vdash s, B_1 \Downarrow_{\mathbf{expbool}} s', b_1 \qquad \Omega \vdash s', B_2 \Downarrow_{\mathbf{expbool}} s'', b_2}{\Omega \vdash s, B_1 \star B_2 \Downarrow_{\mathbf{expbool}} s'', b} \text{ if } b = b_1 \star b_2$$

$$\frac{\Omega \vdash s, F \Downarrow_{\sigma \to \theta} s', \lambda m{:}\sigma.P \qquad \Omega \vdash s', P[M/m] \Downarrow_{\theta} s'', P'}{\Omega \vdash s, FM \Downarrow_{\theta} s'', P'}$$

$$\frac{\Omega \vdash s, V \Downarrow_{\mathbf{varint}} s', v}{\Omega \vdash s, !V \Downarrow_{\mathbf{expint}} s', n} \text{ if } s'(v) = n \qquad \frac{\Omega \vdash s, V \Downarrow_{\mathbf{varbool}} s', v}{\Omega \vdash s, !V \Downarrow_{\mathbf{expbool}} s', b} \text{ if } s'(v) = b$$

$$\frac{\Omega \vdash s, E \Downarrow_{\mathbf{expint}} s', n \qquad \Omega \vdash s', V \Downarrow_{\mathbf{varint}} s'', v}{\Omega \vdash s, V := E \Downarrow_{\mathbf{comm}} (s'' \mid v \mapsto n), \mathbf{skip}}$$

$$\frac{\Omega \vdash s, B \Downarrow_{\mathbf{expbool}} s', b \qquad \Omega \vdash s', V \Downarrow_{\mathbf{varbool}} s'', v}{\Omega \vdash s, V := B \Downarrow_{\mathbf{comm}} (s'' \mid v \mapsto b), \mathbf{skip}}$$

$$\frac{\Omega \vdash s, C \Downarrow_{\mathbf{comm}} s', \mathbf{skip} \qquad \Omega \vdash s', M \Downarrow_{\sigma} s'', M'}{\Omega \vdash s, C; M \Downarrow_{\sigma} s'', M'}$$

$$\frac{\Omega \mid v' : \mathbf{varint} \vdash (s \mid v' \mapsto 0), M[v/v'] \Downarrow_{\sigma} (s' \mid v' \mapsto n), M'}{\Omega \vdash s, \mathbf{newint}\ v\ \mathbf{in}\ M \Downarrow_{\sigma} s'', M'} v' \notin \mathrm{dom}(\Omega)$$

$$\frac{\Omega \mid v' : \mathbf{varint} \vdash (s \mid v' \mapsto \mathbf{false}), M[v/v'] \Downarrow_{\sigma} (s' \mid v' \mapsto n), M'}{\Omega \vdash s, \mathbf{newbool}\ v\ \mathbf{in}\ M \Downarrow_{\sigma} s'', M'} v' \notin \mathrm{dom}(\Omega)$$

$$\frac{\Omega \vdash s, P'[x/P] \Downarrow_{\theta} s', P''}{\Omega \vdash s, \mathbf{let}\ x\ \mathbf{be}\ P\ \mathbf{in}\ P' \Downarrow_{\theta} s', P''}$$

$$\frac{\Omega \vdash s, B \Downarrow_{\mathbf{expbool}} s', \mathbf{false}}{\Omega \vdash s, \mathbf{while}\ B\ \mathbf{do}\ C \Downarrow_{\mathbf{comm}} s', \mathbf{skip}}$$

$$\frac{\Omega \vdash s, B \Downarrow_{\mathbf{expbool}} s', \mathbf{true} \qquad \Omega \vdash s', C; \mathbf{while}\ B\ \mathbf{do}\ C \Downarrow_{\mathbf{comm}} s'', \mathbf{skip}}{\Omega \vdash s, \mathbf{while}\ B\ \mathbf{do}\ C \Downarrow_{\mathbf{comm}} s'', \mathbf{skip}}$$

Figure 2.2: IA evaluation rules

- $\Omega \vdash E_1 \cong_{\mathbf{expint}} E_2$ *if for all* $s \in \mathit{States}(\Omega)$ *and all* $n \in \mathbb{Z}$,
$$\Omega \vdash s, E_1 \Downarrow_{\mathbf{expint}} s', n \quad \textit{if and only if} \quad \Omega \vdash s, E_2 \Downarrow_{\mathbf{expint}} s', n,$$
- $\Omega \vdash B_1 \cong_{\mathbf{expbool}} B_2$ *if for all* $s \in \mathit{States}(\Omega)$ *and all* $b \in \{\mathbf{true}, \mathbf{false}\}$,
$$\Omega \vdash s, B_1 \Downarrow_{\mathbf{expbool}} s', b \quad \textit{if and only if} \quad \Omega \vdash s, B_2 \Downarrow_{\mathbf{expbool}} s', b,$$
- $\Omega \vdash C_1 \cong_{\mathbf{comm}} C_2$ *if for all* $s \in \mathit{States}(\Omega)$
$$\Omega \vdash s, C_1 \Downarrow_{\mathbf{comm}} s', \mathbf{skip} \quad \textit{if and only if} \quad \Omega \vdash s, C_2 \Downarrow_{\mathbf{comm}} s', \mathbf{skip},$$
- $\Omega \vdash V_1 \cong_{\mathbf{varint}} V_2$ *if* $\Omega \vdash !V_1 \cong_{\mathbf{expint}} !V_2$ *and for all* $n \in \mathbb{Z}$,
$$\Omega \vdash (V_1 := n) \cong_{\mathbf{comm}} (V_2 := n),$$
- $\Omega \vdash V_1 \cong_{\mathbf{varbool}} V_2$ *if* $\Omega \vdash !V_1 \cong_{\mathbf{expbool}} !V_2$ *and for all* $b \in \{\mathbf{true}, \mathbf{false}\}$,
$$\Omega \vdash (V_1 := b) \cong_{\mathbf{comm}} (V_2 := b),$$
- $\Omega \vdash F_1 \cong_{\sigma \to \theta} F_2$ *if for all* $\Omega' \supseteq \Omega$ *and all* M *such that* $\Omega' \vdash M : \sigma$,
$$\Omega' \vdash F_1 M \cong_\theta F_2 M.$$

Definition 2.5 (Extensional equivalence of open terms) *If* $\Omega \mid \Gamma \vdash P_i : \theta$, $i = 1, 2$, *and* $\Gamma = \{x_1 : \theta_1, \ldots, x_n : \theta_n\}, \theta_i \neq \mathbf{var}\tau$ *for any* τ, *we say that* $\Omega \mid \Gamma \vdash P_1 \cong_\theta P_2$ *if for all* $\Omega' \subseteq \Omega$ *and all* $\Omega' \vdash P'_j : \theta_j$, $j = 1, \ldots n$:

$$\Omega' \vdash P_1[x_1, x_2, \ldots, x_n / P'_1, P'_2, \ldots, P'_n] \cong_\theta P_2[x_1, x_2, \ldots, x_n / P'_1, P'_2, \ldots, P'_n]$$

Theorem 2.1 (Operational extensionality) *IA contextual equivalence and extensional equivalence coincide:*

$$\Omega \mid \Gamma \vdash P_1 \equiv_\theta P_2 \quad \textit{if and only if} \quad \Omega \mid \Gamma \vdash P_1 \cong_\theta P_2.$$

The rest of this section is a technical presentation, showing how Pitts's proof of the theorem can be adapted to IA with active expressions. It can be omitted without loss of continuity.

The major difference between the languages lies in the introduction of a rule allowing side-effects at any ground type. This invalidates the following lemma used by Pitts:

Lemma 2.1 (Side-effect free expressions) *If* $\Omega \vdash s, P \Downarrow_\theta s', P'$ *and* $\theta \neq \mathbf{comm}$ *then* $s = s'$.

The essential definition of parametric logical relation, which relies on the lemma above, must be changed. We borrow the following notations and definitions from Pitts:

Definition 2.6 *We define the following:*

Lift: *If X is a set then* the lift of X *is* $(X_\perp, \leq)$, *the partially ordered set with elements* $X \cup \{\perp_X\}$, *where* $\perp_X \notin X$ *and ordering* $\perp_X \leq x$, *for all* $x \in X$.

Binary relations: $\mathit{Rel}(\Omega) \stackrel{def}{=} \{\mathcal{R} \subseteq \mathit{States}(\Omega)_\perp \times \mathit{States}(\Omega)_\perp \mid (\perp, \perp) \in \mathcal{R}\}$.

Identity relation: $\mathcal{I}_\Omega \stackrel{def}{=} \{(s, s) \mid s \in \mathit{States}(\Omega)_\perp\}$.

Smash product: $\mathcal{R}_1 \otimes \mathcal{R}_2 \in \mathit{Rel}(\Omega \cup \Omega')$ *for* $\mathcal{R}_1 \in \mathit{Rel}(\Omega)$, $\mathcal{R}_2 \in \mathit{Rel}(\Omega')$ *is defined to be*

$$\mathcal{R}_1 \otimes \mathcal{R}_2 \stackrel{def}{=} \{(s_1 \otimes s_2, s'_1 \otimes s'_2) \mid (s_1, s'_1) \in \mathcal{R}_1, (s_2, s'_2) \in \mathcal{R}_2\}$$

where Ω, Ω' *have disjoint domains and*

$$s_1 \otimes s_2 \stackrel{def}{=} \begin{cases} s_1 \cup s_2 & \textit{if } s_1 \neq \perp \neq s_2 \\ \perp & \textit{otherwise} \end{cases}$$

The new definition of parametric logical relations is:

Definition 2.7 (Parametric logical relation) *For each finite set Ω of global variables, each type θ, and each relation $\mathcal{R} \in \mathit{Rel}(\Omega)$, we define a binary relation between semi-closed terms of IA of type θ with global variables in Ω, denoted $\Omega \vdash P_1\ \mathcal{R}_\theta\ P_2$, where $\Omega \vdash P_i : \theta$, $i = 1, 2$, simultaneously by induction on the structure of θ as follows:*

- $\Omega \vdash E_1\ \mathcal{R}_{\mathbf{expint}}\ E_2$ *if, for all* $n_1, n_2 \in \mathbb{Z}$, *and for all* $(s_1, s_2) \in \mathcal{R}$,

  $\Omega \vdash s_1, E_1 \Downarrow_{\mathbf{expint}} s'_1, n_1$ *and* $\Omega \vdash s_2, E_2 \Downarrow_{\mathbf{expint}} s'_2, n_2$ *implies* $n_1 = n_2$ *and* $(s'_1, s'_2) \in \mathcal{R}$,
  $\Omega \vdash s_1, E_1 \Uparrow_{\mathbf{expint}}$ *and* $\Omega \vdash s_2, E_2 \Downarrow_{\mathbf{expint}} s'_2, n_2$ *implies* $(\perp, s'_2) \in \mathcal{R}$,
  $\Omega \vdash s_1, E_1 \Downarrow_{\mathbf{expint}} s'_1, n_1$ *and* $\Omega \vdash s_2, E_2 \Uparrow_{\mathbf{expint}}$ *implies* $(s'_1, \perp) \in \mathcal{R}$;

- $\Omega \vdash B_1\ \mathcal{R}_{\mathbf{expbool}}\ B_2$, *similar;*

- $\Omega \vdash V_1\ \mathcal{R}_{\mathbf{varint}}\ V_2$ *if, for all* $v_1, v_2 \in \Omega$, *and for all* $(s_1, s_2) \in \mathcal{R}$,

  $\Omega \vdash s_1, V_1 \Downarrow_{\mathbf{varint}} s'_1, v_1$ *and* $\Omega \vdash s_2, V_2 \Downarrow_{\mathbf{varint}} s'_2, v_2$
  implies for all $n \in \mathbb{Z}$, $\big((s'_1 \mid v_1 \mapsto n), (s'_2 \mid v_2 \mapsto n)\big) \in \mathcal{R}$ *and* $s'_1(v_1) = s'_2(v_2)$,
  $\Omega \vdash s_1, V_1 \Downarrow_{\mathbf{varint}} s'_1, v_1$ *and* $\Omega \vdash s_2, V_2 \Uparrow_{\mathbf{varint}}$
  implies for all $n \in \mathbb{Z}$, $\big((s'_1 \mid v_1 \mapsto n), \perp\big) \in \mathcal{R}$,
  $\Omega \vdash s_1, V_1 \Uparrow_{\mathbf{varint}}$ *and* $\Omega \vdash s_2, V_2 \Downarrow_{\mathbf{varint}} s'_2, v_2$
  implies for all $n \in \mathbb{Z}$, $\big(\perp, (s'_2 \mid v_2 \mapsto n)\big) \in \mathcal{R}$;

- $\Omega \vdash V_1\ \mathcal{R}_{\mathbf{varbool}}\ V_2$ *similar;*

- $\Omega \vdash C_1 \ \mathcal{R}_{\mathbf{comm}} \ C_2$ *if, for all* $(s_1, s_2) \in \mathcal{R}$,

$$\begin{array}{l}\Omega \vdash s_1, C_1 \ \Downarrow_{\mathbf{comm}} \ s_1', \mathbf{skip} \ \textit{and} \ \Omega \vdash s_2, C_2 \ \Downarrow_{\mathbf{comm}} \ s_2', \mathbf{skip} \ \textit{then} \ (s_1', s_2') \in \mathcal{R}, \\ \Omega \vdash s_1, C_1 \ \Downarrow_{\mathbf{comm}} \ s_1', \mathbf{skip} \ \textit{and} \ \Omega \vdash s_2, C_2 \Uparrow_{\mathbf{comm}} \ \textit{implies} \ (s_1', \bot) \in \mathcal{R} \\ \Omega \vdash s_1, C_1 \Uparrow_{\mathbf{comm}} \ \textit{and} \ \Omega \vdash s_2, C_2 \ \Downarrow_{\mathbf{comm}} \ s_2', \mathbf{skip} \ \textit{implies} \ (\bot, s_2') \in \mathcal{R}\end{array}$$

- $\Omega \vdash F_1 \ \mathcal{R}_{\sigma \to \theta} \ F_2$ *if, for all* $\mathcal{R}' \in Rel(\Omega')$ *with* Ω, Ω' *disjoint, and all* M_1, M_2 *such that* $\Omega \mid \Omega' \vdash M_i : \sigma$, $i = 1, 2$,

$$\Omega \mid \Omega' \vdash M_1 \ (\mathcal{R} \otimes \mathcal{R}')_\sigma \ M_2 \ \textit{implies} \ \Omega \mid \Omega' \vdash (F_1 M_1) \ (\mathcal{R} \otimes \mathcal{R}')_\theta \ (F_2 M_2).$$

The last clause, concerning relational parametricity for functions, was used in a denotational setting by O'Hearn and Tennent [OT93a], but the more detailed way in which divergence is taken into account has been first incorporated by O'Hearn and Reynolds in translating IA (with active expressions) into the linear polymorphic lambda calculus [OR00].

The definition above is extended to open terms as follows:

Definition 2.8 *Given* $\mathcal{R} \in Rel(\Omega)$ *and terms* $\Omega \mid \Gamma \vdash P_i : \theta$, $i = 1, 2$, $\Gamma = \{x_1 : \theta_1, \ldots, x_n : \theta_n\}$, $\Omega \mid \Gamma \vdash P_1 \ \mathcal{R}_\theta \ P_2$ *is defined to hold if for all* $\mathcal{R}' \in Rel(\Omega')$ *with* Ω', Ω *disjoint and for all terms* $\Omega \mid \Omega' \vdash P_{i,j} : \theta_j$, $i = 1, 2$, $j = 1, n$ *then for all* $j, \Omega \mid \Omega' \vdash P_{1,j} \ (\mathcal{R} \otimes \mathcal{R}')_{\theta_j} \ P_{2,j}$ *implies*

$$\Omega \mid \Omega' \vdash P_1[x_1, x_2, \ldots, x_n / P_{1,1}, P_{1,2}, \ldots, P_{1,n}](\mathcal{R} \otimes \mathcal{R}')_\theta \ P_2[x_1, x_2, \ldots, x_n / P_{2,1}, P_{2,2}, \ldots, P_{2,n}].$$

The definition above reduces to Definition 2.7 on the facing page in the case $\Gamma = \emptyset$, because of the following weakening property of the parametric logical relations:

Lemma 2.2 (Weakening) *If* $\mathcal{R} \in Rel(\Omega)$ *and* $\mathcal{R}' \in Rel(\Omega')$, *with* Ω, Ω' *disjoint, and if* Γ, Γ' *are also disjoint then* $\Omega \mid \Gamma \vdash P_1 \ \mathcal{R}_\theta \ P_2$ *implies* $\Omega \mid \Omega' \mid \Gamma \mid \Gamma' \vdash P_1 \ (\mathcal{R} \otimes \mathcal{R}')_\theta \ P_2$.

In addition, $\Omega \mid \Gamma \vdash P_1 \ \mathcal{R}_\theta \ P_2$ *with* $\Gamma = \emptyset$ *holds if and only if* $\Omega \vdash P_1 \ \mathcal{R}_\theta \ P_2$.

Proof: Immediate by induction on θ, using associativity of $\otimes$ and the weakening property of evaluation in Lemma 2.4 on page 22. End Of Proof.

The parametric logical relation respects extensional equivalence:

Lemma 2.3 *If* $\Omega \vdash P_1 \ \mathcal{R}_\theta \ P_2$ *and* $\Omega \vdash P_i \cong_\theta P_i'$, *for* $i = 1, 2$, *then* $\Omega \vdash P_1' \ \mathcal{R}_\theta \ P_2'$.

Proof: Immediate from induction on the structure of θ, and the definitions 2.4 and 2.7 on the facing page. End Of Proof.

The main intermediate result is the following:

Proposition 2.1 *Parametric logical relations are preserved by the term-forming operations of IA:*

1. *For* $\mathbf{k} \in \{\mathbf{true}, \mathbf{false}, n, \mathbf{skip}\}, \emptyset \vdash \mathbf{k}\ (\mathcal{I}_\emptyset)_\sigma\ \mathbf{k}$, *where* σ *is, respectively,* **expbool**, **expint**, **comm***;*

2. *If* $v \in dom(\Omega)$ *then* $\Omega \vdash v\ (\mathcal{I}_\Omega)_{\mathbf{var}\sigma}\ v$*;*

3. *If* $\Omega \mid \Gamma \vdash B_1\ \mathcal{R}_{\mathbf{expbool}}\ B_2$ *and* $\Omega \mid \Gamma \vdash M_1\ \mathcal{R}_\sigma\ M_2$ *and* $\Omega \mid \Gamma \vdash M_1'\ \mathcal{R}_\sigma\ M_2'$ *then*
$$\Omega \mid \Gamma \vdash (\mathbf{if}\ B_1\ \mathbf{then}\ M_1\ \mathbf{else}\ M_1')\ \mathcal{R}_\sigma\ (\mathbf{if}\ B_2\ \mathbf{then}\ M_2\ \mathbf{else}\ M_2');$$

4. *If* $\Omega \mid \Gamma \vdash E_1\ \mathcal{R}_{\mathbf{exp}\tau}\ E_2$ *and* $\Omega \mid \Gamma \vdash E_1'\ \mathcal{R}_{\mathbf{exp}\tau}\ E_2'$ *then*
$$\Omega \mid \Gamma \vdash (E_1 \star E_1')\ \mathcal{R}_{\mathbf{exp}\tau'}\ (E_2 \star E_2'),\ \textit{for any operator}\ \star;$$

5. *If* $\Omega \mid \Gamma \vdash F_1\ \mathcal{R}_{\sigma\to\theta}\ F_2$ *and* $\Omega \mid \Gamma \vdash M_1\ \mathcal{R}_\sigma\ M_2$ *then* $\Omega \mid \Gamma \vdash (F_1 M_1)\ \mathcal{R}_\theta\ (F_2 M_2)$*;*

6. *If* $\Omega \mid \Gamma \vdash V_1\ \mathcal{R}_{\mathbf{var}\tau}\ V_2$ *then* $\Omega \mid \Gamma \vdash (!V_1)\ \mathcal{R}_{\mathbf{exp}\tau}\ (!V_2)$*;*

7. *If* $\Omega \mid \Gamma \vdash V_1\ \mathcal{R}_{\mathbf{var}\tau}\ V_2$ *and* $\Omega \mid \Gamma \vdash E_1\ \mathcal{R}_{\mathbf{exp}\tau}\ E_2$ *then*
$$\Omega \mid \Gamma \vdash (V_1 := E_1)\ \mathcal{R}_{\mathbf{comm}}\ (V_2 := E_2);$$

8. *If* $\Omega \mid \Gamma \vdash M_1\ \mathcal{R}_\sigma\ M_2$ *and* $\Omega \mid \Gamma \vdash C_1\ \mathcal{R}_{\mathbf{comm}}\ C_2$ *then*
$$\Omega \mid \Gamma \vdash (C_1; M_1)\ \mathcal{R}_\sigma\ (C_2; M_2);$$

9. *If* $\Omega \mid \Gamma \mid m : \sigma \vdash M_1\ \mathcal{R}_{\sigma'}\ M_2$ *then* $\Omega \mid \Gamma \vdash (\lambda m{:}\sigma.M_1)\ \mathcal{R}_{\sigma\to\sigma'}\ (\lambda m{:}\sigma.M_2)$*;*

10. *If* $\Omega \mid v : \mathbf{var}\tau \mid \Gamma \vdash M_1[v/v']\ (\mathcal{R} \otimes \mathcal{I}_{\{v\}})_\sigma\ M_2[v/v']$, *for some* $v \notin dom(\Omega)$, *then*
$$\Omega \mid \Gamma \vdash (\mathbf{new}\tau\ v'\ \mathbf{in}\ M_1)\ \mathcal{R}_\sigma\ (\mathbf{new}\tau\ v'\ \mathbf{in}\ M_2);$$

11. *If* $\Omega \mid \Gamma \vdash B_1\ \mathcal{R}_{\mathbf{expbool}}\ B_2$ *and* $\Omega \mid \Gamma \vdash C_1\ \mathcal{R}_{\mathbf{comm}}\ C_2$ *then*
$$\Omega \mid \Gamma \vdash (\mathbf{while}\ B_1\ \mathbf{do}\ C_1)\ \mathcal{R}_{\mathbf{comm}}\ (\mathbf{while}\ B_2\ \mathbf{do}\ C_2);$$

12. *If* $\Omega \mid \Gamma \vdash P_1\ \mathcal{R}_\theta\ P_2$ *and* $\Omega \mid \Gamma \mid m : \theta \vdash P_1'\ \mathcal{R}_{\theta'}\ P_2'$ *then*
$$\Omega \mid \Gamma \vdash (\mathbf{let}\ m\ \mathbf{be}\ P_1\ \mathbf{in}\ P_1')\ \mathcal{R}_{\theta'}\ (\mathbf{let}\ m\ \mathbf{be}\ P_2\ \mathbf{in}\ P_2').$$

PROOF:
The proof is virtually the same as in the case of IA without side effects in expressions. The restrictions on high-order functions do not raise any problems; for iterations the same line of reasoning applies as in the case of recursion in Pitts's original version.

With the appropriately modified Definitions 2.7 and 2.8 proving the property for active phrases is relatively straightforward; the cases of $\sigma = \mathbf{var}\tau, \mathbf{exp}\tau, \mathbf{comm}$ must all be considered. We only look in detail at the case $\sigma = \mathbf{expint}$, the sub-case not involving non-termination. All other cases have similar proofs.

We first consider the case when $\Gamma = \emptyset$; by Lemma 2.2 on page 19 we know that the definition of a parametric relation for open terms reduces to the parametric relation for semi-closed terms.

We have to show that

$$\Omega \vdash C_1\ \mathcal{R}_{\mathbf{comm}}\ C_2 \text{ and } \Omega \vdash E_1\ \mathcal{R}_{\mathbf{expint}}\ E_2 \text{ implies } \Omega \vdash (C_1; E_1)\ \mathcal{R}_{\mathbf{expint}}\ (C_2; E_2).$$

If the premises are true, then, by definition 2.7 on page 18:

$$\Omega \vdash s_1, C_1\ \Downarrow_{\mathbf{comm}}\ s_1', \mathbf{skip} \text{ and } \Omega \vdash s_2, C_2\ \Downarrow_{\mathbf{comm}}\ s_2', \mathbf{skip} \text{ and } (s_1, s_2) \in \mathcal{R}$$

implies $(s_1', s_2') \in \mathcal{R}$; also

$$\Omega \vdash s_1', E_1\ \Downarrow_{\mathbf{expint}}\ s_1'', n_1 \text{ and } \Omega \vdash s_2', E_2\ \Downarrow_{\mathbf{expint}}\ s_2'', n_2 \text{ and } (s_1, s_2) \in \mathcal{R}$$

implies $(s_1'', s_2'') \in \mathcal{R}$ and $n_1 = n_2$. But, according to the operational semantics of the language:

$$\frac{\Omega \vdash s_i, C_i\ \Downarrow_{\mathbf{comm}}\ s_1', \mathbf{skip} \qquad \Omega \vdash s_i', E_i\ \Downarrow_{\mathbf{expint}}\ s_i'', n_i}{\Omega \vdash s_i, C_i; E_i\ \Downarrow_{\mathbf{expint}}\ s_i'', n_i}$$

for $i = 1, 2$. If $(s_1, s_2) \in \mathcal{R}$ then, from the above, it follows that $(s_1'', s_2'') \in \mathcal{R}$ and $n_1 = n_2$ so, by definition, $\Omega \vdash (C_1; E_1)\ \mathcal{R}_{\mathbf{expint}}\ (C_2; E_2)$.

If the terms are not semi-closed then we use Definition 2.8 on page 19 along with the fact that substitution distributes over the sequential composition operator to immediately reduce the problem to semi-closed terms. END OF PROOF.

From this point, once the definition of parametric logical relations has been adjusted to fit phrases with side effects and once we have shown logical relations are preserved by the term-forming operations of IA with active expressions, the proof of Theorem 2.1 on page 17 follows exactly Pitts's argument, which relies only on Proposition 2.1 and Lemma 2.2, without using Lemma 2.1 (side-effect free expressions) or without assuming it in further definitions.

The details of the rest of the proof are as given in [Pit96] and will be omitted here. We will just re-state the following key theorem:

Theorem 2.2 (Fundamental property)

- *For any term* $\Omega \mid \Gamma \vdash P : \theta$, $\Omega \mid \Gamma \vdash P\ (\mathcal{I}_\Omega)_\theta\ P$.
- *If* $\Omega \mid \Gamma \mid \Gamma' \vdash P_1\ (\mathcal{I}_\Omega)_\theta\ P_2$ *then for all worlds* $\Omega' \supseteq \Omega$ *and semi-closed contexts* $C[-_\theta] : \theta'$ *with* $\mathit{Free}(C[-_\theta]) \subseteq \Omega'$ *and* $\Gamma \subseteq \mathit{Traps}(C[-_\theta])$,

$$\Omega' \mid \Gamma' \vdash C[P_1]\ (\mathcal{I}_{\Omega'})_{\theta'}\ C[P_2].$$

2.3 Equational reasoning

In this section we will consider several typical equivalences of IA that are interesting both because they capture important computational intuitions and because they illustrate the operationally-based proof techniques described in the previous section. Although the language here differs from the one used by Pitts, and although some of the technical definitions also differ, the proofs are almost identical to the ones in his paper.

These example equivalences are shown in detail also to defend the claim that the regular-language semantics offers significantly simpler techniques; therefore, one might want to contrast the proofs here with the regular-language-based proofs in Section 4.2. The examples presented here can also be proved using denotationally-based techniques, such as the O'Hearn and Reynolds translation into the polymorphic linear lambda calculus [OR00], but they are more complicated and require more background than the operationally-based proofs. What the two systems have in common is parametricity-based reasoning; this way of reasoning is also described and illustrated with more examples by Wadler [Wad89].

The following lemma is useful in proving equivalences:

Lemma 2.4

Equivariance. *Consider a bijection* $\beta : dom(\Omega) \to dom(\Omega')$, *and the substitution* $P[\beta]$*; given a state* $s \in States(\Omega)$, *by* $s[\beta]$, *we denote the state in* $States(\Omega')$ *mapping each variable* $v' \in dom(\Omega')$ *to* $s(\beta^{-1}v')$. *If* $\Omega \vdash s, P \Downarrow_\theta s', P'$ *then* $\Omega' \vdash s[\beta], P[\beta] \Downarrow_\theta s'[\beta], P'[\beta]$.

Determinacy. *If* $\Omega \vdash s, P \Downarrow_\theta s_i, P'_i$ *for* $i = 1, 2$ *then* $s_1 = s_2$ *and* $P'_1 = P'_2$ *up to* α*-equivalence.*

Weakening and strengthening. *Suppose that* $\Omega = \Omega' \mid \Omega''$ *with worlds* Ω', Ω'' *disjoint and states* $s' \in States(\Omega')$, $s'' \in States(\Omega'')$. *Given* $\Omega' \vdash P : \theta$, *then* $\Omega \vdash (s' \mid s''), P \Downarrow_\theta s_1, P'$ *if and only if* $\Omega' \vdash P' : \theta$ *and* $s_1 = (s'_1 \mid s'')$ *for some* s'_1 *so that* $\Omega' \vdash s', P \Downarrow_\theta s'_1, P'$.

The examples we are interested in are the following.

Example 2.1 (Meyer-Sieber [MS88, Example 1])

$$\Omega \mid \Gamma \vdash \textbf{newint } v \textbf{ in } C \equiv_{\textbf{comm}} C, \qquad \textit{where } v \notin \textit{Free}(C).$$

This most simple of equivalences reflects a *locality* principle. It fails in models of imperative computation relying on a global store, such as the original Scott and Strachey model [SS71]. It says that a globally defined procedure cannot modify a local variable. It was first proved using the "possible worlds" model of Reynolds and Oles, constructed using functor categories [Ole82].

PROOF: Using the operational extensionality theorem (Theorem 2.1) it is enough to show that for all worlds Ω and all semi-closed commands $\Omega \vdash C : \textbf{comm}$ and all states $s, s' \in$

$States(\Omega)$, $\Omega \vdash s, C \Downarrow_{\mathbf{comm}} s', \mathbf{skip}$ if and only if $\Omega \vdash s, \mathbf{newint}\ v\ \mathbf{in}\ C \Downarrow_{\mathbf{comm}} s', \mathbf{skip}$. The only way the second evaluation can be deduced is if

$$\Omega \mid v' : \mathbf{varint} \vdash (s \mid v' \mapsto 0), C[v/v'] \Downarrow_{\mathbf{comm}} (s' \mid v' \mapsto n), \mathbf{skip}.$$

For some $v' \notin \Omega$ and some $n \in \mathbb{Z}$. But since $v \notin Free(C)$, $C[v/v'] = C$, so, by the weakening property (Lemma 2.4 on the preceding page) the equation above holds if and only if $\Omega \vdash s, C \Downarrow_{\mathbf{comm}} s', \mathbf{skip}$, as required. END OF PROOF.

Example 2.2 (Meyer-Sieber [MS88, Example 3])

$$\Omega \mid \Gamma \vdash \begin{array}{l}\mathbf{newint}\ v_1\ \mathbf{in}\\ \quad\mathbf{newint}\ v_2\ \mathbf{in}\ C\end{array} \equiv_{\mathbf{comm}} \begin{array}{l}\mathbf{newint}\ v_2\ \mathbf{in}\\ \quad\mathbf{newint}\ v_1\ \mathbf{in}\ C.\end{array}$$

The principle illustrated by this example is that of *non-observability* for locations. On the two sides of the equivalence, variables v_1 and v_2 intuitively represent different "locations" because they are "allocated" (on the stack) in different orders. However, no command C should be able to distinguish between the two.
PROOF: The argument is similar to the previous example, using the equivariance property. END OF PROOF.

Example 2.3 (O'Hearn-Reynolds [OR00, Section 7.1])

$$f : \mathbf{comm} \to \mathbf{comm} \vdash \begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ \quad v := 0; f(v := 1);\\ \quad \mathbf{if}\ !v = 1\ \mathbf{then\ diverge\ else\ skip}\end{array} \equiv_{\mathbf{comm}} f(\mathbf{diverge}).$$

This example captures the intuition that changes to the state are in some way irreversible. A procedure executing an argument which is a command inflicts upon the state changes that cannot be undone from within the procedure. This is why if procedure f uses its argument both sides will fail to terminate; if procedure f does not use its argument the behaviour of each side will be identical because of the locality of x, as seen before.

Programming languages not obeying this principle need a *snapback* operator to save then restore state [Sie94]. While such an operation is not impossible, it cannot have the kind of efficient implementation we have come to associate with imperative programming.

The first model to address this issue correctly was O'Hearn and Reynolds's interpretation of IA using the polymorphic linear lambda calculus [OR00]. Reddy also addressed this issue using a novel "object semantics" approach [Red96], but in a particular flavour of IA known as interference-controlled ALGOL [OP$^+$99]. A further development of this model, that also satisfies this equivalence, is O'Hearn and Reddy's [OR95a], a model fully abstract for the second order subset.
PROOF:
Let us abbreviate the left-hand side term as C_1 and the right-hand side term as C_2.

Using operational extensionality (Theorem 2.1) we need to show that for all procedures $\Omega \vdash F : \mathbf{comm} \to \mathbf{comm}$ and states $s, s' \in \mathit{States}(\Omega)$:

$$\Omega \vdash s, C_1[f/F] \Downarrow_{\mathbf{comm}} s', \mathbf{skip} \text{ iff } \Omega \vdash s, C_2[f/F] \Downarrow_{\mathbf{comm}} s', \mathbf{skip}.$$

Using the operational semantic rules in Figure 2.2 on page 16, $\Omega \vdash s, C_1[f/F] \Downarrow_{\mathbf{comm}} s', \mathbf{skip}$ iff for some $v' \notin \Omega$ and $n \neq 1$:

$$\Omega \mid v' : \mathbf{varint} \vdash (s \mid v' \mapsto 0), F(v' := 1) \Downarrow_{\mathbf{comm}} (s' \mid v' \mapsto n), \mathbf{skip}. \tag{2.1}$$

Since $v' \notin \Omega$, by the weakening property, $\Omega \vdash s, C_2[f/F] \Downarrow_{\mathbf{comm}} s', \mathbf{skip}$ is true iff:

$$\Omega \mid v' : \mathbf{varint} \vdash (s \mid v' \mapsto 0), F(\mathbf{diverge}) \Downarrow_{\mathbf{comm}} (s' \mid v' \mapsto n), \mathbf{skip}. \tag{2.2}$$

We now need to prove that 2.1 holds iff 2.2 holds.

Define the following relation $\mathcal{R} \in \mathit{Rel}(\{v'\})$:

$$\mathcal{R} = \{(\bot, \bot)\} \cup \{(s_1, s_2) \mid s_1(v') = 0 = s_2(v')\} \cup \{(s_1, s_2) \mid s_1(v') = 1,\ s_2 = \bot\}.$$

Directly from definition of logical relations (Def. 2.7 on page 18) it follows that

$$\Omega \mid v' : \mathbf{varint} \vdash v' := 1\ (\mathcal{I}_\Omega \otimes \mathcal{R})_{\mathbf{comm}}\ \mathbf{diverge}.$$

From the fundamental property (Theorem 2.2 on page 21), $\Omega \vdash F\ (\mathcal{I}_\Omega)\ F$, so by definition of logical relation at type $\mathbf{comm} \to \mathbf{comm}$ we have:

$$\Omega \mid v' : \mathbf{varint} \vdash F(v' := 1)\ (\mathcal{I}_\Omega \otimes \mathcal{R})_{\mathbf{comm}}\ F(\mathbf{diverge}). \tag{2.3}$$

From our definition of $\mathcal{R}$, for any states $s' \in \mathit{States}(\Omega)$ and $s_2 \in \mathit{States}(\Omega)_\bot$:

$$\big((s' \mid v' \mapsto n), s_2\big) \in (\mathcal{I}_\Omega \otimes \mathcal{R}) \iff n = 0,\ s_2 = (s' \mid v' \mapsto 0) \text{ or } n = 1,\ s_2 = \bot. \tag{2.4}$$

From equations 2.3 and 2.4 for states $\big((s \mid v' \mapsto 0), (s \mid v' \mapsto 0)\big) \in (\mathcal{I}_\Omega \otimes \mathcal{R})$; if equation 2.1 holds then it is not possible that $\Omega \mid v' : \mathbf{varint} \vdash (s \mid v' \mapsto 0), F(\mathbf{diverge}) \Uparrow_{\mathbf{comm}}$, since it would require $\big((s' \mid v \mapsto n_{\neq 1}), \bot\big) \in (\mathcal{I}_\Omega \otimes \mathcal{R})$, contradicting equation 2.4.

It follows that $\Omega \mid v' : \mathbf{varint} \vdash (s \mid v' \mapsto 0), F(\mathbf{diverge}) \Downarrow_{\mathbf{comm}} s_2, \mathbf{skip}$, for some s_2. Equation 2.4 then implies $s_2 = (s' \mid v' \mapsto 0)$, thus equation 2.2 holds.

This shows that 2.1 implies 2.2. The converse is shown using a similar argument.

END OF PROOF.

The examples seen so far illustrate quite well the operationally-based proof techniques; other proofs are similar. Proofs in the denotational models [OT93a, OR00] are also similar. The key step in the proof is the discovery of a parametric relation, extending the identity relation. Note that the ability to relate states with non-termination in the logical relations is critical in the proof of the snapback example above. The lack of such a mechanism makes the equivalence fail in the original denotational parametric model of O'Hearn and Tennent [OT93a].

The following examples will be given only for their intuitive or historical importance, without proofs.

Example 2.4 (Stoughton [MS88, Example 5])

$$f : \mathbf{comm} \to \mathbf{comm} \vdash \begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ \quad v := 0; f(v := !v + 2);\\ \quad \mathbf{if}\ !v\ \mathbf{mod}\ 2 = 0\ \mathbf{then\ diverge\ else\ skip}\end{array} \equiv \mathbf{diverge}.$$

The principle illustrated here is that of *invariant preservation*: although procedure f has read and write access to variable v, that access is only through command $v := !v + 2$, so it can only be incremented by two. Therefore, variable v will always hold an even value.

Example 2.5 (Oles [Ole82, adapted])

$$f : \mathbf{comm} \to \mathbf{expbool} \to \mathbf{comm} \vdash \begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ \quad v := 0;\\ \quad f(v := 1, !v = 0)\end{array} \equiv_{\mathbf{comm}} \begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ \quad v := 0;\\ \quad f(v := -1, !v = 0)\end{array}$$

The two sides of the equivalence represent two possible implementation of a *switch* object. The switch is initially in the "on" state. The first argument is a "method" that changes the state of the switch to "off"; the second one returns a boolean expressions representing the state. The first implementation changes the state by assigning one, the second by assigning negative one. The behaviour of the two implementations should, however, be identical, illustrating a principle of *representation independence*. Although Oles does not prove this example, it can be proved using his possible-worlds model.

Example 2.6 (O'Hearn-Tennent [OT93a])

$$f : \mathbf{comm} \to \mathbf{expbool} \to \mathbf{comm} \vdash$$
$$\begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ \quad v := 0;\\ \quad f(v := 1, !v = 0)\end{array} \equiv_{\mathbf{comm}} \begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ \quad v := 0;\\ \quad f(v := !v + 1, !v = 0)\end{array}$$

This example is similar to that before, but the two implementations differ more substantially. In the second implementation, repeated application of the switch method does change the state, but it changes it in such a way that the second method continues to see it as being in the "off" position. This example cannot be validated using the original possible worlds model, but it can be validated using O'Hearn and Tennent's parametric model. This stronger idea of representation independence is an example of *parametricity.*

In our restricted language, this example may actually fail depending on how arithmetic over the finite data set is handled. If overflow leads to abnormal termination then the right-hand-side will eventually terminate abnormally. If overflow is handled by "wrap-around," (i.e. arithmetic is modulo the maximum value) then the equivalence fails again, as the ever-increasing variable will eventually reach 0 again. But if special values are used, then the two sides remain equivalent. Once the "infinity" value is reached, incrementing it further will not change it. Further discussion of this issue is given on page 59.

Interestingly enough, most of the difficult equivalences of IA without side-effects in expressions are also equivalent in IA with active expressions. However, a large class of equivalences of the former fail in the latter. Most importantly, most equivalences arising out of mathematical or logical properties fail because of the introduction of side-effects. The following are only two such non-equivalences:

Example 2.7 (Failed equivalences)

$$\Gamma \vdash E + E' \not\equiv_{\mathbf{expint}} E' + E$$
$$\Gamma \vdash \mathbf{if}\ E = E\ \mathbf{then\ diverge\ else\ skip} \not\equiv_{\mathbf{comm}} \mathbf{diverge}$$

The final observation in this section has to do with the distinction between *equivalence* and *equality* in the presence of side-effects. Pitts's Lemma on side-effect free expressions identifies these two notions in the absence of side-effects but, as it turns out, this identification is not crucial in the larger scheme because the definitions and the proofs can be suitably adapted to deal with side-effects. The Operational Extensionality Theorem does not make any reference, in fact, to equality entailing equivalence. Equivalence is the stronger property, and it is preserved by substitution even in the presence of side-effects.

But the fact that equality no longer implies contextual equivalence has important consequences on the way we can reason about the programming language, since we can no longer substitute in context *equal* phrases. Consider the following procedure, which diverges if its arguments are equal:

$$eq : \mathbf{expint} \to \mathbf{expint} \to \mathbf{comm}$$
$$\stackrel{\text{def}}{=} \lambda e{:}\mathbf{expint}.\lambda e'{:}\mathbf{expint}.\mathbf{if}\ e = e'\ \mathbf{then\ diverge\ else\ skip}$$

The following remark illustrates the fact that in the presence of side-effects replacing an expression by another, which is equal, does not necessarily result in a program equivalent to the original program.

Remark 2.1 *If*

$$\Gamma \mid e_0 : \mathbf{expint} \vdash P \equiv_\theta P' \quad \textit{and} \quad \Gamma \vdash eq\, E_0\, E_0' \equiv_{\mathbf{comm}} \mathbf{diverge},$$

it does not follow that:

$$\Gamma \vdash P[e_0/E_0] \equiv_\theta P'[e_0/E_0']$$

So, although in general we can always substitute equivalent phrases in equivalent phrases preserving the equivalence, we may not substitute *equal* phrases in equivalent phrases and expect to preserve equivalence.

PROOF: Any two identical phrases P, P' consisting just of identifier e_0, and equal expressions, with different side-effects, $E_i \stackrel{\text{def}}{=} v := i; 0$, it is obvious that $eq\, E_0\, E_1 \equiv \mathbf{diverge}$ but $E_0 \not\equiv E_1$. END OF PROOF.

This is not the case in IA without active expressions, because equality implies equivalence:

Proposition 2.2 *In the absence of side-effects, if* $\Gamma \vdash E_i \not\equiv_{\mathbf{expint}} \mathbf{diverge_{int}}$*,* $i = 1, 2$*.*

$$\Gamma \vdash eq(E_1)(E_2) \equiv \mathbf{diverge_{comm}} \Longrightarrow \Gamma \vdash E_1 \equiv_{\mathbf{expint}} E_2,$$

where $\mathbf{diverge_{int}} \stackrel{def}{=} \mathbf{diverge}; 0$.

PROOF: Immediate from Pitts's side-effect free expression lemma (Lemma 2.2 in [Pit96]) and the operational extensionality theorem. END OF PROOF.

2.4 Specification logic

The specification language introduced in Chapter 5 is inspired by Reynolds's specification logic for IA (Specification Logic). However, the similarities are superficial, as the absence of side-effects in expressions plays a key role in Specification Logic. We have already mentioned in the Introduction that it is a commonly held belief that side-effects in expressions are incompatible with proving program correctness. In the presence of side-effects, the inability to substitute equals for equals has the logical consequence that substitution of equals may not preserve validity of a specification. This makes "static" (mathematical and logical) reasoning virtually impossible.

Although there have been attempts to incorporate active expressions in a programming logic, for example Boehm's programming logic [Boe82, Boe85], these logics did not become popular. One can speculate that their lack of success was due to the rather peculiar syntax or semantics they used. For example, Boehm confines the side-effects of a phrase using an $\langle E \rangle$ operator on expressions with "snap-back" semantics: evaluate E, restore the state to whatever it was *before* starting evaluation, then produce the value. For example, the assertion: $(\langle v := 3; \mathbf{true} \rangle \ \mathbf{and}\ \langle !v \rangle = 3)$ is equivalent not to **true** but to $(\langle !v \rangle = 3)$. We will not follow this approach.

Another, more profound, problem also discussed in the Introduction is that computational effects (division by zero, overflow, non-termination) may occur in expressions even in the absence of side-effects through assignment. Several ways around this problem have been proposed [BCJ84]; [Ten87] gives a brief outline of the problem and a simple, common-sense solution.

We will give a brief and informal introduction to Specification Logic because we share the same methodological objectives. However, the fundamental semantic ideas will be seen to be substantially different from ours.

Specification Logic builds on the familiar Hoare specification logic of the simple imperative language [Hoa69], incorporating call-by-name procedures and (side-effect-free) functions. Assertions are formulas about properties of states. They are similar to boolean expressions, but they include additional notations, such as quantifiers. The type of assertions is **assert**; the assertion-forming expressions are given in Figure 2.3 ($\star$ ranges over the standard set of boolean operators).

Specifications are formulas about properties of programs. The atomic specification is the Hoare triple (pre-condition, program, post-condition), to which we add logical

$$\frac{\Gamma \vdash B : \mathbf{expbool}}{\Gamma \vdash B : \mathbf{assert}} \qquad \frac{\Gamma \vdash P : \mathbf{assert} \qquad \Gamma \vdash P' : \mathbf{assert}}{\Gamma \vdash P \star P' : \mathbf{assert}}$$

$$\frac{\Gamma \mid x : \theta \vdash A : \mathbf{assert}}{\Gamma \vdash \forall x : \theta . A : \mathbf{assert}} \qquad \frac{\Gamma \mid x : \theta \vdash A : \mathbf{assert}}{\Gamma \vdash \exists x : \theta . A : \mathbf{assert}}$$

$$\frac{\Gamma \vdash A : \mathbf{assert} \qquad \Gamma \vdash C : \mathbf{comm} \qquad \Gamma \vdash A' : \mathbf{assert}}{\Gamma \vdash \{A\}\ C\ \{A'\} : \mathbf{spec}}$$

$$\frac{\Gamma \vdash A : \mathbf{assert}}{\Gamma \vdash \{A\} : \mathbf{spec}} \qquad \frac{\Gamma \vdash S : \mathbf{spec} \qquad \Gamma \vdash S' : \mathbf{spec}}{\Gamma \vdash S \star S' : \mathbf{spec}} \qquad \frac{\Gamma \mid x : \theta \vdash S : \mathbf{spec}}{\Gamma \vdash \forall x : \theta . S : \mathbf{spec}}$$

Figure 2.3: Syntax of assertions and specifications

$$\{A\}\ \mathbf{skip}\ \{A\} \qquad \{P(E)\}\ V := E\ \{P(!V)\}$$

$$\frac{\{A\}\ C_0\ \{A'\} \qquad \{A'\}\ C_1\ \{A''\}}{\{A\}\ C_0; C_1\ \{A''\}}$$

$$\frac{\{A\ \mathbf{and}\ B\}\ C_0\ \{A'\} \qquad \{A\ \mathbf{and\ not}\ B\}\ C_1\ \{A'\}}{\{A\}\ \mathbf{if}\ B\ \mathbf{then}\ C_0\ \mathbf{else}\ C_1\ \{A'\}}$$

$$\frac{\{A\ \mathbf{and}\ B\}\ C\ \{A\}}{\{A\}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \{A\ \mathbf{and\ not}\ B\}}$$

$$\frac{\{A_0\ \mathbf{implies}\ A_1\} \qquad \{A_1\}\ C\ \{A_2\}}{\{A_0\}\ C\ \{A_2\}} \qquad \frac{\{A_0\}\ C\ \{A_1\} \qquad \{A_1\ \mathbf{implies}\ A_2\}}{\{A_0\}\ C\ \{A_2\}}$$

Figure 2.4: Hoare-like inference rules

connectives and quantifiers. The type of specification is **spec** and their syntax is given in Figure 2.3. (the operator $\star$ stands for conjunction $\wedge$ or implication $\Rightarrow$.)

Some of the typical partial-correctness Hoare-like inference rules for specifications are given in Figure 2.4. Additionally, and importantly, specifications and their connectives form an intuitionistic logic, therefore the standard inference rules apply for specification connectives and quantifiers.

The inadequacy of Hoare's logical framework in the presence of procedures is illustrated by the fact that the assignment axiom no longer holds in a standard semantic model:

$$\{P(E)\}v := E\{P(!v)\}.$$

An obvious counter-example is $P \stackrel{\text{def}}{=} (\lambda e.e = !v + 1)$ and $E \stackrel{\text{def}}{=} (!v + 1)$; in Reynolds's terminology $v := !v + 1$ is a command that *interferes* with $\lambda e.e = !v + 1$.

Reynolds was not the first to give a programming logic for procedures; Hoare's [Hoa71] is the earliest attempt, but Reynolds's approach was more general and systematic. The basic idea is to introduce two new atomic specifications, in addition to the Hoare triple, non-interference and good variables:

TYPING RULES

$$\frac{\Gamma \vdash P : \theta \qquad \Gamma \vdash P' : \theta'}{\Gamma \vdash P \# P' : \mathbf{spec}} \qquad \frac{\Gamma \vdash V : \mathbf{var}\tau}{\Gamma \vdash \mathbf{gv}_\tau(V) : \mathbf{spec}}$$

The informal semantics of non-interference is that executing P, in any way, does not change the way P' evaluates. For example, if P is a command and P' a phrase, executing P does not change the value produced by P'. But the mathematical semantics required to capture this simple intuition is subtle, because phrases P, P' could involve higher-order procedures, which interact with the store in non-obvious ways. Informally, a good variable is a variable that immediately following an assignment reads back the same value that it was assigned to.

The denotational model of Specification Logic was first discovered by Tennent [Ten90] then refined by O'Hearn and Tennent [OT93b]. They observed that in order to have a powerful enough notion of non-interference between two phrases $P \# P'$ it is not enough that the execution of P does not change the values of P', but *at any intermediate stage in the execution of P the value of P' must remain the same.*[1]

Using non-interference and good-variable assumptions, Hoare-like axioms can be formulated. Some of them are given in Figure 2.5 on the following page[2].

The *Non-interference Decomposition* rule captures the property that there are no "anonymous" (such as through global variables) sources of interference. The *Constancy* rule allows an assertion (A) to be assumed whenever necessary in a proof. With *Non-interference decomposition* it is possible to assume that if a command (C) does not interfere with an expression (E) in reasoning about a specification ($\{A\}\ C\ \{A'\}$) then it is safe to assume that no phrase (P) interferes with that expression. The *Assignment* rule is the usual Hoare-like assignment rule, but with the proper non-interference and good-variable conditions. The *Local variable* rule allows discharging of all good-variable and non-interference assumption involving the variable becoming local (v). The *Non-interference abstraction* gives an invariant-like rule for programs with non-locally defined procedures.

Finally, there is a rule for reasoning about programs with non-local procedures and discharging the assumptions when the procedure is introduced:

[1] An operational model of non-interference has not yet been developed. Formulating this condition operationally raises technical difficulties. The author has been researching this topic separately [Ghi02].

[2] We present them in natural-deduction style for clarity. Reynolds's original presentation is in axiomatic style.

Non-interference decomposition:

$$\frac{\Gamma \vdash x \# x'}{\Gamma \vdash P \# P'} \text{ for all } x \in \mathit{Free}(P), x' \in \mathit{Free}(P')$$

Constancy:

$$\frac{\Gamma \vdash C \# A \qquad \Gamma \vdash \{A\} \Rightarrow \{A_0\}C\{A_1\}}{\Gamma \vdash \{A_0 \textbf{ and } A\}C\{A_1 \textbf{ and } A\}}$$

Non-interference composition

$$\frac{\Gamma \vdash C \# E \qquad \Gamma \vdash P \# E \Rightarrow \{A\}\ C\ \{A'\}}{\Gamma \vdash \{A\}\ C\ \{A'\}}$$

Assignment

$$\frac{\Gamma \mid e_0 : \textbf{expint} \vdash \textbf{gv}_\tau(V) \qquad \Gamma \mid e_0 : \textbf{expint} \vdash V \# P}{\Gamma \vdash \{P(E)\}\ V := E\ \{P(!V)\}}$$

Local variable declaration

$$\frac{\Gamma \vdash \forall v : \textbf{var}\tau.\textbf{gv}_\tau(v) \wedge v \# P_i \wedge \cdots \wedge P_j \# v \wedge \cdots \Rightarrow \{A\}\ C\ \{A'\}}{\Gamma \vdash \{A\}\ \textbf{new}\tau\ v\ \textbf{in}\ C\ \{A'\}}$$

Non-interference abstraction (simple version)

$$\frac{\Gamma \vdash F \# A \qquad \Gamma \vdash \{A\}\ C\ \{A\}}{\Gamma \vdash \{A\}\ F(C)\ \{A\}}$$

Figure 2.5: Selected specification logic inference rules

INFERENCE RULE

$$\dfrac{\Gamma \vdash S' \wedge S_{\text{par}} \wedge S_{\text{proc}} \Rightarrow \{A_{\text{proc}}\}\ P_{\text{proc}}\ \{A'_{\text{proc}}\} \qquad \Gamma \vdash S \wedge S'' \wedge S_{\text{proc}} \Rightarrow \{A\}\ P\ \{A'\}}{\Gamma \vdash S' \wedge S'' \Rightarrow \{A\}\ \textbf{let}\ f\ \textbf{be}\ \lambda m{:}\sigma.P_{\text{proc}}\ \textbf{in}\ P\ \{A'\}}$$

where

$$S_{\text{proc}} \stackrel{\text{def}}{=} \forall m : \sigma.S_{\text{par}} \Rightarrow \{A_{\text{proc}}\}\ f(m)\ \{A'_{\text{proc}}\} \wedge \forall e : \textbf{expint}.\cdots x_i \ \#\ e \cdots \Rightarrow f \ \#\ e$$

and $m \notin \mathit{Free}(s')$, $f \notin \mathit{Free}(A_{\text{proc}}, A'_{\text{proc}}, A, A', S', S'', S_{\text{par}})$.

Informally, what this rule means is that if program P satisfies specification $\{A\}\ P\ \{A'\}$, subject to some assumptions about a non-local procedure f, and a phrase P_{proc} meets those assumptions, then P_{proc} can be used as an implementation for f, and the assumptions can be discharged.

We must also point out that Specification Logic remains an intuitionistic first-order logic, and the usual inference rules for connectives and quantifiers apply.

O'Hearn showed how some of the examples from the previous section can be also proved using Specification Logic [O'H90, Section 5.4]. For example in order to prove Example 2.4 on page 25 we need to prove that $f : \textbf{comm} \rightarrow \textbf{comm} \vdash \{\textbf{true}\}\ C\ \{\textbf{false}\}$, where $C \stackrel{\text{def}}{=} \textbf{newint}\ v\ \textbf{in}\ v := 0; f(v := !v{+}2); \textbf{if}\ !v\ \textbf{mod}\ 2 = 0\ \textbf{then diverge else skip}$.

PROOF: The first part of the derivation tree is:

$$\dfrac{\dfrac{\dfrac{[f \ \#\ v]}{f \ \#\ (!v\ \textbf{mod}\ 2 = 0)} \qquad \dfrac{\dfrac{\dfrac{[\textbf{gv}(v)] \qquad \dfrac{[v \ \#\ e]}{v \ \#\ (e\ \textbf{mod}\ 2 = 0)}}{\{(e\ \textbf{mod}\ 2 = 0)[e/!v+2]\}\ v := !v+2\ \{(e\ \textbf{mod}\ 2 = 0)[e/!v]\}}}{\{(!v+2)\ \textbf{mod}\ 2 = 0\}\ v := !v+2\ \{!v\ \textbf{mod}\ 2 = 0\}}}{\{!v\ \textbf{mod}\ 2 = 0\}\ v := !v+2\ \{!v\ \textbf{mod}\ 2 = 0\}}}{\{!v\ \textbf{mod}\ 2 = 0\}\ f(v := !v+2)\ \{!v\ \textbf{mod}\ 2 = 0\}}\ [\text{Non-interf. abstr.}]}{f \ \#\ v \wedge \textbf{gv}(v) \wedge v \ \#\ e \Rightarrow \{!v\ \textbf{mod}\ 2 = 0\}\ f(v := !v+2)\ \{!v\ \textbf{mod}\ 2 = 0\}}$$

Separately we can easily derive the theorem that $\{B\}\ \textbf{if}\ B\ \textbf{then diverge else skip}\ \{\textbf{false}\}$ with any boolean B, which we can instantiate with $(!v\ \textbf{mod}\ 2 = 0)$.

Using the rule for sequential composition we can easily show that:

$$f \ \#\ v \wedge \textbf{gv}(v) \wedge v \ \#\ e \Rightarrow \{\textbf{true}\}\ C'\ \{\textbf{false}\}.$$

where $C' = v := 0; f(v := !v + 2); \textbf{if}\ !v\ \textbf{mod}\ 2 = 0\ \textbf{then diverge else skip}$. The assumptions are discharged using the rule for local variable declaration:

$$\dfrac{f \ \#\ v \wedge \textbf{gv}(v) \wedge v \ \#\ e \Rightarrow \{\textbf{true}\}\ C'\ \{\textbf{false}\}}{\{\textbf{true}\}\ \textbf{newint}\ v\ \textbf{in}\ C'\ \{\textbf{false}\}}$$

with $C = \textbf{newint}\ v\ \textbf{in}\ C'$. END OF PROOF.

Chapter 3

Game Semantics of IA

How should we explain to someone what a game is? I imagine that we should describe games *to him, and we might add: "This* and similar things *are called 'games'".*

Ludwig Wittgenstein

In this chapter we will consider the game model for IA. Technical details will be presented only insofar as they are required to understand the regular-language semantics of the following chapter and to prove the Representation Lemma (Lemma 4.2 on page 69) for the regular-language semantics. But the details needed to prove the full-abstraction theorem for the game model of IA will be omitted; for those details the reader is referred to [AM96].

There are several good introductory papers on game semantics [Abr96, McC97, AM, Jür]; this introduction to the game model of IA draws from all of them.

Traditionally, denotational models of logic, and of computation, were functional in nature. The meaning of a proposition was seen as a function from the values of its variables to its truth value; the meaning of a computation was seen as a function from its "inputs," in some generalized sense, to its "outputs." This abstract view leads to elegant mathematical models but it usually fails to deal properly with situations where the *dynamics* of the system are important.

For example, a logical system with subtle dynamic properties is linear logic [Gir87], a logic that restricts the structural properties of weakening and contraction in its formal proof system. This effectively corresponds to a view of propositions as "resources" that cannot be arbitrarily reused. Even in ordinary circumstances this idea sometimes makes sense. Using contraction one can prove, in standard propositional logic, that $A \wedge A \rightarrow A$. But if the proposition A refers to a resource (e.g. *Mary has one dollar.*), then two distinct natural readings of the conjunction arise, the "resource-sensitive":

If Mary has one dollar and Mary has one dollar then Mary has two dollars.

as well as the "classical":

If Mary has one dollar and Mary has one dollar then Mary has one dollar.

Linear logic distinguishes between these two types of conjunctions.

Blass [Bla92] gave the first model of linear logic using games. Abramsky and Jagadeesan [AJ92] gave the first complete model of linear logic, also using game models.

Programming languages also have subtle dynamic properties. For example the language PCF [Plo77], a streamlined functional language, raised the issue of *sequential evaluation*, i.e. that a function must evaluate its arguments sequentially. This is why operators such as *parallel-or (por)* are not definable in the language:

$$por(b, b') = \begin{cases} \mathit{true} & \text{if } b = \mathit{true} \\ \mathit{true} & \text{if } b' = \mathit{true} \\ \mathit{false} & \text{if } b = b' = \mathit{false} \\ \mathit{undefined} & \text{if } b \text{ is undefined or } b' \text{ is undefined.} \end{cases}$$

The operator *por* is not a constant function, and it requires the two arguments to be executed in parallel (or interleaved) fashion. Observe that if the arguments are executed sequentially and deterministically then if the first argument is executed first then $por(\mathit{undefined}, \mathit{true}) = \mathit{undefined}$, as opposed to *true*; if the second argument is executed first then $por(\mathit{true}, \mathit{undefined}) = \mathit{undefined}$, as opposed to *true*.

Independently, Hyland and Ong [HO00], Abramsky, Jagadeesan and Malacaria [AMJ94] and Nickau [Nic94] gave fully abstract models of PCF, using game models. Shortly after, O'Hearn and Riecke independently gave a traditional, domain-based, fully abstract model of PCF [OR95b].

3.1 Lorenzen games

Before presenting game models of computation let us briefly consider game models of logic. They are simpler than the former, and the metaphor of interpreting a proposition as a game between a verifier and a falsifier is quite intuitive.

A logical formula is seen as a (dialogue) game between two opponents: a verifier (V) and a falsifier (F). F is attacking a formula, and V defends against the attack. Suppose we want to interpret a propositional logic with the following formulas:

$$A ::= A \wedge A' \mid A \vee A' \mid \neg A \mid \phi,$$

where ϕ ranges over a set of atomic facts. The rules of the game are dictated by the syntax of the formula:

$A \wedge A'$: V wins if it wins in both of the two components; F wins if it wins in at least one of the two components.

$A \vee A'$: V wins if it wins in at least one of the two components; F wins if it wins both components.

$\neg A$: V wins if and only if F wins at A.

ϕ: if ϕ is true, then V wins; if ϕ is false then F wins.

Define $A \to B \stackrel{\text{def}}{=} \neg A \vee B$.

A proposition is true if V has a winning strategy, i.e. it can win *any game*, regardless of how F plays. A proposition is false if F has a winning strategy.

Example 3.1

$$A \to A \textit{ is always true.}$$

PROOF: We need to prove that:

$$A \vee \neg A \text{ is always true.}$$

V has a winning strategy: let F play and, if it wins, mimic its behaviour in the other component and, given the reversal of the roles by negation, win in that component. Therefore, V is guaranteed to win in one of the components and, by definition of disjunction, the game itself.

A typical play of this strategy is described in Figure 3.1.

END OF PROOF.

A	$\vee$	$\neg$	A	Interpretation
	$?_{\mathsf{F}}$			F attacks the formula
$?_{\mathsf{V}}$				V defends one of the components
$\vdots$				play within A
F				F wins the game of A
		$?_{\mathsf{V}}$		V defends the other component
			$?_{\mathsf{F}}$	F defends A
			$\vdots$	the *same* play within A
			F	F wins the game of A, as before
		V		...so V wins the negation game
	V			V wins, having won at least one component

Figure 3.1: A play in $A \to A$.

The example in Figure 3.1 assumes that F plays the same in the second game for A. But what if that is not the case? It turns out that V still has a winning strategy, as informally described in Figure 3.2 on the next page.

This strategy of V to repeat the same play between components is called a *copy-cat strategy.* The importance of this strategy in game semantics cannot be over-emphasized. The ability of V to switch the play between components is characteristic of classical logic.

A	$\vee$	$\neg$	A	Interpretation
	$?_{\mathsf{F}}$			F attacks the formula
		$?_{\mathsf{V}}$		V defends second component
			$?_{\mathsf{F}}$	F defends A
$?_{\mathsf{V}}$				V defends the other A
			m_1	F makes a move in one component
m_1				V replies with the same move in the other component
			m_2	
m_2				V keeps mimicking F's moves, in the opposite component
$\vdots$			$\vdots$	
			V	if V wins this game...
V				...it also wins this game
		F		F wins this game
	V			but V wins the overall game.
F				if F wins this game...
			F	... it must also win this game...
		V		... and lose this game
	V			so V wins the overall game.

Figure 3.2: The winning strategy of V in $A \to A$.

Example 3.2

$$A \wedge A \to A \textit{ is always true.}$$

PROOF: The winning strategy of V is to play copy-cat between the first two occurrences of A and the last one. Typical plays according to this strategy are given in Figure 3.3 on the following page. END OF PROOF.

Notice the following asymmetric behaviour in the second play of Figure 3.3:

- V attempts to defend the second conjunct although it has lost the first one. This is futile, because even if V wins it will have lost the conjunct as a whole, having lost one of the components;

- at the end of the game, having won one of the disjuncts, V does not attempt to defend the second disjunct, because it has won the overall game already.

This mix of "eager" and "lazy" behaviour does not matter as far as the overall outcome of a game of classical logic is concerned. But it becomes important in the case of a resource-sensitive logic, such as linear logic. Blass [Bla92] applies this distinction in his interpretation of linear logic, by distinguishing between operator variants in which *both* components must be played and in which *exactly one* of the components must be played. They correspond respectively to the multiplicative and the additive operators of linear logic.

3.2 Hyland–Ong games

Before giving the formal definitions for the game model we will try to build an intuitive background for the key concepts involved.

The transition from the Lorenzen game semantics of logic to the game semantics of programming languages uses the Curry-Howard isomorphism: the idea is to understand types as propositions and lambda terms as proofs as, i.e. types are games and terms are strategies. We adapt the rest of the concepts as follows.

Instead of a verifier and a falsifier the game is played between a *player* (P), which represents the term, and an *opponent* (O), which represents its environment (all non-local identifiers). Instead of attacks and defenses, the actions are questions and answers. Abramsky gives the following helpful intuitive interpretation of a game as an interaction [Abr96] (also see Figure 3.4):

O question (q_O):	request for output;	P question (q_P):	request for input;
O answer (a_O):	input;	P answer (a_P):	output.

Each game starts with an O question and it ends when P provides the answer. There is no intuitive notion of winning or losing such a game.

$\neg$	$(A$	$\wedge$	$A)$	$\vee$	A	Interpretation
				$?_{\mathsf{F}}$		F attacks disjunction
$?_{\mathsf{V}}$						V defends negation
		$?_{\mathsf{F}}$				F attacks conjunction
	$?_{\mathsf{V}}$					V defends A
	$\vdots$					play in A
	V					V wins
			$?_{\mathsf{V}}$			V defends the other conjunct
			$\vdots$			same play in A
			V			V wins again
		V				so V wins conjunction
F						so F wins negation
					$?_{\mathsf{V}}$	V defends A
					$\vdots$	same play in A
					V	V wins again
				V		V wins disjunction

$\neg$	$(A$	$\wedge$	$A)$	$\vee$	A	Interpretation
				$?_{\mathsf{F}}$		F attacks disjunction
$?_{\mathsf{V}}$						V defends negation
		$?_{\mathsf{F}}$				F attacks conjunction
	$?_{\mathsf{V}}$					V defends A
	$\vdots$					play in A
	F					F wins
			$?_{\mathsf{V}}$			V defends the other conjunct
			$\vdots$			play in A
			F			F wins again
		F				F wins the conjunct
V						V wins the negation
				V		V wins the disjunct

Figure 3.3: Some plays in $A \wedge A \to A$.

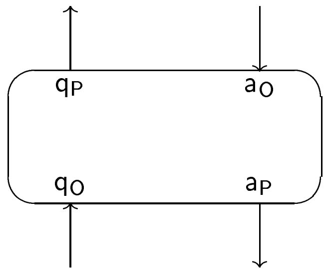


Figure 3.4: Games as interactions

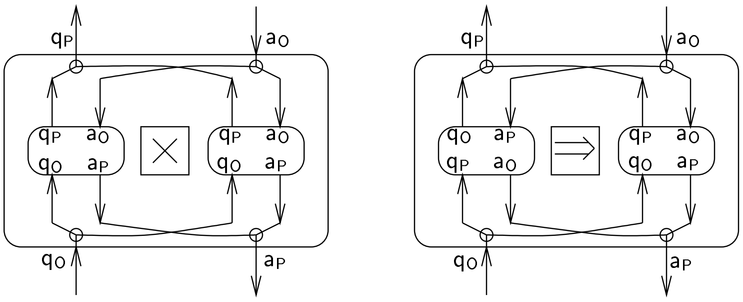


Figure 3.5: Composite interactions

If we take an interactive view of games, then the games for product and function types are constructed as follows. For the product $A \times B$, put the two type-components side-by side and connect the inputs and the outputs. The two components, A and B, do not handle the interactions in an arbitrary way, but their behaviours are synchronized according to a protocol, which will be defined later.

For the function type $A \to B$ first the player-opponent polarity of A's interactions is reversed, then the two components are interconnected as in Figure 3.5; the way the two components handle input and output requests is also in accordance with a well defined protocol.

One of the simplest interesting games is the game for the natural numbers, consisting of one O question (q_{O}) and a set of P answers consisting of the natural numbers (n_{P}). In this simple game, P never asks questions (there are no requests for input) and, consequently, O never needs to answer (there is no input). Figure 3.6 on the following page depicts, using the interaction analogy, the structures of the games for $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N} \to \mathbb{N}$. For the function $\mathbb{N} \to \mathbb{N}$ the result component takes requests from the environ-

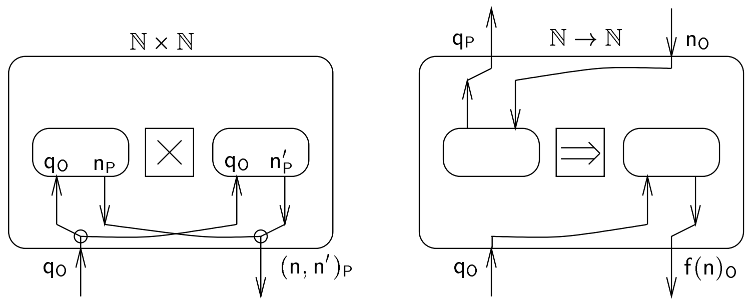


Figure 3.6: Structures of types built with $\mathbb{N}$.

ment and returns the answer, while the argument component is used to request from the environment a value for the argument to a function, then reads it as input.

Terms of a given type are interpreted as *strategies*, i.e. by the protocols governing the responses of the system to actions of the environment. Strategies define sets of plays, which can also be seen as structured traces of the actions of the system.

Let us now look, still in an informal but, one hopes, intuitive manner, at some plays for terms which belong to the types described:

- $3 : \mathbb{N}$ has the play:

	$\mathbb{N}$	*Interpretation*
O	q	there is a request for output
P	3	the system produces 3 as output.

- $(3, 5) : \mathbb{N} \times \mathbb{N}$ has the play:

	$\mathbb{N}$	$\times$	$\mathbb{N}$	*Interpretation*
O	q			there is a request for output
P	3			the system produces the first component of the pair
O			q	there is a second request for output
P			5	the system produces the second component.

  Notice that the play reflects a left-to-right evaluation of the two components.

- $\lambda n{:}\mathbb{N}.n + 3 : \mathbb{N} \to \mathbb{N}$ has the play:

	$\mathbb{N}$	$\to$	$\mathbb{N}$	*Interpretation*
O			q	there is a request for output
P	q			the system requests input: the value for n
O	n			some value n is input
P			$n + 3$	the value $n + 3$ is output as a result.

Let us now consider some more complex examples:

- $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ (curried addition):

	$\mathbb{N}$	$\to$	$\mathbb{N}$	$\to$	$\mathbb{N}$	*Interpretation*
O					q	initial request for output
P	q					request for input, the first argument
O	m					some value m provided as input
P			q			request for input, the second argument
O			n			some value n provided
P					$m+n$	final output.

  Again, notice how the left-to-right evaluation of arguments is made explicit; also, notice how the *sequential* evaluation of the arguments is made explicit, with the second argument being read only after the first argument has been provided.

- $\lambda f{:}\mathbb{N} \to \mathbb{N}.f(1) : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$:

	$(\mathbb{N}$	$\to$	$\mathbb{N})$	$\to$	$\mathbb{N}$	*Interpretation*
O					q	Initial request for output
P			q			Request to input value of f
O	q					Request to output the argument of f?
P	1					The system outputs 1
O			$f(1)$			The environment provides value of $f(1)$ as input
P					$f(1)$	...which is the final output.

Traces become more complicated when functions are used as part of their own argument, because there is a nesting of the two function evaluations; this is illustrated by the term:

$$\lambda f{:}\mathbb{N} \to \mathbb{N}.f(f(1)) : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N},$$

with a typical play displayed in Figure 3.7 on the next page. Notice that both O and P ask for output, respectively input, twice in the first and second type component, but in different contexts. In order to keep track of the two interleaved traces we use pointers which indicate what move (action) enabled (caused, triggered) any other move.

The final key concept for which we give an intuitive introduction is that of composition of strategies. Given two systems representing strategies of types $\sigma : A \to B$ and $\tau : B \to C$ then a composite strategy $\sigma;\tau : A \to C$ is obtained by connecting the outputs of the B component of σ to the inputs of the B components of τ and *vice versa*, as visualized in Figure 3.8 on page 43. In this way, whenever B in τ asks for input, it is supplied by σ using its own B components. An important consequence of composition is that all the interactions originating in the B components are hidden. This is quite similar to the "parallel composition with hiding" of CSP [Hoa85].

In Figure 3.9 on page 44, we see how the strategies of $\lambda n.n{+}1$ and $\lambda f.f(1)$ compose in the game-semantic interpretation of function application. We examine the typical plays presented before, where the wavy lines represent the interactions realizing the composition. The dashed lines represent pointers that in the original strategies represented an enabling of actions in the plays before compositions. Notice that an *indirect* causation relation still exists between the same moves though, as we can see tracing back the dotted and the solid-line pointers denoting causations of interactions between and within

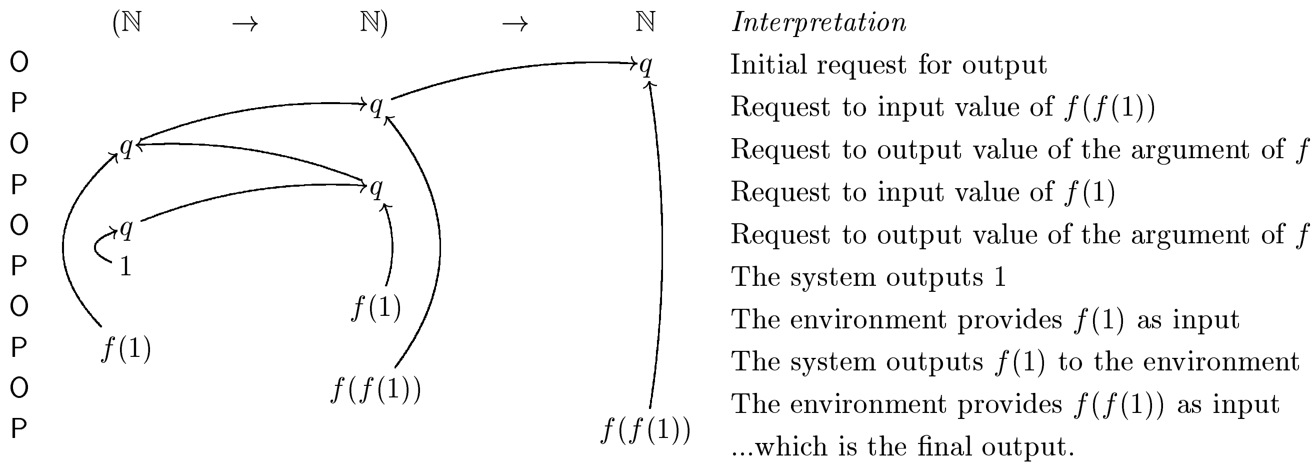


Figure 3.7: Typical play for $\lambda f{:}\mathbb{N} \to \mathbb{N}.f(f(1)) : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$

components. Finally, all the interactions in the area delimited by the dotted lines have become *internal* and are therefore unobservable. So the result is just the play $q \; 2$ in component $\mathbb{N}$.

3.3 The IA model

Having presented informally the ideas behind the game model of computation, we can introduce formally the technical definitions. Not all the definitions will be given and not all proofs will be reproduced in this section, but only those definitions necessary in order to show that the regular-language semantics presented in the following chapter is an accurate representation of the game-semantic model.

A *game* has two participants: P and O. A *play* of the game consists of a sequence of moves. Each move is *justified* by an earlier move in the play, unless it is an *initial* move, which needs no justification. It is O who always moves first.

We use meta-variable M to denote sets and s to range over sequences of moves; we use m to range over elements of sets (moves) and n to range over elements of sequences (move occurrences). The concatenation of two sequences is $\mathsf{s}_1\mathsf{s}_2$ and ϵ is the empty sequence. A singleton sequence is identified with its element and, if not ambiguous, a move occurrence may be identified notationally with the move. If M is a set, $\mathsf{s} \upharpoonright \mathsf{M}$ is the *restriction* of s to M, i.e. s with all elements not in M removed.

Definition 3.1 (Arena) *An arena is a structure* $\mathsf{Ar}_A = \langle \mathsf{M}_A, \lambda_A, \vdash_A \rangle$ *with:*

- M_A *a set of* moves*;*

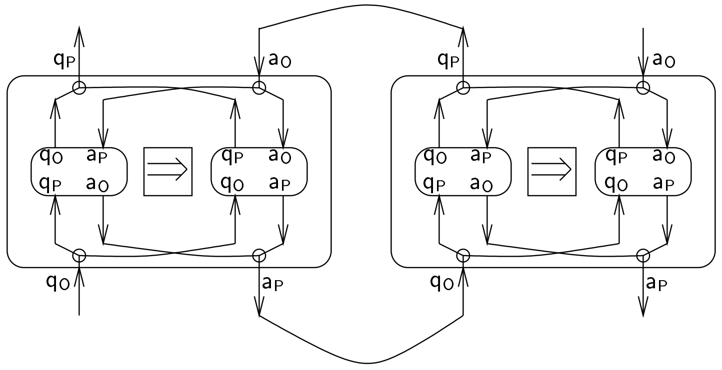


Figure 3.8: Composition of strategies

- $\lambda_A : \mathsf{M}_A \to \{P, O\} \times \{Q, A\}$ *is a* labeling *function. We write:*

$$\{P, O\} \times \{Q, A\} = \{OQ, OA, PQ, PA\} \qquad \lambda_A = \langle \lambda_A^{OP}, \lambda_A^{QA} \rangle,$$

  and we define $\overline{\lambda}_A$ *by reversing the* O, P, *labels:*

$$\overline{\lambda}_A(\mathsf{m}) = OQ \text{ if and only if } \lambda_A(\mathsf{m}) = PQ, \quad \overline{\lambda}_A(\mathsf{m}) = OA \text{ if and only if } \lambda_A(\mathsf{m}) = PA.$$

  If $\lambda^{OP}(\mathsf{m}) = O$ *we call* m *an* O*-move; otherwise we call it a* P*-move.*

- $\vdash_A \subseteq (\mathsf{M}_A + \{\star\}) \times \mathsf{M}_A$ *is an* enabling *relation with the properties:*
  - $\star \vdash \mathsf{m}$ *implies* $\lambda_A(\mathsf{m}) = OQ$ *and for all* m', $\mathsf{m}' \vdash \mathsf{m}$ *if and only if* $\mathsf{m}' = \star$;
  - $\mathsf{m} \vdash \mathsf{m}'$ *and* $\lambda_A^{QA}(\mathsf{m}') = A$ *then* $\lambda_A^{QA}(\mathsf{m}) = Q$;
  - $\mathsf{m} \vdash \mathsf{m}'$ *and* $\mathsf{m} \neq \star$ *implies* $\lambda^{OP}(\mathsf{m}) \neq \lambda^{OP}(\mathsf{m}')$.

An arena establishes the basic structure of a game: a game consists of moves with labels along with the enabling relation, defining what moves can follow what moves. A play then is a sequence of occurrences of moves together with the pointers connecting every non-initial move-occurrence with an earlier enabling move-occurrence; we call this a *justified* sequence, the pointers *justification* pointers, and the enabling move-occurrences *justifiers.* This is an important issue, because in general the way in which a sequence can be justified is not unique.

Definition 3.2 (Player view) *Given a justified sequence* s *the player view* $\ulcorner \mathsf{s} \urcorner$ *is a justified sequence defined by:*

$$\ulcorner \epsilon \urcorner = \epsilon$$

$$\ulcorner \mathsf{sm} \urcorner = \ulcorner \mathsf{s} \urcorner \mathsf{m} \qquad \text{if } \mathsf{m} \text{ is a } P\text{-move}$$

$$\ulcorner \mathsf{sm} \urcorner = \mathsf{m} \qquad \text{if } \star \vdash \mathsf{m}$$

$$\ulcorner \mathsf{s\ m\ s'\ m'} \urcorner = \ulcorner \mathsf{s} \urcorner\ \mathsf{m\ m'} \qquad \text{if } \mathsf{m}' \text{ is an } O\text{-move.}$$

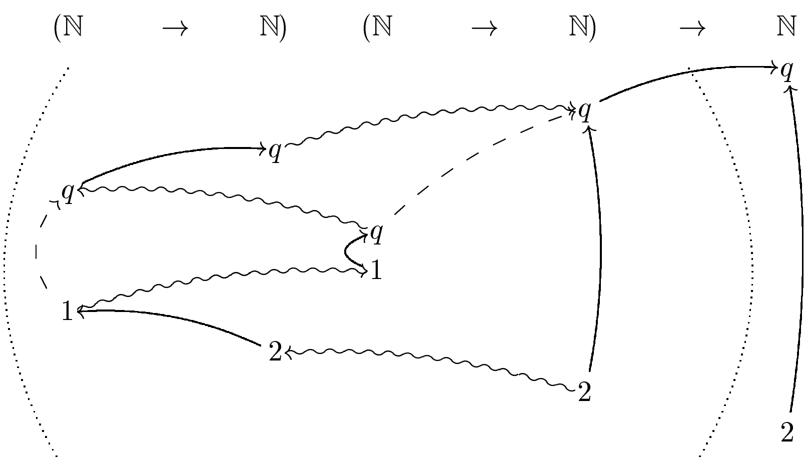


Figure 3.9: The application $\big(\lambda f.f(1)\big)(\lambda n.n+1)$, interpreted by composition of strategies.

An *Opponent view* $\llcorner \mathsf{s} \lrcorner$ is defined analogously. The concept of view is essential; several threads of dialogue may be interleaved and the view, for Player or Opponent, is the relevant context at any given moment.

We can now define the rules of the game:

Definition 3.3 (Legal plays) *A justified sequence is legal if and only if it satisfies the following principles:*

Alternation: *if* $\mathsf{s} = \mathsf{s m m' s'}$ *then* $\lambda^{OP}(\mathsf{m}) \neq \lambda^{OP}(\mathsf{m'})$*;*

Bracketing: *pointers justifying answers never intersect, i.e. an answer always is to the last pending question, so there are no legal plays of the form:* $\cdots\ \mathsf{q} \cdots\ \mathsf{q'} \cdots\ \mathsf{a}\ \cdots\ \mathsf{a'}$;

Visibility: *if* $\mathsf{s'm}$ *is a prefix of* s *and* m *is a P-move (O-move), then its justifier is in* $\ulcorner \mathsf{s'} \urcorner$ *(*$\llcorner \mathsf{s'} \lrcorner$*).*

The set of all legal positions of an arena Ar is L_A.

Definition 3.4 *A move-occurrence* n *in* s *is* hereditarily justified *by move occurrence* $\mathsf{n'}$ *if there is a chain of justification pointers from* n *to* $\mathsf{n'}$ *in* s*:*

$$\mathsf{s} = \ \cdots\ \mathsf{n'} \cdots\ \cdots\ \cdots\ \mathsf{n}\ \cdots$$

We write $\mathsf{s} \upharpoonright \mathsf{n}$ *for the subsequence of* s *consisting of move occurrences hereditarily justified by move-occurrence* n*. We write* $\mathsf{s} \upharpoonright \mathsf{I}$ *for the above generalized to a set of initial moves* I*.*

Definition 3.5 (Game) *A game is a structure* $\partial_A = \langle \mathsf{M}_A, \lambda_A, \vdash_A, \mathsf{P}_A \rangle$ *where*

- $\langle \mathsf{M}_A, \lambda_A, \vdash_A \rangle$ *is an arena;*
- P_A *is a non-empty prefix-closed set of legal positions, called the set of* valid positions *of* ∂_A*, such that if* $\mathsf{s} \in \mathsf{P}_A$ *then* $\mathsf{s} \upharpoonright \mathsf{I} \in \mathsf{P}_A$ *for any set* I *of initial moves of* s.

We denote that s is a prefix of s' by $s \sqsubseteq s'$.

Every term of the language is interpreted as a *strategy*, i.e. the predetermined way P moves at any given position:

Definition 3.6 (Strategy) *A strategy* Σ *is a set of even-length positions such that:*

$$\begin{array}{ll} \epsilon \in \Sigma & \textit{empty} \\ \textit{if } \ \mathsf{snn}' \in \Sigma \ \textit{ then } \ \mathsf{s} \in \Sigma & \textit{prefix-closed} \\ \textit{if } \ \mathsf{snn}', \mathsf{snn}'' \in \Sigma \ \textit{ then } \ \mathsf{n}' = \mathsf{n}''. & \textit{determinism} \end{array}$$

A technically important role is played by the so called *innocent* strategies, i.e. strategies that depend only on the view, not on the play as a whole:

Definition 3.7 (Innocence) *A strategy* Σ *is innocent if and only if* $\mathsf{snn}' \in \Sigma$*,* $\mathsf{s}' \in \Sigma$*,* $\mathsf{s}'\mathsf{n} \in \mathsf{P}_A$ *and* $\ulcorner \mathsf{sn} \urcorner = \ulcorner \mathsf{s}'\mathsf{n} \urcorner$ *imply* $\mathsf{s}'\mathsf{nn}' \in \Sigma$.

That is, the next move n' is determined only by the current P-view, $\ulcorner \mathsf{sn} \urcorner$. So strategy Σ can be viewed as a partial function from P-views to P-moves; we call it the *view-function* of Σ. The smallest possible strategy is the (unresponsive) empty strategy $\perp = \{\epsilon\}$. We write $\Sigma : A$ to indicate that Σ is an innocent strategy for game ∂_A. One could go back to the examples of the previous section and see that, indeed, the plays considered there conform to the restrictions enumerated thus far: alternation, bracketing, visibility, innocence.

We can now look at how games and strategies can be composed.

Definition 3.8 (Connectives)

- *Given games* $\partial_A = \langle \mathsf{M}_A, \lambda_A, \vdash_A, \mathsf{P}_A \rangle$ *and* $\partial_B = \langle \mathsf{M}_B, \lambda_B, \vdash_B, \mathsf{P}_B \rangle$ *the game* $\partial_{A \multimap B}$ *is defined by:*

$$\begin{aligned} \mathsf{M}_{A \multimap B} &= \mathsf{M}_A + \mathsf{M}_B \qquad \textit{(disjoint union)} \\ \lambda_{A \multimap B} &= [\overline{\lambda}_A, \lambda_B] \\ \star \vdash_{A \multimap B} \mathsf{m} &\iff \star \vdash_B \mathsf{m} \\ \mathsf{m} \vdash_{A \multimap B} \mathsf{m}' &\iff \mathsf{m} \vdash_A \mathsf{m}' \textit{ or } \mathsf{m} \vdash_B \mathsf{m}' \textit{ or } \star \vdash_B \mathsf{m} \textit{ or } \star \vdash_A \mathsf{m}' \\ \mathsf{P}_{A \multimap B} &= \{\mathsf{s} \in \mathsf{L}_{A \multimap B} \mid \mathsf{s} \upharpoonright \mathsf{M}_A \in \mathsf{P}_A, \ \ \mathsf{s} \upharpoonright \mathsf{M}_B \in \mathsf{P}_A\}. \end{aligned}$$

- *Given games* $\partial_A = \langle \mathsf{M}_A, \lambda_A, \vdash_A, \mathsf{P}_A \rangle$ *and* $\partial_B = \langle \mathsf{M}_B, \lambda_B, \vdash_B, \mathsf{P}_B \rangle$ *the game* $\partial_{A \,\&\, B}$ *is defined by:*

$$\mathsf{M}_{A \,\&\, B} = \mathsf{M}_A + \mathsf{M}_B \qquad \textit{(disjoint union)}$$

$$
\begin{aligned}
\lambda_{A\,\&\,B} &= [\lambda_A, \lambda_B] \\
\star \vdash_{A\,\&\,B} \mathsf{m} &\iff \star \vdash_A \mathsf{m} \text{ or } \star \vdash_B \mathsf{m} \\
\mathsf{m} \vdash_{A\,\&\,B} \mathsf{m}' &\iff \mathsf{m} \vdash_A \mathsf{m}' \text{ or } \mathsf{m} \vdash_B \mathsf{m}' \\
\mathsf{P}_{A\,\&\,B} &= \{\mathsf{s} \in \mathsf{L}_{A\,\&\,B} \mid \mathsf{s} \upharpoonright \mathsf{M}_A \in \mathsf{P}_A,\ \ \mathsf{s} \upharpoonright \mathsf{M}_B = \epsilon\} \\
&\quad \cup \{\mathsf{s} \in \mathsf{L}_{A\,\&\,B} \mid \mathsf{s} \upharpoonright \mathsf{M}_A = \epsilon,\ \ \mathsf{s} \upharpoonright \mathsf{M}_B \in \mathsf{P}_B\}.
\end{aligned}
$$

Definition 3.9 (Composition) *Given strategies $\Sigma : A \multimap B$ and $\Sigma' : B \multimap C$ we define $\Sigma; \Sigma'$ as:*

$$\{\mathsf{s} \upharpoonright \mathsf{M}_A + \mathsf{M}_C \bigm| \mathsf{s} \upharpoonright \mathsf{M}_A + \mathsf{M}_B \in \Sigma,\ \mathsf{s} \upharpoonright \mathsf{M}_B + \mathsf{M}_C \in \Sigma'\}$$

where s ranges over justified sequences of moves in $\mathsf{M}_A + \mathsf{M}_B + \mathsf{M}_C$.

It can be shown that the composition of strategies, as defined above is a strategy for $\eth_{A \multimap C}$. One could go back to the example described in Figure 3.9 on page 44 and see that indeed, when composing the two strategies the restrictions expressed above were respected.

Definition 3.10 (Copy-cat) *The identity strategy $\mathsf{id} : A \multimap A$ is given by the copy-cat strategy:*

$$\{\mathsf{s} \in \mathsf{P}_{A_1 \multimap A_2} \mid \textit{if } \mathsf{s}' \sqsubseteq \mathsf{s}, \textit{even}(\textit{length}(\mathsf{s}')),\ \textit{then } \mathsf{s}' \upharpoonright A_1 = \mathsf{s}' \upharpoonright A_2\},$$

where we use A_1, A_2 to denote the two occurrences of A in the type.

Corresponding to the product $A \,\&\, B$, we can define pairing.

Definition 3.11 (Pairing) *Given strategies $\Sigma_i : A \multimap B_i$, for $i = 0, 1$ we define their pairing as the strategy $\langle \Sigma_0, \Sigma_1 \rangle : A \multimap B_0 \,\&\, B_1$ such that:*

$$\langle \Sigma_0, \Sigma_1 \rangle = \bigcup_{i=0,1} \{\mathsf{s} \in \mathsf{L}_{A \multimap B_0 \,\&\, B_1} \mid \mathsf{s} \upharpoonright A + B_i \in \Sigma_i \ \ \textit{and}\ \ \mathsf{s} \upharpoonright A + B_{1-i} = \epsilon\}.$$

Projections $\mathsf{p}_i : B_0 \,\&\, B_1 \multimap B_i$ are defined by the obvious copy-cat strategies.

The use of linear connectives so far is not accidental. Strategies for $A \multimap B$ are similar to linear implication because they can only use the component A at most once. In order to model arbitrary functions we need something akin to classical implication, where component A can be reused. The standard representation of classical implication using the linear one is:

$$A \to B = !A \multimap B,$$

where '!' is a "reuse" operator called the *exponential*. We model it using games as follows:

Definition 3.12 (Exponential) *Given game* $\partial_A = \langle \mathsf{M}_A, \lambda_A, \vdash_A, \mathsf{P}_A \rangle$*, the game of* $\partial_{!A}$ *is defined as:*

$$\begin{aligned} \mathsf{M}_{!A} &= \mathsf{M}_A \\ \lambda_{!A} &= \lambda_A \\ \mathsf{P}_{!A} &= \{\mathsf{s} \in \mathsf{L}_{!A} \mid \star \vdash_A \mathsf{m} \textit{ implies } \mathsf{s} \upharpoonright \mathsf{m} \in \mathsf{P}_A\}. \end{aligned}$$

That is, a play in $!A$ consists of interleaved plays of A.

A certain class of games, in which O asks only one opening question, plays an important technical role:

Definition 3.13 (Well-opened games) *A game* ∂_A *is well opened if and only if, for all* $\mathsf{sm} \in \mathsf{P}_A$*, if* m *is initial then* $\mathsf{s} = \epsilon$*.*

So a well-opened game is one in which there is only one "main" thread of play tracing back to the (unique) opening question of O. If B is well-opened it can be shown that $!A \multimap B$, i.e. $A \to B$, is well-opened as well.

Theorem 3.1 (CCC) *If we take well-opened games to be objects and innocent strategies of the form* $\Sigma : !A \multimap B$ *to be morphisms* $A \to B$*, the resulting structure is a Cartesian closed category.*

It is a standard property of this algebraic structure[1] that it can model (higher-order) functions. Before giving the proof for the theorem, the following technical definitions are required:

Definition 3.14 (Promotion and dereliction)

- *Given* $\Sigma : !A \multimap B$ *we define the* promotion *strategy* $\Sigma^\dagger : !A \multimap !B$ *as*

$$\Sigma^\dagger = \{\mathsf{s} \in \mathsf{L}_{!A \multimap !B} \mid \mathsf{m} \textit{ initial implies } s \upharpoonright m \in \Sigma\}.$$

- *We define the* dereliction *strategy* $\mathsf{der}_A : !A \multimap A$ *as the copy-cat strategy on* A*.*

These two strategies can be thought of as "adaptors" between the "single-threaded" strategies used by the linear connectives and the "multi-threaded" strategies used by the classic connectives (promotion) and *vice versa* (dereliction).

PROOF: *(Of Theorem 3.1)*
On the left-hand side we have the relevant category-theoretic constructs and on the right-hand side the game-theoretic implementations.

Identity. $A \xrightarrow{id_A} A \stackrel{\text{def}}{=} !A \stackrel{\mathsf{der}_A}{\multimap} A.$

Composition. $A \xrightarrow{\Sigma_1} B \xrightarrow{\Sigma_2} C \stackrel{\text{def}}{=} !A \stackrel{\Sigma_1^\dagger}{\multimap} !B \stackrel{\Sigma_2}{\multimap} C.$

[1] Standard introductory texts such as [BW90] describe this structure in detail.

Exponential. $A \Rightarrow B \stackrel{\text{def}}{=} !A \multimap B$.

It can be quite easily checked that there exists an isomorphism $\Lambda(-)$ between the following strategies:

$$\Lambda_B : \Sigma_{!(A \,\&\, B) \multimap C} \cong \Sigma_{!A \multimap (!B \multimap C)},$$

as strategies for the game on the left differ from strategies for the game on the right only because of the tagging used in disjoint unions.

This isomorphism gives the CCC evaluation map, modeling application:

$$ev_{A,B} = \Lambda_B^{-1}(\mathsf{id}_{!A \multimap B}),$$

which is simply a copy-cat strategy between the game $!A \multimap B$ and another copy of itself.

Product. $A_1 \times A_2 = A_1 \,\&\, A_2$, with CCC projections $\pi_i : A_1 \times A_2 \to A_i$ defined as

$$A_1 \times A_2 \xrightarrow{\pi_i} A_i = !(A_1 \,\&\, A_2) \stackrel{\mathsf{der}}{\multimap} (A_1 \,\&\, A_2) \stackrel{\mathsf{p}_i}{\multimap} A_i.$$

Pairing is the same as defined earlier (Definition 3.11 on page 46).

This is an extremely abbreviated proof. The details of the constructions followed by this presentation are given in [McC98, Chapter 3].

Any CCC comes equipped with a *terminal* object $\mathbb{I}$, which, in this particular category, is the game $\partial_{\mathsf{I}_\emptyset} = \langle \emptyset, \emptyset, \emptyset, \{\epsilon\} \rangle$; in addition, this particular CCC also has a family of *recursion morphisms* Y_A for any object A, such that $\mathsf{Y}_A : (A \Rightarrow A) \to A$, with the property that it is a fixed-point operator [Pla66]:

$$\mathsf{Y}_A = \langle id_{A \Rightarrow A}, id_{A \Rightarrow A} \rangle ; \langle id_{A \Rightarrow A}, \mathsf{Y}_A \rangle ; ev_{A \Rightarrow A, A}.$$

END OF PROOF.

Before giving the game-semantic model of IA, following the presentation in [AM96], it helps to slightly reformulate the syntax of the language so that every term-forming IA combinator is introduced as a functional constant. This new syntax, which is essentially the same as the usual one, is given in Figure 3.10 on the next page. Note that full IA also has a recursion combinator $\mathtt{rec}_\theta : (\theta \to \theta) \to \theta$ which is missing from first-order IA.

For example, the program $x :=$ **if** b **then** $!x + 1$ **else** 0 will be translated into:

$$\mathtt{asg}(x)(\mathtt{cnd}(b)(\mathtt{opr}_{+}(\mathtt{drf}(x))(1))(0)).$$

Iteration is translated using the recursion combinator:

$$\textbf{while } B \textbf{ do } C = \mathtt{rec}_{\mathbf{comm}}(\lambda c{:}\mathbf{comm}.\textbf{if } B \textbf{ then } C; c \textbf{ else skip}), \tag{3.1}$$

with c not free in B, C. Also, divergence is expressible using recursion:

$$\textbf{diverge} = \mathtt{rec}_{\mathbf{comm}}(\lambda c{:}\mathbf{comm}.\textbf{if true then } c \textbf{ else skip}).$$

Functional constant	Term-forming combinator
$\mathtt{cnd} : \mathbf{expbool} \to \sigma \to \sigma \to \sigma$	$\mathbf{if}\ B\ \mathbf{then}\ M\ \mathbf{else}\ M'$
$\mathtt{opr}_\star : \mathbf{exp}\tau \to \mathbf{exp}\tau \to \mathbf{exp}\tau'$	$E \star E'$
$\mathtt{seq} : \mathbf{comm} \to \sigma \to \sigma$	$C; M$
$\mathtt{drf} : \mathbf{var}\tau \to \mathbf{exp}\tau$	$!V$
$\mathtt{asg} : \mathbf{var}\tau \to \mathbf{exp}\tau \to \mathbf{comm}$	$V := E$
$\mathtt{new}_\tau : (\mathbf{var}\tau \to \sigma) \to \sigma$	$\mathbf{new}\tau\ x\ \mathbf{in}\ M$

Figure 3.10: An alternative syntax for IA

The semantic valuations of terms have the form:

$$[\![\Gamma \vdash P : \theta]\!] = \Sigma_P : [\![\Gamma]\!] \to [\![\theta]\!] .$$

where types are interpreted as games (objects of the CCC defined by Theorem 3.1) and terms as strategies (morphisms of the same CCC).

Definition 3.15 (Interpretation of types) *IA types are interpreted as objects in the category of games:*

- $[\![\mathbf{expbool}]\!] = \partial_{\mathbf{expbool}} = \langle \mathsf{M}_{\mathbf{expbool}}, \lambda_{\mathbf{expbool}}, \vdash_{\mathbf{expbool}}, \mathsf{P}_{\mathbf{expbool}} \rangle$ *where:*
  - $\mathsf{M}_{\mathbf{expbool}} = \{q\} \cup \{tt, f\!f\}$
  - $\lambda_{\mathbf{expbool}}(q) = \mathsf{OQ},\ \lambda_{\mathbf{expbool}}(tt) = \mathsf{PA},\ \lambda^{\mathsf{OP}}_{\mathbf{expbool}}(f\!f) = \mathsf{A}$
  - $\vdash_{\mathbf{expbool}} = \{(\star, q), (q, tt), (q, f\!f)\}$
- $[\![\mathbf{expint}]\!] = \partial_{\mathbf{expint}} = \langle \mathsf{M}_{\mathbf{expint}}, \lambda_{\mathbf{expint}}, \vdash_{\mathbf{expint}}, \mathsf{P}_{\mathbf{expint}} \rangle$ *where:*
  - $\mathsf{M}_{\mathbf{expint}} = \{q\} \cup \{n \mid n \in \mathbb{Z}\}$
  - $\lambda_{\mathbf{expbool}}(q) = \mathsf{OQ},\ \lambda_{\mathbf{expbool}}(n) = \mathsf{PA},\ n \in \mathbb{Z}$
  - $\vdash_{\mathbf{expint}} = \{(\star, q)\} \cup \{(q, n) \mid n \in \mathbb{Z}\}$
- $[\![\mathbf{comm}]\!] = \partial_{\mathbf{comm}} = \langle \mathsf{M}_{\mathbf{comm}}, \lambda_{\mathbf{comm}}, \vdash_{\mathbf{comm}}, \mathsf{P}_{\mathbf{comm}} \rangle$ *where:*
  - $\mathsf{M}_{\mathbf{comm}} = \{run, done\}$
  - $\lambda_{\mathbf{comm}}(run) = \mathsf{OQ},\ \lambda_{\mathbf{expbool}}(done) = \mathsf{PA}$
  - $\vdash_{\mathbf{comm}} = \{(\star, run), (run, done)\}$
- $[\![\mathbf{var}\tau]\!] = A_{acc\tau} \times [\![\mathbf{exp}\tau]\!]$, *where*

$$A_{acc\mathbf{bool}} = [\![\mathbf{comm}]\!] \times [\![\mathbf{comm}]\!] , \quad A_{acc\mathbf{int}} = \prod_{1 \le i \le |\mathcal{Z}|} [\![\mathbf{comm}]\!] .$$

where $|\mathcal{Z}|$ is the size of the data set of integers, finite or countably infinite.

- $[\![\sigma \to \theta]\!] = [\![\sigma]\!] \Rightarrow [\![\theta]\!]$.

In the game of $\mathbf{var}\tau$ we will change the names of the moves in the following way: instead of q, we will denote the question in the game for the $\mathbf{exp}\tau$ component by *read*; instead of *run*, *done* tagged with n in the game for the n^{th} **comm** component we will use *write*(n), *ok*.

The interpretation of the variable type $\mathbf{var}\tau$ as a product of an acceptor type and an expression type was first given by Reynolds [Rey81a]. Sometimes, the acceptor type is further refined as:

$$A_{acc\tau} \stackrel{\text{def}}{=} \mathbf{exp}\tau \Rightarrow \mathbf{comm},$$

but Abramsky and McCusker [AM96, Section 2] point out that this further step is not consistent with the call-by-name function mechanism of IA. In the function type $\mathbf{exp}\tau \Rightarrow$ **comm** the expression is not evaluated, so an acceptor of that type would store it as a "thunk." But this is not how assignment works; assignment is not "by name" but "by value," with the expression on the right-hand-side being evaluated during the assignment operation. Refining the acceptor type as in the definition above avoids this problem. A more precise categorical justification of the proper construction is also given.

The environment Γ is interpreted as:

$$[\![\Gamma]\!] = \prod_{x:\theta \in \Gamma} [\![\theta]\!].$$

Constants do not need an environment, as they have no free identifiers, so we will omit the empty environment from the definitions. Constants can be interpreted by the following strategies, morphisms in the CCC of Theorem 3.1. Because strategies are, by definition, prefix-closed it suffices to give their sets of complete plays. Also, for these strategies the justification pointers can be unambiguously reconstructed for any given complete play.

Definition 3.16 (Interpretation of constants) *IA constants are interpreted as morphisms in the category of games (i.e. strategies):*

- $[\![n : \mathbf{expint}]\!] = \Sigma_n : \mathbb{I} \to [\![\mathbf{expint}]\!]$, $\Sigma_n = \{q \cdot n\}$.
- $[\![\mathbf{true} : \mathbf{expbool}]\!] = \Sigma_{\mathbf{true}} : \mathbb{I} \to [\![\mathbf{expbool}]\!]$, $\Sigma_{\mathbf{true}} = \{q \cdot tt\}$.
- $[\![\mathbf{false} : \mathbf{expbool}]\!] = \Sigma_{\mathbf{true}} : \mathbb{I} \to [\![\mathbf{expbool}]\!]$, $\Sigma_{\mathbf{false}} = \{q \cdot f\!f\}$.
- $[\![\mathbf{skip} : \mathbf{comm}]\!] = \Sigma_{\mathbf{skip}} : \mathbb{I} \to [\![\mathbf{comm}]\!]$, $\Sigma_{\mathbf{comm}} = \{run \cdot done\}$.
- $[\![\mathsf{cnd} : \mathbf{expbool} \to \sigma \to \sigma \to \sigma]\!] = \Sigma_{\mathsf{cnd}} : \mathbb{I} \to ([\![\mathbf{expbool}]\!] \Rightarrow [\![\sigma]\!] \Rightarrow [\![\sigma]\!] \Rightarrow [\![\sigma]\!])$, *where* Σ_{cnd} *is:*

$$\begin{array}{ccccccc}
[\![\mathbf{expbool}]\!] & \Rightarrow & [\![\sigma]\!] & \Rightarrow & [\![\sigma]\!] & \Rightarrow & [\![\sigma]\!] \\
 & & & & & & \mathsf{q} \\
q & & & & & & \\
tt & & & & & & \\
 & & \mathsf{q} & & & & \\
 & & \mathsf{a} & & & & \\
 & & & & & & \mathsf{a}
\end{array}
\qquad
\begin{array}{ccccccc}
[\![\mathbf{expbool}]\!] & \Rightarrow & [\![\sigma]\!] & \Rightarrow & [\![\sigma]\!] & \Rightarrow & [\![\sigma]\!] \\
 & & & & & & \mathsf{q} \\
q & & & & & & \\
f\!f & & & & & & \\
 & & & & \mathsf{q} & & \\
 & & & & \mathsf{a} & & \\
 & & & & & & \mathsf{a}
\end{array}$$

- $[\![\mathtt{opr}_\star : \mathbf{exp}\tau \to \mathbf{exp}\tau \to \mathbf{exp}\tau']\!] = \Sigma_{\mathtt{opr}_\star} : \mathbb{I} \to ([\![\mathbf{exp}\tau]\!] \Rightarrow [\![\mathbf{exp}\tau]\!] \Rightarrow [\![\mathbf{exp}\tau']\!])$, *where* $\Sigma_{\mathtt{opr}_\star}$ *is:*

$$\begin{array}{ccccc}
[\![\mathbf{exp}\tau]\!] & \Rightarrow & [\![\mathbf{exp}\tau]\!] & \Rightarrow & [\![\mathbf{exp}\tau']\!] \\
 & & & & q \\
q & & & & \\
\mathsf{a}' & & & & \\
 & & q & & \\
 & & \mathsf{a}'' & & \\
 & & & & \mathsf{a}
\end{array}$$

where $\mathsf{a} = \mathsf{a}' \star \mathsf{a}''$, *with* $\star$ *the obvious interpretation of arithmetic or logic operator* $\mathtt{opr}_\star$.

- $[\![\mathtt{seq} : \mathbf{comm} \to \sigma \to \sigma]\!] = \Sigma_{\mathtt{seq}} : \mathbb{I} \to ([\![\mathbf{comm}]\!] \Rightarrow [\![\sigma]\!] \Rightarrow [\![\sigma]\!])$, *where* $\Sigma_{\mathtt{seq}}$ *is:*

$$\begin{array}{ccccc}
[\![\mathbf{comm}]\!] & \Rightarrow & [\![\sigma]\!] & \Rightarrow & [\![\sigma]\!] \\
 & & & & \mathsf{q} \\
run & & & & \\
done & & & & \\
 & & \mathsf{q} & & \\
 & & \mathsf{a} & & \\
 & & & & \mathsf{a}
\end{array}$$

- $[\![\mathtt{drf} : \mathbf{var}\tau \to \mathbf{exp}\tau]\!] = \Sigma_{\mathtt{drf}} : \mathbb{I} \to ([\![\mathbf{var}\tau]\!] \Rightarrow [\![\mathbf{exp}\tau]\!])$, *where* $\Sigma_{\mathtt{drf}}$ *is:*

$$\begin{array}{ccc}
[\![\mathbf{var}\tau]\!] & \Rightarrow & [\![\mathbf{exp}\tau]\!] \\
 & & q \\
read & & \\
\mathsf{a} & & \\
 & & \mathsf{a}
\end{array}$$

- $[\![\mathtt{asg} : \mathbf{var}\tau \to \mathbf{exp}\tau \to \mathbf{comm}]\!] = \Sigma_{\mathtt{asg}} : \mathbb{I} \to ([\![\mathbf{var}\tau]\!] \Rightarrow [\![\mathbf{exp}\tau]\!] \Rightarrow [\![\mathbf{comm}]\!])$, *where* $\Sigma_{\mathtt{asg}}$ *is:*

$$\begin{array}{ccccc}
[\![\mathbf{var}\tau]\!] & \Rightarrow & [\![\mathbf{exp}\tau]\!] & \Rightarrow & [\![\mathbf{comm}]\!] \\
 & & & & run \\
 & & q & & \\
 & & \mathsf{a} & & \\
write(\mathsf{a}) & & & & \\
ok & & & & \\
 & & & & done
\end{array}$$

- $[\![\mathtt{rec}_\theta : (\theta \to \theta) \to \theta]\!] = \Sigma_{\mathtt{rec}_\theta} : \mathbb{I} \to ([\![\theta]\!] \Rightarrow [\![\theta]\!]) \Rightarrow [\![\theta]\!]$, *with* $\Sigma_{\mathtt{rec}_\theta} = \Lambda_\theta(\mathsf{Y}_\theta)$.

In the above, we have used a tabular representation of strategies as sets of complete plays, every move being aligned with the type component in which it occurs. For example, the strategies for the conditional, written as sets of justified sequences, are:

$$\Sigma_{\text{cmd}} = \left\{ \mathsf{q}^4 \cdot q^1 \cdot tt^1 \cdot \mathsf{q}^2 \cdot \mathsf{a}^2 \cdot \mathsf{a}^4 \ , \ \mathsf{q}^4 \cdot q^1 \cdot tt^1 \cdot \mathsf{q}^3 \cdot \mathsf{a}^3 \cdot \mathsf{a}^4 \right\}.$$

The tag 1 corresponds to type **expbool**, tag 2 two the first σ, tag 3 to the second σ and tag 3 to the result type σ.

In the case of **diverge**, the plays are infinite, so no set of *complete* plays exists; the strategy can be therefore represented as the empty set of complete plays.

Identifiers are interpreted by the copy-cat strategy:

Definition 3.17 (Identifiers) $[\![\Gamma \vdash x : \theta]\!] = \pi_x : [\![\Gamma]\!] \to [\![\theta]\!]$.

Terms are formed from constants, abstraction and application:

Definition 3.18 (Abstraction and application)

Abstraction. *If*

$$[\![\Gamma, x : \theta' \vdash P : \theta]\!] = p : [\![\Gamma]\!] \times [\![\theta']\!] \to [\![\theta]\!]$$

then

$$[\![\Gamma \vdash \lambda x{:}\theta'.P : \theta' \to \theta]\!] = \Lambda_{\theta'}(p) : [\![\Gamma]\!] \to \left([\![\theta']\!] \Rightarrow [\![\theta]\!]\right).$$

Application. *If*

$$[\![\Gamma \vdash F : \theta' \to \theta]\!] = f : [\![\Gamma]\!] \to \left([\![\theta']\!] \Rightarrow [\![\theta]\!]\right)$$
$$[\![\Gamma \vdash M : \theta']\!] = m : [\![\Gamma]\!] \to [\![\theta']\!]$$

then

$$[\![\Gamma \vdash FM]\!] = \langle f, m\rangle ; ev_{\theta',\theta} : [\![\Gamma]\!] \to [\![\theta]\!].$$

All definitions are exactly as in [McC98, AM96], where it is shown that indeed they have the required categorical properties.

The only structure not interpreted yet is local-variable definition `new`. The reason is that the strategy interpreting this combinator is substantially different from all the strategies used so far: it is not an innocent strategy (Definition 3.7 on page 45). We remember that an innocent strategy is a strategy in which P must choose its next move based only on its current view, i.e. on the current context of the game. But, in order to interpret `new`, P must be able to see the entire course of the game. Intuitively, the reason is simple: a local variable has *state*, and changes inflicted on the state have global consequences. If O modifies the state of the variable by writing to it, say by playing *write*(6) in some thread that is outside of the current view of P, then P must be aware of that move so that it can correctly answer 6 if a *read* move is subsequently played.

According to a convincing argument initially put forth by Abramsky and McCusker, the need for *knowing* (i.e. not innocent) strategies in order to interpret state represents the key distinction, the boundary, between functional and imperative programming. All features of IA, except `new`, can be modeled by innocent strategies, which means they can be encoded in PCF.

The following knowing strategy is stateful.

Definition 3.19 (Cell) *Let* $\Sigma_{\mathsf{cell}\tau} : \mathsf{I}_\emptyset \multimap\, ! [\![\mathbf{var}\tau]\!]$ *so that plays in* $! [\![\mathbf{var}\tau]\!]$ *have the form*

$$\mathit{read} \cdot \mathsf{a}_0 \cdot \mathit{write}(\mathsf{a}_1) \cdot \mathit{ok} \cdot \mathit{read} \cdot \mathsf{a}_1 \cdots$$

where each read and write(a) *are initial.*

Notice that in general plays in $! [\![\mathbf{var}\tau]\!]$ need not observe any causal relation between *write* and *read*, so that plays such as

$$\mathit{read} \cdot \mathsf{7} \cdot \mathit{write}(\mathsf{4}) \cdot \mathit{ok} \cdot \mathit{read} \cdot \mathsf{9}$$

are legal. The strategy cell introduces the global causality condition that the last value written is the value read, which is a condition that characterizes store.

We can now define `new`.

Definition 3.20 (Block variable) *If*

$$[\![\Gamma, x : \mathbf{var}\tau \vdash C : \sigma]\!] = c : [\![\Gamma]\!] \times [\![\mathbf{var}\tau]\!] \to [\![\sigma]\!]$$

then

$$[\![\Gamma \vdash \mathtt{new}_\tau(\lambda x{:}\mathbf{var}\tau.C : \sigma)]\!] = \langle \mathsf{id}_\Gamma, \Sigma_{\mathsf{cell}\tau}\rangle ; c : [\![\Gamma]\!] \to [\![\sigma]\!]\,,$$

where the composition $-;-$ *above is composition of strategies (not of morphisms).*

It is interesting to point out that a "low level" definition of `new` in terms of the underlying games and strategies, rather than a "high level" definition in terms of CCC morphisms is really required. Block variables cannot be modeled as:

$$\mathtt{new}_\tau f \stackrel{\text{def}}{=} f(\mathsf{cell}\tau)$$

because the categorical representation would require the strategy interpreting $\mathsf{cell}\tau$ to be promoted to $\Sigma^\dagger_{\mathsf{cell}\tau}$. This means that an arbitrary number of copies of (well behaved) variables are created! But, clearly, exactly one variable is required in the interpretation of `new`. This insight resonates with Reddy's observation[2] that an object-creating operator **new** C in a call-by-name object-oriented language must be of type $(\theta \to \mathbf{comm}) \to \mathbf{comm}$ rather than θ, the type of C [Red98, p. 11]:

[2]Reddy credits this observation to Reynolds [Rey81a].

> The type of **new** C illustrates how the *physical* nature of objects is reconciled with the *mathematical* character of Algol. If **new** C were to be regarded as a value of type θ then the mathematical nature of Algol would prohibit stateful objects entirely.

Abramsky and McCusker show that the above interpretation of IA is fully abstract:

Theorem 3.2 (Full Abstraction) *Let* $\Gamma \vdash M_i : \theta$, $i = 1, 2$; *then*

$$\Gamma \vdash M_1 \equiv_\theta M_2 \quad \textit{if and only if} \quad [\![\Gamma \vdash M_1 : \theta]\!]^{\mathsf{comp}} = [\![\Gamma \vdash M_2 : \theta]\!]^{\mathsf{comp}},$$

where Σ^{comp} *is the set of complete plays of a strategy* Σ.

In words, IA phrases are observationally equivalent if and only if their sets of complete plays in the game model are equal. We should point out that this characterization of meaning as sets of complete plays is specific to IA and not in general the case for game models of arbitrary programming languages. In the absence of side-effects (PCF, IA with passive expressions) strategies need to be quotiented by composition, and in the presence of other computational features (control, references) sets of partial plays are needed. From this point of view we can say that IA is ideally suitable to a game-theoretic analysis.

Chapter 4

Regular-language Semantics

Anybody can play weird, that's easy. What's hard is to be as simple as Bach.

Charles Mingus

If we restrict IA to its recursion-free finitary first-order fragment then much of the games apparatus described in the previous chapter becomes unnecessary. The justification pointers for all sets of complete plays of strategies are uniquely determined by the plays themselves, so need not be explicitly represented. Moreover, these sets are regular and can be described by a meta-language of extended regular expressions.

4.1 Semantic definitions

4.1.1 Lexical, alphabet and language operations

Several *lexical* operations are first needed. They involve tagging a symbol or changing the tagging of a symbol, resulting in a new symbol:

Definition 4.1 (Lexical operations)

Tag. *Given two symbols* $\alpha_0, \alpha \in \mathcal{A}$, $\alpha_0^{\langle\alpha\rangle}$ *is a new symbol obtained by* tagging *the former with the latter.*

We define the alphabet $\mathcal{A}^{\langle\alpha\rangle} = \{\alpha_0^{\langle\alpha\rangle} \mid \alpha_0 \in \mathcal{A}\}$. *Conversely, we define the alphabet of all strings* not *tagged by a symbol* $\mathcal{A}^{\overline{\alpha}} = \{\alpha' \in \mathcal{A} \mid$ *there is no* α_0 *such that* $\alpha' = \alpha_0^{\langle\alpha\rangle}\}$.

Increment. *The lexical operation* $-\!\uparrow$ *is defined as follows:*

$$\alpha\!\uparrow = \begin{cases} \alpha_0^{\langle n+1\rangle} & \textit{if } \alpha = \alpha_0^{\langle n\rangle},\ n \in \mathbb{N} \\ \alpha & \textit{otherwise.} \end{cases}$$

We define the alphabet $\mathcal{A}\!\uparrow = \{\alpha\!\uparrow \mid \alpha \in \mathcal{A}\}$.

Decrement. *The lexical operation* $-\downarrow$ *is defined as follows:*

$$\alpha\downarrow = \begin{cases} \alpha_0^{\langle n-1\rangle} & \text{if } \alpha = \alpha_0^{\langle n\rangle},\ n \in \mathbb{N},\ n > 0 \\ \alpha & \text{otherwise} \end{cases}$$

We define the alphabet $\mathcal{A}\downarrow = \{\alpha\downarrow \mid \alpha \in \mathcal{A}\}$.

If a symbol is tagged more than once we will write the tags as follows:

$$\left(\alpha_0^{\langle\alpha_1\rangle}\right)^{\langle\alpha_2\rangle} = \alpha_0^{\langle\alpha_1\alpha_2\rangle}.$$

Definition 4.2 (Extended Regular Expressions) *The sets* $\mathcal{R}_{\mathcal{A}}$ *of extended regular expressions over finite alphabets* $\mathcal{A}$ *are defined inductively as the smallest sets for which:*

Constants. $\emptyset, \epsilon \in \mathcal{R}_{\mathcal{A}}$; *if* $\alpha \in \mathcal{A}$ *then* $\alpha \in \mathcal{R}_{\mathcal{A}}$.

Concatenation. *If* $R, R' \in \mathcal{R}_{\mathcal{A}}$ *then* $R \cdot R' \in \mathcal{R}_{\mathcal{A}}$.

Iteration. *If* $R \in \mathcal{R}_{\mathcal{A}}$, *then* $R^* \in \mathcal{R}_{\mathcal{A}}$.

Set operators. *If* $R, R' \in \mathcal{R}_{\mathcal{A}}$ *then* $R + R', R \cap R' \in \mathcal{R}_{\mathcal{A}}$.

Restriction. *If* $R \in \mathcal{R}_{\mathcal{A}},\ \mathcal{A}' \subseteq \mathcal{A}$ *then* $R \upharpoonright \mathcal{A}' \in \mathcal{R}_{\mathcal{A}'}$.

Substitution. *If* $R, R' \in \mathcal{R}_{\mathcal{A}},\ \omega \in \mathcal{A}^*$ *then* $R[\omega/R'] \in \mathcal{R}_{\mathcal{A}}$.

Tagging. *If* $R \in \mathcal{R}_{\mathcal{A}},\ \alpha \in \mathcal{A}$ *then* $R^{\langle\alpha\rangle} \in \mathcal{R}_{\mathcal{A}^{\langle\alpha\rangle}}$.

Increment/decrement. *If* $R \in \mathcal{R}_{\mathcal{A}}$, $R\uparrow \in \mathcal{R}_{\mathcal{A}\uparrow}$, $R\downarrow \in \mathcal{R}_{\mathcal{A}\downarrow}$.

Shuffle. *If* $R, R' \in \mathcal{R}_{\mathcal{A}}$ *then* $R \bowtie R' \in \mathcal{R}_{\mathcal{A}}$.

If $\mathcal{A}$ is a finite alphabet, so are $\mathcal{A}^{\langle\alpha\rangle}$, $\mathcal{A}\uparrow$, $\mathcal{A}\downarrow$.

Constant $\emptyset$ denotes the empty language; constant ϵ denotes the empty string. The constant α is the language of the singleton sequence. Restriction is removing from all sequences in the language of a regular expression all symbols not in $\mathcal{A}'$ (same as the game-semantic restriction on page 42).

The language of substitution $R[\omega/R']$ is the language of R where all occurrences of substring ω have been replaced by the strings of R'. If a finite number of substitutions must be performed simultaneously we can write either $R[\omega_1/R'_1]\cdots[\omega_n/R'_n]$ or $R[\kappa]$ where κ is the finite function $(\omega_1 \mapsto R'_1, \ldots, \omega_n \mapsto R'_n)$.

The tagging of a language is the tagging of all symbols in its strings (similarly increment, decrement). The shuffle of two regular languages is defined as:

Definition 4.3 (Shuffle)

$$\mathcal{L}_1 \bowtie \mathcal{L}_2 = \bigcup_{\omega_i \in \mathcal{L}_i,\ i=1,2} \omega_1 \bowtie \omega_2,$$

where

$$\begin{aligned} &\omega \bowtie \epsilon = \epsilon \bowtie \omega = \omega \\ &\alpha_1 \cdot \omega_1 \bowtie \alpha_2 \cdot \omega_2 = \alpha_1 \cdot (\omega_1 \bowtie \alpha_2 \cdot \omega_2) + \alpha_2 \cdot (\alpha_1 \cdot \omega_1 \bowtie \omega_2). \end{aligned}$$

Proposition 4.1 *Every extended regular expression $R \in \mathcal{R}_{\mathcal{A}}$ denotes a regular language over $\mathcal{A}$.*

PROOF: In addition to the normal regular expression operators $(\cdot, *, +)$ we have:

Set operators. Regular languages are closed under these set operations.

Restriction. From a finite-state automaton accepting R we can obtain the finite-state automaton accepting $R \restriction \mathcal{A}'$ by replacing all transitions on inputs $\alpha \notin \mathcal{A}'$ with ϵ-transitions.

Substitution. The language of $R[\omega/R']$ is, by definition, the image of a regular-language homomorphism. It is known that regular languages are closed under homomorphisms.

Tagging, increment, decrement. Same reason as for substitution.

Shuffle. The shuffle operation has been studied quite extensively, [Jan85] is a starting point in the literature. Shuffle is known to preserve regularity.

END OF PROOF.

We also need the notion of the *effective alphabet* of a regular expression, which is the set of all symbols appearing in the language denoted by that regular expression. The effective alphabet only depends on R.

Definition 4.4 (Effective alphabet) *For any $R \in \mathcal{R}_{\mathcal{A}_0}$, the effective alphabet of R is*

$$\lfloor R \rfloor = \{\alpha \in \mathcal{A}_0 \mid \omega \restriction \alpha \neq \epsilon \text{ for some } \omega \in R\}.$$

A regular expression is *broadened* by shuffling with all strings *not* in its effective alphabet.

Definition 4.5 (Broadening) $\widetilde{R} = R \bowtie (\mathcal{A} \setminus \lfloor R \rfloor)^*$. *The operation of broadening is relative to an alphabet $\mathcal{A}$, which must be appropriately specified in the context.*

Broadening will be used in modeling local variable declarations.

4.1.2 Interpretation of types

An alphabet $\mathcal{A}[\![-]\!]$ is associated with every data-type τ and ground-type σ; first-order types also have associated alphabets. The alphabets of types contain symbols $\mathsf{q} \in \mathsf{Q}[\![\theta]\!]$ called *questions*, and every question q has a set of of *answers*, $\mathsf{a} \in \mathsf{A}_{\mathsf{q}}[\![\theta]\!]$.

Definition 4.6 (Type alphabets)

$$\mathcal{A}[\![\mathbf{int}]\!] = \mathcal{Z} = \{-Z_{max}, \ldots, -1, 0, 1, \ldots, Z_{max}\} \subset \mathbb{Z}$$
$$\mathcal{A}[\![\mathbf{bool}]\!] = \{tt, \mathit{ff}\}$$
$$\mathsf{Q}[\![\mathbf{exp}\tau]\!] = \{q\}, \quad \mathsf{A}_q[\![\mathbf{exp}\tau]\!] = \mathcal{A}[\![\tau]\!]$$
$$\mathsf{Q}[\![\mathbf{var}\tau]\!] = \{read\} \cup \{write(\alpha) \mid \alpha \in \mathcal{A}[\![\tau]\!]\}, \quad \mathsf{A}_{read}[\![\mathbf{var}\tau]\!] = \mathcal{A}[\![\tau]\!], \quad \mathsf{A}_{write(\alpha)} = \{ok\}$$
$$\mathsf{Q}[\![\mathbf{comm}]\!] = \{run\}, \quad \mathsf{A}_{run}[\![\mathbf{comm}]\!] = \{done\}$$
$$\mathsf{Q}[\![\sigma_1 \to \cdots \to \sigma_k]\!] = \sum_{1 \le i \le k} \mathsf{Q}[\![\sigma]\!]^{\langle i \rangle}, \quad \mathsf{A}_{\mathsf{q}^{\langle i \rangle}}[\![\sigma_1 \to \cdots \to \sigma_k]\!] = (\mathsf{A}_{\mathsf{q}}[\![\sigma_i]\!])^{\langle i \rangle}, \quad 1 \le i \le k$$
$$\mathcal{A}[\![\theta]\!] = \mathsf{Q}[\![\theta]\!] \cup \bigcup_{\mathsf{q} \in \mathsf{Q}[\![\theta]\!]} \mathsf{A}_{\mathsf{q}}[\![\theta]\!].$$

We use meta-variables α to range over symbols, q over symbols which are questions and a over symbols which are answers.

Terms $\Gamma \vdash P : \theta$ are interpreted by an evaluation function $[\![-]\!]$ which maps them into a regular language. This regular language is defined over an alphabet induced by the environment:

Definition 4.7 (Environment alphabets)

$$\mathcal{A}[\![x : \theta]\!] = \mathcal{A}[\![\theta]\!]^{\langle x \rangle}$$
$$\mathcal{A}[\![\Gamma]\!] = \sum_{x:\theta \in \Gamma} \mathcal{A}[\![x : \theta]\!]$$
$$\mathcal{A}[\![\Gamma \vdash P : \theta]\!] = \mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![\theta]\!].$$

Example 4.1 $\mathcal{A}[\![f : \mathbf{comm} \to \mathbf{comm}]\!] = \{run^{\langle f \rangle}, done^{\langle f \rangle}, run^{\langle 1f \rangle}, done^{\langle 1f \rangle}\}$.

Every regular language that denotes the meaning of a term has a certain form, given by its possible initial and final moves. These moves indicate that a complete computation has occurred and give the result of the computation. In formulating the semantic definitions it is useful to introduce the auxiliary notation $(\!|-|\!)$ defined below, taking advantage of the fact that for any type the regular-language interpreting the term has a certain form. Exploiting this structure we can give more compact definitions.

Definition 4.8 (Semantic decompositions)

$$[\![\Gamma \vdash C : \mathbf{comm}]\!] = run \cdot (\!|\Gamma \vdash C : \mathbf{comm}|\!) \cdot done$$
$$[\![\Gamma \vdash E : \mathbf{exp}\tau]\!] = \sum_{\alpha \in [\![\tau]\!]} q \cdot (\!|\Gamma \vdash E : \mathbf{exp}\tau|\!)_\alpha \cdot \alpha$$
$$[\![\Gamma \vdash V : \mathbf{var}\tau]\!] = \sum_{\alpha \in [\![\tau]\!]} read \cdot (\!|\Gamma \vdash V : \mathbf{var}\tau|\!)_{\mathsf{r}\alpha} \cdot \alpha + \sum_{\alpha \in [\![\tau]\!]} write(\alpha) \cdot (\!|\Gamma \vdash V : \mathbf{var}\tau|\!)_{\mathsf{w}\alpha} \cdot ok.$$

Intuitively, the language $(\!|C|\!)$ is the actual computation performed by C; $(\!|E|\!)_\alpha$ is only that particular computation of expression E which produces value α as a result. Variables contain two kinds of computations: $(\!|V|\!)_{\mathsf{r}\alpha}$, which happen when V reads value α, and $(\!|V|\!)_{\mathsf{w}\alpha}$ which happen when V writes value α. The full meaning of a term $[\![P]\!]$ is then the union of all these possible traces $(\!|P|\!)$. If it does not cause confusion we may abbreviate the above notations to $[\![M : \theta]\!]$ or $[\![M]\!]$, and similarly for $(\!|-|\!)$.

4.1.3 Expressions and control structures

The regular-language interpretation of integer and boolean constants is:

Definition 4.9 (Constants)

$$\begin{array}{ll} (\!|n|\!)_n = \epsilon & (\!|n|\!)_{n'} = \emptyset,\ n' \neq n \\ (\!|\mathbf{true}|\!)_{\mathbf{true}} = \epsilon & (\!|\mathbf{true}|\!)_{\mathbf{false}} = \emptyset \\ (\!|\mathbf{false}|\!)_{\mathbf{false}} = \epsilon & (\!|\mathbf{false}|\!)_{\mathbf{true}} = \emptyset \end{array}$$

Definition 4.8 on the preceding page shows that the $[\![-]\!]$ and $(\!|-|\!)$ notations are equally expressive, because each can be formulated in terms of the other. The interpretations of the constants can also be expressed as:

$$[\![n]\!] = q \cdot n \qquad [\![\mathbf{true}]\!] = q \cdot tt \qquad [\![\mathbf{false}]\!] = q \cdot f\!\!f.$$

In the following, we will use whichever of the two forms is more convenient.

The definitions of IA arithmetic-logic operators is:

Definition 4.10 (Operators)

$$(\!|E_1 \star E_2 : \mathbf{exp}\tau'|\!)_\alpha = \sum_{\substack{\alpha_1, \alpha_2 \in \mathcal{A}[\![\tau]\!] \\ \alpha = \alpha_1 \star \alpha_2}} (\!|E_1 : \mathbf{exp}\tau|\!)_{\alpha_1} \cdot (\!|E_2 : \mathbf{exp}\tau|\!)_{\alpha_2}, \quad \alpha \in [\![\tau']\!],$$

Arithmetic operators over a finite set of integers can be interpreted in several ways. The first possibility is to have all operators *modulo* some maximum value, like in JAVA or C++. The second possibility is to leave the operators undefined if the value produced is out of range. This identifies the run-time error of numerical overflow with divergence, which is an expedient approximation, coarse but not entirely unacceptable (see for example [Rey98, Sections 2.7 and 5.1] for a discussion). A third possibility is to use the special values of **Infty**, **−Infty** and **NaN** to denote positive respective negative overflow, or an indeterminate result. This approach is common in floating point operations (e.g. ANSI/IEEE Standard 754-1985). All these approaches to handling overflow are compatible with the regular-language semantics, in that each can be modeled with appropriate changes to the semantic details.

Another semantic detail packaged in the definitions above is order of evaluation, left-to-right. Defining similar, but right-to-left, operators using this style of semantics can be done in the obvious way. Also, for logical operators, we have similar choices regarding lazy or eager implementation of operators. Non-deterministic operators are also relatively easy to introduce. However, operators with parallel evaluation require substantial revisions of the semantic framework, which is unsurprising.

The imperative features are interpreted by:

Definition 4.11 (Commands)

$$
\begin{aligned}
&(\!|\mathbf{skip}|\!) = \epsilon \\
&(\!|\mathbf{diverge}|\!) = \emptyset \\
&(\!|C; C'|\!) = (\!|C|\!) \cdot (\!|C'|\!) \\
&(\!|C; M|\!)_\alpha = (\!|C|\!) \cdot (\!|M|\!)_\alpha \\
&(\!|\mathbf{while}\ B\ \mathbf{do}\ C|\!) = \big((\!|B|\!)_{tt} \cdot (\!|C|\!)\big)^* \cdot (\!|B|\!)_{f\!f} \\
&(\!|\mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C'|\!) = (\!|B|\!)_{tt} \cdot (\!|C|\!) + (\!|B|\!)_{f\!f} \cdot (\!|C'|\!)
\end{aligned}
$$

We can now consider a simple, standard example.

Example 4.2 $\Gamma \vdash$ **while true do** $C \equiv_{\mathbf{comm}}$ **diverge**.

PROOF:

$$
\begin{aligned}
[\![\mathbf{while\ true\ do}\ C]\!] &= \mathit{run} \cdot \big((\!|\mathbf{true}|\!)_{tt} \cdot (\!|C|\!)\big)^* \cdot (\!|\mathbf{true}|\!)_{f\!f} \cdot \mathit{done} \\
&= \mathit{run} \cdot \big(\epsilon \cdot (\!|C|\!)\big)^* \cdot \emptyset \cdot \mathit{done} \\
&= \emptyset = [\![\mathbf{diverge}]\!]\,.
\end{aligned}
$$

END OF PROOF

The regular language semantics reveals some computational intuitions which are interesting in their own right. For example, **skip** is interpreted by the bracketing moves for commands enclosing the empty string. This suggests that it is a command which completes without having any effects. The regular expression interpreting any arithmetic-logic operator is decomposed into α-producing plays, where every such play is any concatenation of plays producing α_1 and α_2 in the arguments, if and only if $\alpha_1 \star \alpha_2 = \alpha$. Composition of commands is simply concatenation of plays. Looping is interpreted as an iteration of plays in the guard of the loop producing true concatenated with complete plays in the body, followed by one single play in the guard, producing false. Remarkably, this is exactly the traced-based interpretation used to interpret iteration as early as the '70s (see for example Section 2.3.4 in [Har79]). It is also similar to Brookes's trace-based interpretation of parallel IA [Bro93]. Non-termination **diverge** is interpreted as the empty set of complete plays.

4.1.4 Free identifiers and functions

Free identifiers, of ground and function type, are given a concrete interpretation; that is, they are represented by a regular language, just like a closed term. This "flattening" of the semantics, so that no higher-order entities such as functions or quantifiers are needed in the model, is arguably the most remarkable feature of game semantics.

In order to interpret free identifiers, we use regular languages which represent the all-important *copy-cat* strategies introduced in Definition 3.10 on page 46.

Definition 4.12 (Copy-cat) *The copy-cat regular languages $\mathcal{K}^\alpha_\theta$ where α is an arbitrary symbol, are defined as*

$$\mathcal{K}^\alpha_{\sigma_1 \to \cdots \to \sigma_k} = \sum_{\mathsf{q} \in \mathsf{Q}[\![\sigma_k]\!]} \sum_{\mathsf{a} \in \mathsf{A}_\mathsf{q}[\![\sigma_k]\!]} \mathsf{q} \cdot \mathsf{q}^{\langle\alpha\rangle} \cdot \left(\sum_{1 \le j < k} \left(\mathcal{L}^{\alpha,j}_{\sigma_j} \right) \right)^* \cdot \mathsf{a}^{\langle\alpha\rangle} \cdot \mathsf{a},$$

where

$$\mathcal{L}^{\alpha,j}_\sigma = \sum_{\mathsf{q} \in \mathsf{Q}[\![\sigma]\!]} \sum_{\mathsf{a} \in \mathsf{A}_\mathsf{q}[\![\sigma]\!]} \mathsf{q}^{\langle j\alpha\rangle} \cdot \mathsf{q}^{\langle j\rangle} \cdot \mathsf{a}^{\langle j\rangle} \cdot \mathsf{a}^{\langle j\alpha\rangle}.$$

The languages $\mathcal{L}^{\alpha,j}_\sigma$ are traces representing a function using an argument; the languages $\mathcal{K}^\alpha_{\sigma_1 \to \cdots \to \sigma_k}$ represent all the possible ways in which a function can use its arguments.

Then, the definition of free identifiers is:

Definition 4.13 (Identifiers) $[\![\Gamma, x : \theta \vdash x : \theta]\!] = \mathcal{K}^x_\theta$.

Example 4.3

$$\begin{aligned}&[\![f : \mathbf{comm} \to \mathbf{comm} \vdash f : \mathbf{comm} \to \mathbf{comm}]\!] \\ &= run \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot done^{\langle 1f\rangle})^* \cdot done^{\langle f\rangle} \cdot done = \mathcal{K}^f_{\mathbf{comm}\to\mathbf{comm}}.\end{aligned}$$

Conceptually, the moves tagged with f represent the effects of calling then returning from the function; moves tagged by $1f$ are the effects caused by f whenever it evaluates its first, and in this example its only, argument. The argument may be evaluated an arbitrary number of times, sequentially (no interleaving), hence the Kleene closure. The moves with only numerical tags correspond to the formal parameters.

Example 4.4

$$\begin{aligned}&[\![f : \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm} \vdash f : \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm}]\!] = \mathcal{K}^f_{\mathbf{comm}\to\mathbf{comm}\to\mathbf{comm}} \\ &= run \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot done^{\langle 1f\rangle} + run^{\langle 2f\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} \cdot done^{\langle 2f\rangle})^* \cdot done^{\langle f\rangle} \cdot done.\end{aligned}$$

The example above illustrates why the games model gives an intuitively nice account of sequentiality, because it makes obvious the property that function f can only evaluate one of its arguments after it has completed the evaluation of the other argument:

$$run \cdot run^{\langle f\rangle} \cdot \overbrace{run^{\langle 1f\rangle} \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot done^{\langle 1f\rangle}}^{\text{evaluate 1st argument}} \cdot \overbrace{run^{\langle 2f\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} \cdot done^{\langle 2f\rangle}}^{\text{evaluate 2nd argument}} \cdot done^{\langle f\rangle} \cdot done \\ \in \mathcal{K}^f_{\mathbf{comm}\to\mathbf{comm}\to\mathbf{comm}}$$

$$run \cdot run^{\langle f\rangle} \cdot \overbrace{run^{\langle 1f\rangle} \cdot run^{\langle 1\rangle} \cdot \underbrace{run^{\langle 2f\rangle} \cdot run^{\langle 2\rangle} \cdot done^{\langle 2\rangle} \cdot done^{\langle 2f\rangle}}_{\text{evaluate 2nd argument}} \cdot done^{\langle 1\rangle} \cdot done^{\langle 1f\rangle}}^{\text{evaluate 1st argument}} \cdot done^{\langle f\rangle} \cdot done \\ \notin \mathcal{K}^f_{\mathbf{comm}\to\mathbf{comm}\to\mathbf{comm}}.$$

Abstraction is interpreted as a re-tagging of symbols in the language. Conceptually, this corresponds to the "moving" of the identifier from the environment to the term. This rule is another instance of the remarkable "flatness" and concreteness of the game semantics.

Definition 4.14 (Abstraction) $[\![\Gamma \vdash \lambda m{:}\sigma.P : \sigma \to \theta]\!] = \left([\![\Gamma, m : \sigma \vdash P : \theta]\!]\uparrow\right)[\kappa]$, *where* $\kappa : \mathcal{A}\,[\![\sigma]\!]^{\langle m\rangle} \to \mathcal{A}\,[\![\sigma]\!]^{\langle 1\rangle}$, $\kappa\left(\alpha^{\langle m\rangle}\right) = \alpha^{\langle 1\rangle}$.

The moves associated with m become "anonymous," and are tagged with a number. In order to keep the tags unique, all other symbols are incremented.

Application is modeled by trace-level substitution:

Definition 4.15 (Application) $[\![PM : \theta]\!] = \left([\![P : \sigma \to \theta]\!]\,[\kappa]\right)\downarrow$, *where* $\kappa\left(\mathsf{q}^{\langle 1\rangle} \cdot \mathsf{a}^{\langle 1\rangle}\right) = \{w \mid \mathsf{q} \cdot w \cdot \mathsf{a} \in [\![M : \sigma]\!]\}$.

The moves corresponding to the outermost identifier bound by lambda are tagged with 1, so upon application the pairs of symbols corresponding to the formal parameter are substituted by the concrete traces of the argument. The rest of the indices are decremented. This mechanism is quite similar to the representation of lambda calculus using de Bruijn indices [Bru72, Bru79].

Example 4.5

$$\begin{aligned}
[\![(\lambda x{:}\mathbf{expint}.x+1)7]\!] &= \left([\![\lambda x{:}\mathbf{expint}.x+1]\!]\,[\kappa]\right)\downarrow \\
&= \Big(\big(([\![x : \mathbf{expint} \vdash x+1]\!])\uparrow[\kappa']\big)[\kappa]\Big)\downarrow \\
&= \Big(\big((\sum_{n,n'\in\mathcal{N}} q \cdot (\!|x : \mathbf{expint} \vdash x|\!) \cdot (\!|1|\!)_{n'} \cdot (n+n'))\uparrow[\kappa']\big)[\kappa]\Big)\downarrow \\
&= \Big(\big((\sum_{n\in\mathcal{N}} q \cdot q^{\langle x\rangle} \cdot n^{\langle x\rangle} \cdot (n+1))\uparrow[\kappa']\big)[\kappa]\Big)\downarrow \\
&= \Big(\big((\sum_{n\in\mathcal{N}} q \cdot q^{\langle x\rangle} \cdot n^{\langle x\rangle} \cdot (n+1))[\kappa']\big)[\kappa]\Big)\downarrow \\
&= \Big((\sum_{n\in\mathcal{N}} q \cdot q^{\langle 1\rangle} \cdot n^{\langle 1\rangle} \cdot (n+1))[\kappa]\Big)\downarrow \\
&= \Big((\sum_{n\in\mathcal{N}} q \cdot q^{\langle 1\rangle} \cdot n^{\langle 1\rangle} \cdot (n+1))[\kappa]\Big)\downarrow \\
&= (q \cdot 8)\downarrow \\
&= q \cdot 8,
\end{aligned}$$

where $\kappa'\left(q^{\langle x\rangle} \cdot n^{\langle x\rangle}\right) = q^{\langle 1\rangle} \cdot n^{\langle 1\rangle}$ *and* $\kappa\left(q^{\langle 1\rangle} \cdot n^{\langle 1\rangle}\right) = \begin{cases} \epsilon & \text{if } n = 7 \\ \emptyset & \text{if } n \neq 7. \end{cases}$

4.1.5 Store

Reading and writing to a variable is achieved by dereferencing and assignment, respectively.

Definition 4.16 (Variable manipulation)

$$\begin{aligned}
(\!|\Gamma \vdash\ !V : \mathbf{exp}\tau|\!)_\alpha &= (\!|\Gamma \vdash V : \mathbf{var}\tau|\!)_{\mathsf{r}\alpha}, \quad \alpha \in \mathcal{A}[\![\tau]\!] \\
(\!|\Gamma \vdash V := E : \mathbf{comm}|\!) &= \sum_{\alpha \in \mathcal{A}[\![\tau]\!]} (\!|\Gamma \vdash E : \mathbf{exp}\tau|\!)_\alpha \cdot (\!|\Gamma \vdash V : \mathbf{var}\tau|\!)_{\mathsf{w}\alpha}.
\end{aligned}$$

Notice that the semantics above imposes no causal correlation between the reads and writes of variables. For example, the expression with side-effects $v := 1; !v$ has the interpretation:

$$[\![v : \mathbf{varint} \vdash v := 1; !v]\!] = \sum_{n \in \mathcal{N}} q \cdot \mathit{write}(1)^{\langle v \rangle} \cdot \mathit{ok}^{\langle v \rangle} \cdot \mathit{read}^{\langle v \rangle} \cdot n^{\langle v \rangle} \cdot n.$$

In other words, upon writing 1 to the variable it is still possible to get any value when reading the variable. Why is this possible? The reason is that the variable-identifier v may be bound by function application to *any* variable-typed term, e.g.:

$$(\lambda v{:}\mathbf{varint}.v := 1; !v)(\mathbf{if}\ !x = 1\ \mathbf{then}\ x := 7; x\ \mathbf{else}\ x := 0; x).$$

Even worse than in the case of side-effect free IA, variables are not even guaranteed to return the same value upon consecutive readings. The reason is the same, the variable-identifier may be bound to a phrase that has side-effects which may include changing the variable itself.

Only variables that are known to be locally declared in the evaluation context are guaranteed to be "well-behaved" in the sense that there is an expected causal connection between the values that are read and the values that are written. This property is captured by the following regular language:

Definition 4.17 (Variable stability)

$$\gamma^v_{\mathbf{var}\tau} = (\mathit{read}^{\langle v \rangle} \cdot \alpha_\tau^{\langle v \rangle})^* \cdot \Big(\sum_{\alpha \in \mathcal{A}[\![\tau]\!]} \mathit{write}(\alpha)^{\langle v \rangle} \cdot \mathit{ok}^{\langle v \rangle} \cdot (\mathit{read}^{\langle v \rangle} \cdot \alpha^{\langle v \rangle})^* \Big)^*,$$

where $\alpha_{\mathbf{int}} = 0,\ \alpha_{\mathbf{bool}} = \mathbf{false}$.

Initially, the value read from the variable v is the default value α_τ. Any legal sequence consists of a *write* followed by an arbitrary number of *read*s, all yielding the value that was written. In the terminology used by Reynolds, a stable variable is a variable which is *good* and *not interfered with.*

The semantics of the local-variable block consists of two operations: imposing the stable-variable behaviour and then removing all occurrences of actions of that variable as it becomes invisible outside its binding scope:

Definition 4.18 (Block variable)

$$[\![\Gamma \vdash \mathbf{new}\tau\ v\ \mathbf{in}\ M : \sigma]\!] = \left([\![\Gamma \mid v : \mathbf{var}\tau \vdash M : \sigma]\!] \cap \widetilde{\gamma^{v}_{\mathbf{var}\tau}}\right) \upharpoonright \mathcal{A}^{\overline{v}},$$

where $\mathcal{A} = \mathcal{A}\,[\![\Gamma \mid v : \mathbf{var}\tau \vdash M : \sigma]\!]$.

The actions not tagged by v are not constrained. All other actions are not constrained by stability, and they depend on the context. They are introduced at the point of the definition of the variable in a block using the broadening operation (Definition 4.5 on page 57). Scope is modeled by restriction, which hides away all interactions of v (Definition 4.2 on page 56). The following section contains numerous example showing this definition at work.

4.2 Examples of equational reasoning

We have presented so far a substantial part of the semantics of first order IA. The part that is missing, introduced later in Section 4.4, is function definition. Although the language fragment presented so far is not complete, it contains all the definitions necessary to prove the example equivalences seen earlier, in Section 2.3.

Example 4.6 (Meyer-Sieber [MS88, Example 1])

$$c : \mathbf{comm} \vdash \mathbf{newint}\ v\ \mathbf{in}\ c \equiv_{\mathbf{comm}} c.$$

PROOF: This presentation of the example is slightly different from the original one, for the sake of simplicity; the original presentation makes the operational proof simpler. But it is immediate from the Operational Extensionality Theorem (2.1) that the two formulations are equivalent.

$$\begin{aligned}
[\![c : \mathbf{comm} \vdash \mathbf{newint}\ v\ \mathbf{in}\ c : \mathbf{comm}]\!] &= \left([\![c : \mathbf{comm}, v : \mathbf{var}\tau \vdash c : \mathbf{comm}]\!] \cap \widetilde{\gamma^{v}_{\tau}}\right) \upharpoonright \mathcal{A}^{\overline{v}} \\
&= \left(run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \cap \widetilde{\gamma^{v}_{\tau}}\right) \upharpoonright \mathcal{A}^{\overline{v}} \\
&= run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \\
&\text{because } run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \in (\mathcal{A}^{\overline{v}})^* \\
&= [\![c : \mathbf{comm} \vdash c : \mathbf{comm}]\!].
\end{aligned}$$

END OF PROOF.

Example 4.7 (Meyer-Sieber [MS88, Example 3])

$$c : \mathbf{comm} \vdash \begin{array}{l}\mathbf{newint}\ v_1\ \mathbf{in} \\ \quad \mathbf{newint}\ v_2\ \mathbf{in}\ c\end{array} \equiv_{\mathbf{comm}} \begin{array}{l}\mathbf{newint}\ v_2\ \mathbf{in} \\ \quad \mathbf{newint}\ v_1\ \mathbf{in}\ c.\end{array}$$

PROOF: As before, the presentation is slightly different but, given operational extensionality, equivalent:

$$
\begin{aligned}
&[\![c : \mathbf{comm} \vdash \mathbf{newint}\ v_1\ \mathbf{in}\ \mathbf{newint}\ v_2\ \mathbf{in}\ c : \mathbf{comm}]\!]\\
&\quad = \left([\![c : \mathbf{comm}, v_1 : \mathbf{varint} \vdash \mathbf{newint}\ v_2\ \mathbf{in}\ c : \mathbf{comm}]\!] \cap \widetilde{\gamma^{v_1}_{\mathbf{int}}}\right) \restriction \mathcal{A}^{\overline{v_1}}\\
&\quad = \left(([\![c : \mathbf{comm}, v_1 : \mathbf{varint}, v_2 : \mathbf{varint} \vdash c : \mathbf{comm}]\!] \cap \widetilde{\gamma^{v_2}_{\mathbf{int}}} \restriction \mathcal{A}^{\overline{v_2}}) \cap \widetilde{\gamma^{v_1}_{\mathbf{int}}}\right) \restriction \mathcal{A}^{\overline{v_1}}\\
&\quad = \left((run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \cap \widetilde{\gamma^{v_2}_{\mathbf{int}}} \restriction \mathcal{A}^{\overline{v_2}}) \cap \widetilde{\gamma^{v_1}_{\mathbf{int}}}\right) \restriction \mathcal{A}^{\overline{v_1}}\\
&\quad = run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done\\
&\quad\ \text{because } run \cdot run^{\langle c\rangle} \cdot done^{\langle c\rangle} \cdot done \in \left(\mathcal{A}^{\overline{v_i}}\right)^*, i = 1, 2\\
&\quad = [\![c : \mathbf{comm} \vdash \mathbf{newint}\ v_2\ \mathbf{in}\ \mathbf{newint}\ v_1\ \mathbf{in}\ c : \mathbf{comm}]\!]
\end{aligned}
$$

END OF PROOF.

Example 4.8 (O'Hearn-Reynolds [OR00, Section 7.1])

$$
f : \mathbf{comm} \to \mathbf{comm} \vdash \quad \begin{array}{l} \mathbf{newint}\ v\ \mathbf{in} \\ \quad v := 0; f(v := 1); \\ \quad \mathbf{if}\ !v = 1\ \mathbf{then\ diverge\ else\ skip} \end{array} \quad \equiv_{\mathbf{comm}} f(\mathbf{diverge}).
$$

PROOF:
We proceed in a "bottom up" fashion. The following evaluation is routine:

$$
\begin{aligned}
&[\![v : \mathbf{varint}, f : \mathbf{comm} \to \mathbf{comm} \vdash f(v := 1) : \mathbf{comm}]\!]\\
&= (run \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot done^{\langle 1f\rangle})^* \cdot done^{\langle f\rangle} \cdot done)\\
&\qquad [run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} / write(1)^{\langle v\rangle} \cdot ok^{\langle v\rangle}]\\
&= run \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot write(1)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle})^* \cdot done^{\langle f\rangle} \cdot done.
\end{aligned}
$$

The following evaluation is also routine:

$$
[\![\mathbf{if}\ !v = 1\ \mathbf{then\ diverge\ else\ skip} : \mathbf{comm}]\!] \;=\; \sum_{1 \neq \alpha \in \mathcal{N}} run \cdot read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \cdot done.
$$

Using the two above, we have that:

$$
\begin{aligned}
&[\![v := 0; f(v := 1); \mathbf{if}\ !v = 1\ \mathbf{then\ diverge\ else\ skip} : \mathbf{comm}]\!]\\
&= run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot write(1)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle})^* \cdot done^{\langle f\rangle}\\
&\qquad \cdot \sum_{1 \neq \alpha \in \mathcal{N}} read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \cdot done.
\end{aligned}
$$

The first part of the interpretation of v as a block variable is the intersection with $\widetilde{\gamma^{v}_{\mathbf{int}}}$. We notice that:

- if the iteration $(*)$ is empty then the stability of v forces all the subsequent *read*s to produce 0;
- if the iteration is non-empty then the stability of v forces the subsequent *read*s to produce 1; but the condition $\alpha \neq 1$ stipulates that 1 cannot be produced. Therefore the entire trace is empty in this case.

So,

$$\begin{aligned} & run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot (run^{\langle 1f\rangle} \cdot write(1)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle})^* \cdot done^{\langle f\rangle} \\ & \qquad \cdot \sum_{1\neq\alpha\in\mathcal{N}} read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \cdot done \cap \widetilde{\gamma^{\langle v\rangle}_{\mathbf{int}}} \\ & = run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot done^{\langle f\rangle} \cdot read^{\langle v\rangle} \cdot 0^{\langle v\rangle} \cdot done. \end{aligned}$$

After restriction to $\mathcal{A}^{\overline{v}}$, we have that

$$[\![LHS]\!] = run \cdot run^{\langle f\rangle} \cdot done^{\langle f\rangle} \cdot done.$$

But it can be immediately seen that this is also the interpretation of the right-hand side.

End Of Proof.

Example 4.9 (Stoughton [MS88, Example 5])

$$f : \mathbf{comm} \to \mathbf{comm} \vdash \begin{array}{l} \mathbf{newint}\ v\ \mathbf{in} \\ \quad v := 0; f(v := !v + 2); \\ \quad \mathbf{if}\ !v\ \mathbf{mod}\ 2 = 0\ \mathbf{then\ diverge\ else\ skip} \end{array} \equiv \mathbf{diverge}.$$

Proof:
Following the same bottom-up approach, it is routine to evaluate

$$\left[\!\!\left[f : \mathbf{comm} \to \mathbf{comm} \vdash \begin{array}{l} v := 0; f(v := !v + 2); \\ \mathbf{if}\ !v\ \mathbf{mod}\ 2 = 0\ \mathbf{then\ diverge\ else\ skip} : \mathbf{comm} \end{array} \right]\!\!\right]$$

to

$$\begin{aligned} & run \cdot write(0)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot run^{\langle f\rangle} \cdot \Big(\sum_{\alpha\in\mathcal{N}} run^{\langle 1f\rangle} \cdot read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle} \\ & \cdot write(\alpha + 2)^{\langle v\rangle} \cdot ok^{\langle v\rangle} \cdot done^{\langle 1f\rangle}\Big)^* \cdot done^{\langle f\rangle} \cdot \Big(\sum_{\alpha\in\mathcal{N},\ \alpha \bmod 2=1} read^{\langle v\rangle} \cdot \alpha^{\langle v\rangle}\Big) \cdot done. \end{aligned}$$

It is immediately seen that upon introducing the stable-variable constraint for v this regular expression becomes $\emptyset$, because for any $k \geq 0$ iterations the value stored in v is even, equal to $2 \times k$. This contradicts the clause α mod $2 = 1$ in the second part of the trace.

Therefore $[\![LHS]\!] = \emptyset = [\![\mathbf{diverge}]\!]$.

End Of Proof.

Example 4.10 (Oles [Ole82, adapted])

$$f : \mathbf{comm} \to \mathbf{expbool} \to \mathbf{comm} \vdash \begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ v := 0;\\ f(v := 1, !v = 0)\end{array} \equiv_{\mathbf{comm}} \begin{array}{l}\mathbf{newint}\ v\ \mathbf{in}\\ v := 0;\\ f(v := -1, !v = 0)\end{array}$$

PROOF:
As in the previous examples, evaluating the phrase bottom-up is mechanical:

$$[\![f : \mathbf{comm} \to \mathbf{expbool} \to \mathbf{comm} \vdash v := 0; f(v := 1, !v = 0) : \mathbf{comm}]\!]$$

is given by

$$\begin{aligned} & run \cdot write(0)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot run^{\langle f \rangle} \cdot \Big(\overbrace{run^{\langle 1f \rangle} \cdot write(1)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot done^{\langle 1f \rangle}}^{\mathcal{R}_{write}} \\ & \quad + \underbrace{q^{\langle 2f \rangle} \cdot read^{\langle v \rangle} \cdot 0^{\langle v \rangle} \cdot tt^{\langle 2f \rangle}}_{\mathcal{R}_{tt}} + \underbrace{\sum_{0 \neq \alpha \in \mathcal{N}} q^{\langle 2f \rangle} \cdot read^{\langle v \rangle} \cdot \alpha^{\langle v \rangle} \cdot \mathit{ff}^{\langle 2f \rangle}}_{\mathcal{R}_{\mathit{ff}}} \Big)^* \cdot done^{\langle f \rangle} \cdot done. \end{aligned}$$

When the stability property for v is imposed it is obvious that in the iterated part, R_{tt} must occur only before R_{write}, and R_{ff} only after. So

$$\begin{aligned} & \widetilde{\gamma^v_{\mathbf{int}}} \cap \Big(run \cdot write(0)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot run^{\langle f \rangle} \cdot \Big(\overbrace{run^{\langle 1f \rangle} \cdot write(1)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot done^{\langle 1f \rangle}}^{\mathcal{R}_{write}} \\ & \quad + \underbrace{q^{\langle 2f \rangle} \cdot read^{\langle v \rangle} \cdot 0^{\langle v \rangle} \cdot tt^{\langle 2f \rangle}}_{\mathcal{R}_{tt}} + \underbrace{\sum_{0 \neq \alpha \in \mathcal{N}} q^{\langle 2f \rangle} \cdot read^{\langle v \rangle} \cdot \alpha^{\langle v \rangle} \cdot \mathit{ff}^{\langle 2f \rangle}}_{\mathcal{R}_{\mathit{ff}}} \Big)^* \cdot done^{\langle f \rangle} \cdot done \Big) \end{aligned}$$

$$= run \cdot write(0)^{\langle v \rangle} \cdot ok^{\langle v \rangle} \cdot run^{\langle f \rangle} \cdot \Big(\mathcal{R}_{tt}^* \cdot \mathcal{R}_{write}^* \cdot \big(\epsilon + \mathcal{R}_{write} \cdot (\mathcal{R}_{write} + \mathcal{R}_{\mathit{ff}})^* \big) \Big) \cdot done^{\langle f \rangle} \cdot done.$$

Restriction to $\mathcal{A}^{\overline{v}}$ gives $[\![LHS]\!]$:

$$\begin{aligned} & run \cdot run^{\langle f \rangle} \cdot \Big((q^{\langle 2f \rangle} \cdot tt^{\langle 2f \rangle})^* \cdot (run^{\langle 1f \rangle} \cdot done^{\langle 1f \rangle})^* \\ & \qquad \cdot \big(\epsilon + run^{\langle 1f \rangle} \cdot done^{\langle 1f \rangle} \cdot (run^{\langle 1f \rangle} \cdot done^{\langle 1f \rangle} + q^{\langle 2f \rangle} \cdot \mathit{ff}^{\langle 2f \rangle})^* \big) \Big) \cdot done^{\langle f \rangle} \cdot done. \end{aligned}$$

Notice that this interpretation captures the dynamics of the term quite well. In English, it shows that in function f the second argument will always evaluate to true, until the first argument is evaluated; thereafter the second argument will evaluate to false, regardless of whether the first argument is evaluated or not.

Evaluating RHS gives the same regular expression. END OF PROOF.

Example 2.6 on page 25 is similar and we will not prove it. Instead, we will show the proof of another example which was also first proved using the O'Hearn and Tennent parametricity model.

Example 4.11 (O'Hearn-Tennent [OT93a, Sec. 5, ex. 1])

$$f : \mathbf{comm} \to \mathbf{comm} \vdash \mathbf{newint}\ v\ \mathbf{in}\ f(v := !v + 1) \equiv_{\mathbf{comm}} f(\mathbf{skip}).$$

This example seems just an instance of the locality property, but the fact that the state keeps changing with every argument evaluation creates technical problems.

PROOF:
We will impose the local-variable constraint on

$$\begin{aligned}&[\![f : \mathbf{comm} \to \mathbf{comm} \vdash f(v := !v + 1) : \mathbf{comm}]\!] \\ = {}& \mathit{run} \cdot \mathit{run}^{\langle f \rangle} \cdot \Big(\sum_{\alpha \in \mathcal{N}} \mathit{run}^{\langle 1f \rangle} \cdot \mathit{read}^{\langle v \rangle} \cdot \alpha^{\langle v \rangle} \mathit{write}(\alpha+1)^{\langle v \rangle} \cdot \mathit{ok}^{\langle v \rangle} \cdot \mathit{done}^{\langle 1f \rangle}\Big)^* \cdot \mathit{done}^{\langle f \rangle} \cdot \mathit{done}.\end{aligned}$$

After imposing the local variable constraint the result would become a set of traces of the form:

$$\begin{aligned}\mathit{run} \cdot \mathit{run}^{\langle f \rangle} \cdot \mathit{run}^{\langle 1f \rangle} \cdot \mathit{read}^{\langle v \rangle} \cdot 0^{\langle v \rangle} \cdot \mathit{write}(1)^{\langle v \rangle} \cdot \mathit{ok}^{\langle v \rangle} \cdot \mathit{done}^{\langle 1f \rangle} \cdot \\ \mathit{run}^{\langle 1f \rangle} \cdot \mathit{read}^{\langle v \rangle} \cdot 1^{\langle v \rangle} \cdot \mathit{write}(2)^{\langle v \rangle} \cdot \mathit{ok}^{\langle v \rangle} \cdot \mathit{done}^{\langle 1f \rangle} \cdot \\ \mathit{run}^{\langle 1f \rangle} \cdot \mathit{read}^{\langle v \rangle} \cdot 2^{\langle v \rangle} \cdot \mathit{write}(3)^{\langle v \rangle} \cdot \mathit{ok}^{\langle v \rangle} \cdot \mathit{done}^{\langle 1f \rangle} \cdots \mathit{done}^{\langle f \rangle} \cdot \mathit{done}\end{aligned}$$

Notice that here the result depends on how addition is implemented over a finite set (as discussed on page 59):

- if overflow is interpreted as divergence then the meaning of the LHS is

  $$\mathit{run} \cdot \mathit{run}^{\langle f \rangle} \cdot \Big(\sum_{k \le N_{\max}} \underbrace{\mathit{run}^{\langle 1f \rangle} \cdot \mathit{done}^{\langle 1f \rangle} \cdots \mathit{run}^{\langle 1f \rangle} \cdot \mathit{done}^{\langle 1f \rangle}}_{k \text{ times}}\Big) \cdot \mathit{done}^{\langle f \rangle} \cdot \mathit{done},$$

  which is *not* the same as $[\![RHS]\!]$. So, as pointed out in the case of Example 2.6 on page 25, if overflow leads to abortion then the equivalence actually fails.

- if overflow is interpreted using special values or "wrap-around" then traces of arbitrary length are possible. So, after restriction to $\mathcal{A}^{\overline{v}}$:

  $$[\![LHS]\!] = \mathit{run} \cdot \mathit{run}^{\langle f \rangle} \cdot (\mathit{run}^{\langle 1f \rangle} \cdot \mathit{done}^{\langle 1f \rangle})^* \cdot \mathit{done}^{\langle f \rangle} \cdot \mathit{done} = [\![RHS]\!],$$

  so the equivalence stands. END OF PROOF.

4.3 Relation to game semantics

In this section we will see that the regular-language semantics is a model for first-order IA. We show that this is the case because this model is isomorphic to the fully abstract Abramsky-McCusker game model.

In the following we will use $[\![-]\!]$ to denote a regular-language interpretation and $[\![-]\!]^{\mathsf{comp}}$ to denote the game interpretation as used in Theorem 3.2 on page 54 (sets of complete plays).

For any type $\theta = \sigma_1 \to \cdots \to \sigma_k \to \sigma$ of first-order IA, the set of legal plays of its game representation ∂_θ is isomorphic to a regular language. Let us call this regular language $\mathcal{R}[\![\theta]\!]$. The isomorphism ρ_0 consists of the tagging of all moves in the game model of σ_j with j.

Lemma 4.1 (Type representation)

$$\mathsf{P}^{\mathsf{comp}}_\theta \overset{\rho_0}{\cong} \mathcal{R}[\![\theta]\!]\,,$$

where by $\mathsf{P}^{\mathsf{comp}}_\theta$ *we denote the set of* complete plays *in* σ*.*

PROOF: (by induction on the structure of θ)

Ground types are interpreted by finite sets of complete plays, which are by definition regular. Let $\mathcal{R}[\![\sigma]\!] = \mathsf{P}^{\mathsf{comp}}_\sigma$.

First-order types of the form $\sigma_1 \to \cdots \to \sigma_k \to \sigma$ have sets of complete plays isomorphic to the regular language

$$\mathcal{R}[\![\sigma_1 \to \cdots \to \sigma_k \to \sigma]\!] = \sum_{\mathsf{q},\mathsf{a}} \mathsf{q} \cdot \left(\sum_{1 \le j \le k} \mathcal{R}[\![\sigma_j]\!]^{\langle j \rangle} \right)^* \cdot \mathsf{a}, \tag{4.1}$$

where $\mathsf{q}, \mathsf{a} \in \mathsf{M}_\sigma$, $\lambda^{\mathsf{QA}}_\sigma(\mathsf{q}) = \mathsf{Q}$, $\lambda^{\mathsf{QA}}_\sigma(\mathsf{a}) = \mathsf{A}$. This is because the opening question must be in σ, followed by an arbitrary number of repetitions of a complete set of plays in any σ_j. Moves of σ_j are tagged with j as part of the disjoint summing of the moves of the games; this is consistent with the definition of ρ_0. END OF PROOF.

For any term $x_0 : \theta_0, \ldots, x_k : \theta_k \vdash M : \sigma_1 \to \cdots \to \sigma_k \to \sigma$, let isomorphism ρ_1, from sets of moves to alphabets, be:

- the unique tagging of the moves in θ_j with x_j and, additionally, the tagging with i of all moves in $\sigma_{(j,i)}$, where $\theta_j = \sigma_{(j,0)} \to \cdots \to \sigma_{(j,k_j)} \to \sigma'_j$
- the unique tagging with j of all moves in σ_j.

Lemma 4.2 (Term representation) *For any first-order IA term:*

$$[\![\Gamma \vdash M : \theta]\!] \overset{\rho_1}{\cong} [\![\Gamma \vdash M : \theta]\!]^{\mathsf{comp}}\,.$$

PROOF:
(by induction on the derivation of $\Gamma \vdash M : \theta$)

Language constants. The sets of complete plays for

$$\mathbf{k} ::= n \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{skip} \mid \mathbf{diverge},$$

are finite and we can see by inspection that $[\![\mathbf{k}]\!] = [\![\mathbf{k}]\!]^{\mathsf{comp}}$ in each case.

Identifiers. Consider the set of plays of the copy-cat strategy for the projection $\pi_x : [\![\Gamma]\!] \to [\![\theta]\!]$:

$$\Sigma_{\pi_x} = \{\mathsf{s} \in \mathsf{P}_{[\![\theta]\!]_1 \multimap [\![\theta]\!]_2} \mid \text{ if } \mathsf{s}' \sqsubseteq \mathsf{s},\ \text{even}(\text{length}(\mathsf{s}')),\ \text{then } \mathsf{s}' \upharpoonright [\![\theta]\!]_1 = \mathsf{s}' \upharpoonright [\![\theta]\!]_2\},$$

using definitions 3.10 and 3.11 (pages 46–46) and the definition of projection in the CCC, Theorem 3.1. We show that $\Sigma_{\pi_x}^{\mathsf{comp}}$ is a regular language equal to

$$\mathcal{K}_\theta^x = \mathcal{R}[\![\theta]\!]\,[\mathsf{n_o}/\mathsf{n_o}\cdot\mathsf{n_o}^{\langle x\rangle}][\mathsf{n_p}/\mathsf{n_p}^{\langle x\rangle}\cdot\mathsf{n_p}], \tag{4.2}$$

for all moves $\mathsf{n_o}, \mathsf{n_p} \in \mathsf{M}_\theta$, such that $\lambda^{\mathsf{OP}}(\mathsf{n_o}) = \mathsf{O}$, $\lambda^{\mathsf{OP}}(\mathsf{n_p}) = \mathsf{P}$.

For ground types this follows directly from the definition, by inspection, as the sets involved are finite.

For first-order types, it is also straightforward; the only additional property that needs to be checked is that the justification pointers can be reconstructed from the set of complete plays. Indeed, this follows from the game definition of copy-cat [McC97, page 30], which stipulates that the justifier for a P move is the copy of the justifier of the corresponding O move. The other justification pointers are inherited from the ground-type game, where they can be immediately reconstructed in any complete play, from definitions.

The numeric tags are introduced in Equation 4.1 on the preceding page and the identifier tags are introduced in Equation 4.2; composed together they form ρ_1.

Abstraction. The definition of abstraction, 3.18 on page 52, indicates that

$$[\![\Gamma \vdash \lambda x{:}\sigma.P : \sigma \to \theta]\!]^{\mathsf{comp}} = \Lambda\left([\![\Gamma, x : \sigma \vdash P : \theta]\!]^{\mathsf{comp}}\right) \cong [\![\Gamma, x : \sigma \vdash P : \theta]\!]^{\mathsf{comp}},$$

since the function $\Lambda(-)$ is an isomorphism. By induction hypothesis,

$$[\![\Gamma, x : \sigma \vdash P : \theta]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![\Gamma, x : \sigma \vdash P : \theta]\!].$$

Also,

$$[\![\Gamma, x : \sigma \vdash P : \theta]\!] \cong [\![\Gamma, x : \sigma \vdash P : \theta]\!]\uparrow[\alpha^{\langle x\rangle}/\alpha^{\langle 1\rangle}] = [\![\Gamma \vdash \lambda x{:}\sigma.P : \sigma \to \theta]\!],$$

for all $\alpha \in [\![\sigma]\!]$; this is because the increment $(-)\uparrow$ is an isomorphism and the substitution $[\alpha^{\langle x\rangle}/\alpha^{\langle 1\rangle}]$ also an isomorphism. The composition of these two isomorphisms can be easily seen to be ρ_1^{-1}.

Application. We need to prove that for any $\Gamma \vdash P : \sigma \to \theta, M : \sigma$,

$$[\![PM : \theta]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} \left([\![\Gamma \vdash P : \sigma \to \theta]\!]\,[\mathsf{q}^{\langle 1\rangle}\cdot\mathsf{a}^{\langle 1\rangle}/(\!|M : \sigma|\!)_{\alpha'}]\right)\!\downarrow, \quad \text{for all } \mathsf{q} \in \mathsf{Q}\,[\![\sigma]\!],\ \mathsf{a} \in \mathsf{A_q}\,[\![\sigma]\!].$$

The game semantic interpretation of application of $P : \sigma \to \theta$ to $M : \sigma$ is:

$$[\![\Gamma]\!] \xrightarrow{\langle [\![P]\!], [\![M]\!]\rangle} [\![\sigma]\!] \Rightarrow [\![\theta]\!] \times [\![\sigma]\!] \xrightarrow{\text{ev}} [\![\theta]\!]$$

$$
\begin{array}{c|ccccccc|c|c}
\llbracket \Gamma \rrbracket & \xrightarrow{\langle \llbracket P \rrbracket, \llbracket M \rrbracket \rangle} & \llbracket \sigma \rrbracket^{\langle 1 \rangle} & \Rightarrow & \llbracket \theta \rrbracket^{\langle 1 \rangle} & \times & \llbracket \sigma \rrbracket^{\langle 2 \rangle} & \xrightarrow{\text{ev}} & \llbracket \theta \rrbracket^{\langle 2 \rangle} & \\
 & & & & & & & & \mathsf{q}_\sigma^{\langle 2 \rangle} & \mathsf{O} \\
 & & & & \mathsf{q}_\theta^{\langle 1 \rangle} & & & & & \mathsf{P} \\
\text{like } \llbracket P \rrbracket & & & & & & & & & \vdots \\
 & & \mathsf{q}_\sigma^{\langle 1 \rangle} & & & & & & & \mathsf{O} \\
 & & & & & & \mathsf{q}_\sigma^{\langle 2 \rangle} & & & \mathsf{P} \\
\text{like } \llbracket M \rrbracket & & & & & & & & & \vdots \\
 & & & & & & \mathsf{a}_\sigma^{\langle 2 \rangle} & & & \mathsf{O} \\
 & & \mathsf{a}_\sigma^{\langle 1 \rangle} & & & & & & & \mathsf{P} \\
\llbracket P \rrbracket \text{ resumes} & & & & & & & & & \\
\vdots & & \vdots & & & & & & & \vdots \\
 & & & & \mathsf{a}_\theta^{\langle 1 \rangle} & & & & \mathsf{a}_\sigma^{\langle 2 \rangle} & \mathsf{P}
\end{array}
$$

Figure 4.1: Plays of function application

where *ev* is the *evaluation* strategy. In general, this strategy does not have a regular set of complete plays, so the proof will be made directly at the level of the game semantics, by analyzing all the possible plays (Figure 4.1).

The opening question ($\mathsf{q}_\theta^{\langle 2 \rangle}$) always occurs in $\llbracket \theta \rrbracket^{\langle 2 \rangle}$, then is copied to $\llbracket \theta \rrbracket^{\langle 1 \rangle}$ by ev (as $\mathsf{q}_\theta^{\langle 1 \rangle}$). Subsequently, the strategy for $\llbracket P \rrbracket$ takes control of the play and it holds control of the play until a move ($\mathsf{q}_\sigma^{\langle 1 \rangle}$) occurs in $\llbracket \sigma \rrbracket^{\langle 1 \rangle}$. This move transfers control back to ev, which copies it to $\llbracket \sigma \rrbracket^{\langle 2 \rangle}$ (as $\mathsf{q}_\sigma^{\langle 1 \rangle}$). Subsequent play is then controlled by $\llbracket M \rrbracket$.

This is where the restriction to first-order for P (ground type for M) ensures that application is correctly represented by regular-language homomorphism (substitution). Two observations are essential:

1. a next move to $\llbracket \theta \rrbracket^{\langle 1 \rangle}$ is not possible because the play is governed by $\llbracket M \rrbracket$, which cannot use that type component;
2. since σ is a ground type, $\llbracket M \rrbracket$ must complete its play after moves in $\llbracket \Gamma \rrbracket$ only. A higher order strategy would be able to ask a question in $\llbracket \sigma \rrbracket^{\langle 2 \rangle}$, which ev would copy back to $\llbracket \sigma \rrbracket^{\langle 1 \rangle}$, and give control back to $\llbracket P \rrbracket$. This would cause a "nesting" of plays, and hence non-regularity, rather than the simple interleaving of first-order application.

Once $\llbracket M \rrbracket$ completes, the answer $\mathsf{a}_\sigma^{\langle 2 \rangle}$ is copied by ev from $\llbracket \sigma \rrbracket^{\langle 2 \rangle}$ to $\llbracket \sigma \rrbracket^{\langle 1 \rangle}$ and control switches back to $[\![P]\!]$. Because the justification pointer from $\mathsf{a}_\sigma^{\langle 1\rangle}$ to $\mathsf{q}_\sigma^{\langle 1\rangle}$ hides all play of $[\![M]\!]$ from $[\![P]\!]$, the latter simply resumes play from where it left off.

Finally, once $[\![P]\!]$ produces an answer $\mathsf{a}_\theta^{\langle 1\rangle}$, it is relayed by ev to $[\![\theta]\!]^{\langle 2\rangle}$, closing the entire play.

We can see how the moves from the two σ components function as switches between the strategies of $[\![P]\!]$ and $[\![M]\!]$, inserting complete plays of $[\![M]\!]$ bracketed by $\mathsf{q}_\sigma^{\langle 2\rangle}$ and $\mathsf{a}_\sigma^{\langle 1\rangle}$ in the plays of $[\![P]\!]$, whenever moves $\mathsf{q}_\sigma^{\langle 1\rangle} \cdot \mathsf{a}_\sigma^{\langle 1\rangle}$ occur.

Finally, all the moves from components $[\![\sigma]\!]^{\langle 1\rangle} \Rightarrow [\![\theta]\!]^{\langle 1\rangle} \times [\![\sigma]\!]^{\langle 2\rangle}$ are hidden, resulting in the same regular language as the one defined by substitution.

From the induction hypothesis, $[\![P]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![P]\!]$, $[\![M]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![M]\!]$. Decrement $(-)\!\downarrow$ is also an isomorphism, necessary in order to re-associate moves with the proper components after the elimination of σ.

Term-forming expressions. For sequential composition of commands, we need to prove that

$$[\![\Gamma \vdash C; C']\!] \overset{\rho_1}{\cong} [\![\Gamma \vdash \mathsf{seq}\, C\, C']\!]^{\mathsf{comp}}.$$

We give a regular-language interpretation to seq:

$$[\![\mathsf{seq} : \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm}]\!] = \mathit{run} \cdot \mathit{run}^{\langle 1\rangle} \cdot \mathit{done}^{\langle 1\rangle} \cdot \mathit{run}^{\langle 2\rangle} \cdot \mathit{done}^{\langle 2\rangle} \cdot \mathit{done}.$$

It can be immediately seen that:

$$[\![\mathsf{seq}\, C\, C']\!] = [\![C; C']\!] \quad \text{and} \quad [\![\mathsf{seq}]\!] \overset{\rho_1}{\cong} [\![\mathsf{seq}]\!]^{\mathsf{comp}}.$$

But by induction hypothesis,

$$[\![C]\!] \overset{\rho_1}{\cong} [\![C]\!]^{\mathsf{comp}} \quad \text{and} \quad [\![C']\!] \overset{\rho_1}{\cong} [\![C']\!]^{\mathsf{comp}}.$$

The other term-forming expressions have similar proofs.

Iteration. First-order IA does not have recursion, so we cannot use the definition from Equation 3.1 on page 48, which treats iteration as syntactic sugar for recursion.

However, assuming c_0 not free in C or B, we can calculate the fixed point of

$$W \overset{\mathrm{def}}{=} \Gamma \vdash \lambda c_0{:}\mathbf{comm}.\mathbf{if}\ B\ \mathbf{then}\ C; c_0\ \mathbf{else\ skip} : \mathbf{comm} \to \mathbf{comm}$$

directly using regular languages, from:

$$\begin{aligned} W_0 &= \mathbf{diverge} \\ W_{i+1} &= W(W_i), \end{aligned}$$

as:

$$\mathcal{W}_\infty = \bigcup_{i \in \mathbb{N}} \mathcal{W}_i,$$

where

$$\begin{aligned}
\mathcal{W}_0 &= \emptyset \\
run \cdot \mathcal{W}_{i+1} \cdot done &= [\![\Gamma \vdash \lambda c_0{:}\mathbf{comm}.\mathbf{if}\ B\ \mathbf{then}\ C; c_0\ \mathbf{else\ skip}]\!]\,[run^{\langle 1\rangle} \cdot done^{\langle 1\rangle}/\mathcal{W}_i] \\
&= run \cdot (\!|B|\!)_{tt} \cdot (\!|C|\!) \cdot run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdot done + run \cdot (\!|B|\!)_{ff} \cdot done.
\end{aligned}$$

It can be easily proved by induction on n that

$$\mathcal{W}_n = \sum_{i=0}^{n-1} run \cdot \underbrace{\big((\!|B|\!)_{tt} \cdot (\!|C|\!) \cdots (\!|B|\!)_{tt} \cdot (\!|C|\!)\big)}_{i \text{ times}} \cdot (\!|B|\!)_{ff} \cdot done,$$

from which it follows immediately that

$$[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = \mathcal{W}_\infty = run \cdot \big((\!|B|\!)_{tt} \cdot (\!|C|\!)\big)^* \cdot (\!|B|\!)_{ff} \cdot done.$$

But we also know from the induction hypothesis (part of the main proof) that

$$[\![B]\!] \overset{\rho_1}{\cong} [\![B]\!]^{\mathsf{comp}} \qquad \text{and} \qquad [\![C]\!] \overset{\rho_1}{\cong} [\![C]\!]^{\mathsf{comp}}.$$

Local variables. According to the game-semantic definition of `new` (3.20 on page 53), a complete play of the strategy interpreting it is:

$$\begin{array}{ccccc}
(\mathbf{var}\tau & \to & \sigma) & \to & \sigma \\
 & & & & \mathsf{q} \\
 & & \mathsf{q} & & \\
\mathsf{s} & & & & \\
 & & \mathsf{a} & & \\
 & & & & \mathsf{a}
\end{array}$$

where s is a sequence of moves in which all occurrences of *read* and *write* globally satisfy the constraints made formal by the stability regular expression $\gamma_{\mathbf{var}\tau}$. Using the game-semantic definition of composition and an analysis of possible moves similar to the one we did for application it follows that

$$[\![\mathtt{new}_\tau(\lambda x{:}\mathbf{var}\tau.M)]\!]^{\mathsf{comp}} \overset{\rho_1}{\cong} [\![\mathbf{new}\tau\ x\ \mathbf{in}\ M]\!].$$

End Of Proof.

4.4 Semantics of full first-order IA

We now consider the semantics of the binding construct **let**. Without this construct the language is not syntactically complete, as there is no way to bind function definitions to function-denoting identifiers. But with **let**, the language fragment is actually a stand-alone programming language, and it will make sense to talk about the full abstraction property.

The syntax of **let**, given already in Section 2.1 (on page 13), is:

TYPING RULE

$$\frac{\Gamma \vdash P : \theta \qquad \Gamma, x : \theta \vdash P' : \theta'}{\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ P\ \mathbf{in}\ P' : \theta'}$$

At ground types, this definition is redundant because it can be replaced by abstraction and application:

$$\mathbf{let}\ x\ \mathbf{be}\ P\ \mathbf{in}\ P' = (\lambda x{:}\sigma.P')P.$$

This redundancy does not create any technical difficulties.

Semantically, the most straightforward way to handle binding is in the standard way, by adding an *environment* u as a parameter to the semantic valuation function. The environment is a function mapping the free identifiers of the term to regular languages:

Definition 4.19 (Binding)

$$[\![\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ P\ \mathbf{in}\ P' : \theta]\!]\, u = [\![\Gamma, x : \theta' \vdash P' : \theta]\!]\, \big(u \mid x \mapsto [\![\Gamma \vdash P' : \theta']\!]\, u\big).$$

The interpretation of identifiers will be different in the presence of the environment.

Definition 4.20 (Identifiers)

$$[\![\Gamma \vdash x : \theta]\!]\, u = u(x).$$

All the other semantic definitions of the previous section stay the same, except that all functions $[\![-]\!]$, $(\!|-|\!)$ now take the additional parameter u.

The following property is technically important:

Lemma 4.3 (Term substitution)

$$[\![\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ P\ \mathbf{in}\ P' : \theta]\!]\, u = [\![\Gamma \vdash P'[x/P]]\!]\, u.$$

PROOF:
(by structural induction on the syntax of P')

Basis. If $P' = \mathbf{k}$ is an IA constant then $\mathbf{k} = \mathbf{k}[x/P]$ and the property follows trivially.

If $P' = x'$ is an identifier then we have two cases:

either $(x = x')$**,** in which case $x[x/P] = P$ and

$$\begin{aligned}[\![\Gamma \vdash \textbf{let } x \textbf{ be } P \textbf{ in } x : \theta]\!]\, u &= [\![\Gamma, x : \theta \vdash x : \theta]\!] \big(u \mid x \mapsto [\![\Gamma \vdash P : \theta]\!]\, u\big) \\ &= \big(u \mid x \mapsto [\![\Gamma \vdash P : \theta]\!]\, u\big)(x) \\ &= [\![\Gamma \vdash P : \theta]\!]\, u = [\![\Gamma \vdash x[x/P] : \theta]\!]\, u\end{aligned}$$

or $(x \neq x')$**,** in which case $x'[x/P] = x'$ and

$$\begin{aligned}\big[\![\Gamma \vdash \textbf{let } x \textbf{ be } P \textbf{ in } x' : \theta\big]\!]\, u &= \big[\![\Gamma, x' : \theta \vdash x' : \theta\big]\!] \big(u \mid x \mapsto \big[\![\Gamma \vdash P : \theta'\big]\!]\, u\big) \\ &= \big[\![\Gamma, x' : \theta \vdash x' : \theta\big]\!]\, u = \big[\![\Gamma \vdash x'[x/P] : \theta\big]\!]\, u.\end{aligned}$$

Composite terms. For non-binding terms of IA (i.e. all except *let* and abstraction) the proofs are similar. We present sequential composition in detail:

$$\begin{aligned}\big[\![\textbf{let } x \textbf{ be } P \textbf{ in } C; C'\big]\!]\, u &= \big[\![C; C'\big]\!]\, \big(u \mid x \mapsto [\![P]\!]\, u\big) \\ &= \mathit{run} \cdot (\!|C|\!) \big(u \mid x \mapsto [\![P]\!]\, u\big) \cdot (\!|C'|\!) \big(u \mid x \mapsto [\![P]\!]\, u\big) \cdot \mathit{done} \\ &= \mathit{run} \cdot (\!|C[x/P]|\!) u \cdot (\!|C'[x/P]|\!) u \cdot \mathit{done} \quad \text{(by induction hyp.)} \\ &= \mathit{run} \cdot (\!|C[x/P]; C'[x/P]|\!) u \cdot \mathit{done} \\ &= \mathit{run} \cdot (\!|(C; C')[x/P]|\!) u \cdot \mathit{done} \\ &= \big[\![(C; C')[x/P]\big]\!]\, u.\end{aligned}$$

For the two binding combinators we have two cases:

- $x \neq x'$: similar to the one above, for non-binding combinators.
- $x = x'$: then $[\![\textbf{let } x \textbf{ be } P \textbf{ in let } x \textbf{ be } P' \textbf{ in } P'']\!]\, u = [\![\textbf{let } x \textbf{ be } P' \textbf{ in } P'']\!]\, u$.

END OF PROOF.

It is intuitively clear that the definition of **let** is orthogonal to the purely regular-language semantics of the previous section. This intuition is formalized by the following property:

Lemma 4.4 (Reduction) *For any term $\Gamma \vdash P : \theta$ of IA there exists a **let**-free term $\Gamma \vdash P_0 : \theta$, such that*

$$[\![\Gamma \vdash P : \theta]\!]\, u_\Gamma = [\![\Gamma \vdash P_0 : \theta]\!]\,,$$

where u_Γ is an environment mapping all identifiers of Γ to copy-cat regular expressions:

$$\mathit{dom}(u_\Gamma) = \mathit{dom}(\Gamma), \quad u_\Gamma(x) = \mathcal{K}^x_{\Gamma(x)}.$$

Moreover, $\Gamma \vdash P \equiv_\theta P_0$.

The overloaded notations $[\![-]\!]\,u$ and $[\![-]\!]$ should not create confusion: the former is the environment-based semantics of this section, the latter is the purely regular-language semantics of the previous section.
PROOF: (by structural induction on the syntax of P)

Basic terms. If P is a language constant then we can choose P_0 to be P and the property is trivially correct, as the semantic definitions are identical.

If $P = x$ is an identifier then we can choose P_0 to be x as well. The semantic definitions are identical as well, $[\![x]\!] = [\![x]\!]\,u_\Gamma = \mathcal{K}^x_\theta$.

Composite terms. For all term-forming combinators, other than **let** itself, the proofs are similar. For example, if $P = P'P''$ then, by induction hypothesis there are **let**-free terms P'_0, P''_0 such that $[\![P']\!]\,u_\Gamma = [\![P'_0]\!]$ and $[\![P'']\!]\,u_\Gamma = [\![P''_0]\!]$, from which it follows immediately that $P_0 = P'_0 P''_0$.

Let. For $P = \textbf{let}\ x\ \textbf{be}\ P'\ \textbf{in}\ P''$ we use induction on the number of occurrences n of **let** in P.

Basis. $(n = 1)$ This means that P' and P'' are **let**-free, and so is $P_0 \stackrel{\text{def}}{=} P''[x/P']$. Using the Substitution Lemma,

$$\begin{aligned}[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ P'\ \textbf{in}\ P'' : \theta]\!]\,u_\Gamma &= [\![\Gamma, x : \theta' \vdash P''[x/P'] : \theta]\!]\,u_\Gamma \\ &= [\![\Gamma, x : \theta' \vdash P''[x/P'] : \theta]\!]\,.\end{aligned}$$

Since $P''[x/P']$ is **let**-free we can change to the environment-free semantics.

Inductive step. From the Substitution Lemma:

$$\begin{aligned}[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ P'\ \textbf{in}\ P'' : \theta]\!]\,u_\Gamma &= [\![\Gamma, x : \theta' \vdash P''[x/P'] : \theta]\!]\,u_\Gamma \\ &= [\![\Gamma, x : \theta' \vdash P_0 : \theta]\!]\,,\end{aligned}$$

by induction hypothesis. We can apply the induction hypothesis because $P''[x/P']$ has one less occurrence of **let** that P.

P_0 is equivalent to P, $\Gamma \vdash P \equiv_\theta P_0$ is proved by the same induction method, using the operational semantics. END OF PROOF.

Before stating the full abstraction result there is one other minor technical issue. Full abstraction of IA is defined with respect to the full language, which means that for all inequivalent terms there exists a discriminating context which can tell them apart. But there is no immediate guarantee that if we restrict our attention to the first-order fragment then the discriminating contexts also belong to the first-order fragment. However, this follows from the Operational Extensionality Theorem (Theorem 2.1 on page 17).

Proposition 4.2 *For all terms of first-order IA such that $\Gamma \vdash P_1 \not\equiv_\theta P_2$, there is a context $C[-]$ such that $\vdash C[P_1] \not\equiv_\sigma C[P_2]$ with $\vdash C[P_i] : \sigma$ a closed term of first-order IA.*

PROOF:
From the Operational Extensionality Theorem,

$$\Gamma \vdash P_1 \not\equiv_\theta P_2 \quad \text{iff} \quad \Gamma \vdash P_1 \not\cong_\theta P_2,$$

which, using extensional equivalence of open terms (definition 2.5 on page 17) is the case if and only if

$$\Omega' \vdash P_1[x_1, x_2, \ldots, x_n / P'_1, P'_2, \ldots, P'_n] \cong_\theta P_2[x_1, x_2, \ldots, x_n / P'_1, P'_2, \ldots, P'_n]$$

for some $\Omega' \vdash P'_j : \theta_j$ where Ω' is a subset of Γ.

From the operational semantics of **let**, this means that discriminating contexts can be chosen to have the form:

$$\textbf{let } x_1 \textbf{ be } P'_1 \textbf{ in } \cdots \textbf{let } x_n \textbf{ be } P'_n \textbf{ in } [-].$$

END OF PROOF.

We can now state the two principal properties of the regular-language semantics of first-order IA.

Theorem 4.1 (Full abstraction)

$$\Gamma \vdash P_1 \equiv_\theta P_2 \quad \textit{if and only if} \quad [\![\Gamma \vdash P_1 : \theta]\!]\, u_\Gamma = [\![\Gamma \vdash P_2 : \theta]\!]\, u_\Gamma$$

PROOF: The reduction Lemma (4.4 on page 75) reduces the equivalence of arbitrary terms of first-order IA to equivalence of **let**-free terms.

$$\Gamma \vdash P_1 \equiv_\theta P_2 \text{ if and only if } \Gamma \vdash P'_1 \equiv_\theta P'_2,$$

where P'_i are **let**-free and $\Gamma \vdash P'_i \equiv_\theta P_i$ and $[\![\Gamma \vdash P_i : \theta]\!]\, u_\Gamma = [\![\Gamma \vdash P'_i]\!]$.

The term representation Lemma (4.2 on page 69) shows that the regular-language semantics of **let**-free terms is isomorphic to the Abramsky-McCusker game semantics, i.e.:

$$\Gamma \vdash P'_1 \equiv_\theta P'_2 \quad \text{if and only if} \quad [\![\Gamma \vdash P'_1 : \theta]\!] = [\![\Gamma \vdash P'_2 : \theta]\!]\,.$$

In other words, since the game semantics is fully abstract (Theorem 3.2 on page 54), the regular language semantics is also fully abstract. END OF PROOF.

An immediate corollary of the full abstraction theorem is:

Corollary 4.1 (Decidability) *Equivalence of first-order finitary IA terms is decidable.*

PROOF: The fully abstract regular-language semantics interprets terms as regular languages, for which equivalence is decidable. END OF PROOF.

The language described here supports some straightforward extensions such as arrays or low-level data pointers (*à la* C) which can be introduced as syntactic sugar. A simple *semantic* extension, which, for this first-order fragment is orthogonal to the rest of the semantic model, is *bounded nondeterminism* [HM99]. The only addition to the language is a nondeterministic expression:

$$[\![\mathbf{random}\tau : \mathbf{exp}\tau]\!]\, u = \sum_{\alpha \in \mathcal{A}[\![\tau]\!]} q \cdot \alpha.$$

A more substantial modification of the language is the use of call-by-value instead of call-by-name. The regular-language semantics of this language is presented in [Ghi01a]. Other extensions, such as control or parallelism will require a much more substantial revision of the semantic framework.

4.4.1 Sample type analysis

Before concluding this chapter we will have a look at how we can use the regular-language semantics to gain some insight in the semantic structure of IA types.

The regular languages $\mathcal{R}[\![\theta]\!]$ (Lemma 4.1 on page 69) provide some additional insight into the structure of IA types. Closed terms of IA are interpreted by deterministic regular languages generated by $\mathcal{R}[\![\theta]\!]$.

Remark 4.1

$$\{[\![\ \vdash P : \theta]\!] \mid \ \vdash P : \theta\} = \mathcal{R}[\![\theta]\!]^{det},$$

where the set of deterministic languages generated by R is defined as

$$R^{det} \stackrel{def}{=} \{R' \mid \forall \omega \in R', \forall \omega' \sqsubseteq \omega, even(length(\omega'))\ \textit{implies}\ \forall \omega' \cdot \alpha, \omega' \cdot \alpha' \sqsubseteq \omega, \alpha = \alpha'\}.$$

PROOF: Directly from the fact that strategies interpreting IA are deterministic, in the same sense as in [AM96], and the fact that the regular-language model is fully abstract. The definition of R^{det} is the same as the definition of deterministic strategies, but relativized to the context of regular languages. END OF PROOF.

It is easy to show that:

Proposition 4.3 *If R is a regular language then R^{det} is a regular language.*

The analysis here replicates the one O'Hearn and Reynolds make, using a translation into polymorphic linear lambda calculus [OR00].

Example 4.12 ([OR00, Ex. 4, p. 28])

$$\mathbf{comm} \to \mathbf{comm} \cong \mathbb{N}_\perp.$$

PROOF:
The structure of the type **comm** → **comm** is, according to the remark above, the same as the structure of the deterministic regular languages generated by

$$\mathcal{R}[\![\mathbf{comm} \to \mathbf{comm}]\!] = run \cdot (run^{\langle 1 \rangle} \cdot done^{\langle 1 \rangle})^* \cdot done.$$

This is isomorphic to the set of lifted natural numbers:

$$\big(run \cdot (run^{\langle 1\rangle} \cdot done^{\langle 1\rangle})^* \cdot done\big)^{\mathrm{det}} \overset{\rho}{\cong} \mathbb{N}_\perp,$$

because

$$\begin{aligned}&\big(run \cdot (run^{\langle 1\rangle} \cdot done^{\langle 1\rangle})^* \cdot done\big)^{\mathrm{det}}\\ &\qquad = \{\emptyset\} \cup \big\{\{run \cdot \underbrace{run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdots run^{\langle 1\rangle} \cdot done^{\langle 1\rangle}}_{n \text{ times}} \cdot done\} \mid n \in \mathbb{N}\big\},\end{aligned}$$

therefore the isomorphism is

$$\begin{aligned}\rho(\perp) &= \emptyset = [\![\lambda c{:}\mathbf{comm}.\mathbf{diverge}]\!]\\ \rho(0) &= run \cdot done = [\![\lambda c{:}\mathbf{comm}.\mathbf{skip}]\!]\\ \rho(n) &= run \cdot \underbrace{run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} \cdots run^{\langle 1\rangle} \cdot done^{\langle 1\rangle}}_{n \text{ times}} \cdot done\\ &= [\![\lambda c{:}\mathbf{comm}.\underbrace{c; \cdots ; c}_{n \text{ times}}]\!].\end{aligned}$$

This is consistent with the Reynolds-O'Hearn analysis. END OF PROOF.

Example 4.13 ([OR00, Ex. 6,p. 29]) $\mathbf{comm} \to \mathbf{comm} \to \mathbf{comm} \cong \mathit{list}\{1,2\}_\perp$.

PROOF:

$$\begin{aligned}\mathcal{R}[\![\mathbf{comm} \to \mathbf{comm} \to \mathbf{comm}]\!]^{\mathrm{det}} &= \big(run \cdot (run^{\langle 1\rangle} \cdot done^{\langle 1\rangle} + run^{\langle 2\rangle} \cdot done^{\langle 2\rangle})^* \cdot done\big)^{\mathrm{det}}\\ &= \big\{\emptyset, \{run \cdot done\}\big\} \cup\\ &\qquad \big\{\{run \cdot run^{\langle i_1\rangle} \cdot done^{\langle i_1\rangle} \cdots run^{\langle i_n\rangle} \cdot done^{\langle i_n\rangle} \cdot done\}\\ &\qquad \mid n \in \mathbb{N}, i_j \in \{1,2\}, 1 \le j \le n\big\}\end{aligned}$$

The isomorphism is:

$$\begin{aligned}\rho(\perp) &= \emptyset = [\![\lambda c_1{:}\mathbf{comm}.\lambda c_2{:}\mathbf{comm}.\mathbf{diverge}]\!]\\ \rho([\ ]) &= run \cdot done = [\![\lambda c_1{:}\mathbf{comm}.\lambda c_2{:}\mathbf{comm}.\mathbf{skip}]\!]\\ \rho([i_1, \dots, i_n]) &= run \cdot run^{\langle i_1\rangle} \cdot done^{\langle i_1\rangle} \cdots run^{\langle i_n\rangle} \cdot done^{\langle i_n\rangle} \cdot done\\ &= [\![\lambda c_1{:}\mathbf{comm}.\lambda c_2{:}\mathbf{comm}.c_{i_1}; \cdots ; c_{i_n}]\!], \quad i_j \in \{1,2\}.\end{aligned}$$

END OF PROOF.

Chapter 5

Specification and Verification

The apparent dependence of Tarski-type truth definitions on set theory is in my view one of the most disconcerting features of the current scene in logic and in the foundations of mathematics. Jaakko Hintikka

5.1 Background

Extending Hoare-logic to a language with procedures is a difficult problem. Many solutions have been put forth [Hoa71, dBdBZ80, GL80, Old84, THM83, Sie85], but the most general and usable is, arguably, the specification logic of Reynolds [Rey81b, Rey81c], which we have briefly described in Section 2.4. As already mentioned, what makes the specification and verification of procedural programs particularly difficult is the possibility of surreptitious interactions, called *interference*, between non-local objects. The existence of such interactions invalidates several important Hoare axioms: the so-called frame rule and the assignment axiom.

Reynolds's strategy is to control non-interference by imposing conditions amounting to the fact that non-local objects do not share store. Subject to these conditions, assignment validates a Hoare-like axiom (see page 30) and several frame-like axioms are proved. An important technical issue in Reynolds's logic is that programming-language expressions are side-effect free, which means they may not have any computational effects such as assignments, control or non-termination. Side-effect-free expressions are helpful in assertions, because they enjoy all standard mathematical and logical properties. Therefore, reasoning at the level of the assertions is static and does not require additional programming axioms.

Reynolds gives enough non-trivial examples to provide ample evidence that his logic is useful and usable [Rey81b]. Whether restricting expressions so that they are side-effect free is a major restriction is debatable. An expression-like function $f : \mathbf{expint} \to \mathbf{expint}$ with side effects can, most of the time, be treated as a pure procedure that communicates with its environment via a variable-typed parameter:

$$\mathit{proc} : \mathbf{expint} \to \mathbf{varint} \to \mathbf{comm} \stackrel{\text{def}}{=} \lambda x{:}\mathbf{expint}.\lambda v{:}\mathbf{varint}.v := f(x).$$

So eliminating side effects from functions is not a matter of expressivity, but rather one of style and convenience.

Suppose we need to write a function *fib*, so that $\mathit{fib}(n)$ returns the n^{th} Fibonacci number. This function can be implemented in two ways, "imperatively," using assignments and iteration, or "functionally," using recursion. The two implementations would be, from the point of view of a client, identical; however, if a language with side-effect-free expressions is to be used then the imperative implementation must have the type $\mathbf{expint} \to \mathbf{varint} \to \mathbf{comm}$, while the functional implementation could have either type $\mathbf{expint} \to \mathbf{varint} \to \mathbf{comm}$ (which would be rather silly) or $\mathbf{expint} \to \mathbf{expint}$ (which is more natural). Clearly, to have the low-level design decision of which implementation is chosen reflected in the interface of the program (i.e. in the type of *fib*) is not acceptable. From a methodological view, the only way interfaces would give the programmer the flexibility of possibly adopting imperative implementations is to consistently use procedures instead of functions, except for the most trivial situations.

This brings us to the second issue. Using a procedure instead of a function is possible, but quite tedious. Using functions, we can formulate an assertion such as:

$$\mathit{fib}(n) + \mathit{fib}(n+1) = \mathit{fib}(n+2).$$

Using procedures, we must write something like the following specification:

$$\begin{aligned}&\mathbf{gv}(v) \wedge \mathit{fib} \# n \wedge \mathit{fib} \# y_0 \wedge\ \mathit{fib} \# y_1 \wedge \mathit{fib} \# y_2 \Rightarrow \\ &\{\mathbf{true}\}\ \mathit{fib}(n, v)\ \{!v = y_0\} \wedge \{\mathbf{true}\}\ \mathit{fib}(n+1, v)\ \{!v = y_1\} \wedge \{\mathbf{true}\}\ \mathit{fib}(n+2, v)\ \{!v = y_2\} \\ &\qquad \Rightarrow y_0 + y_1 = y_2,\end{aligned}$$

which is cumbersome. Clearly, because procedures cannot be used in assertions they are in general more awkward to use than functions.

There is another solution which could be considered: allowing imperative implementations of a function in which the use of assignment is restricted to its local variables. These programming structures, called *block expressions*, have been studied ([Ten91, Sections 7.6 and 9.7] and [TT91]) but the typing, semantic and reasoning issues they raise are quite complex and no programming language known to the author attempts to implement them.

On the other hand, functions with side effects are ubiquitous in real-life programming languages, because such functions are convenient. Even when a function is *understood* to behave like a mathematical function it is still quite convenient for the programmer to be able to use operations with side effects in the implementation of the function. Printing intermediate output as a debugging trace, for example, is a simple yet useful such computational side effect. Signaling special circumstances such as an illegal combination of arguments by setting an error flag, as opposed to raising an exception, is another simple, low-level programming idiom which, in the absence of functions with side effects, would require the programmer to abandon the convenience of functions and switch to the more cumbersome procedures.

To summarize, Reynolds's specification logic is the most general and usable logic for procedural languages but it is not possible to use it in the context of the variant of IA presented here, because it uses expressions with side effects.

The known semantic models of specification logic [Ten90, O'H90, OT93b] are abstract, not suitable for model-checking. However, this does not mean that specification logic is inherently unsuitable for model checking. A game-semantic model of IA (with passive expressions) has been found [AM99] as well as a game-semantic model of interference-controlled IA [McC, McC02].

Although specification logic is semantically not compatible with our programming language, it sets a high standard of generality and usefulness which we will attempt to meet. In addition, the semantic model will need to stay true to the spirit of game (and regular-language) semantics, providing immediate support for direct verification by model checking.

5.2 Stability

It is well known that IA variables may be "bad," i.e. assigning to the variable does not guarantee the variable will produce, upon dereferencing, the value that has just been assigned to it:

$$x : \mathbf{varint}, e : \mathbf{expint} \vdash x := e; \mathbf{if}\ !x = e\ \mathbf{then\ diverge\ else\ skip} \quad \not\equiv \quad \mathbf{diverge}.$$

A possible discriminating context would bind x to a phrase such as **if** $!z = e$ **then** y **else** z with y, z distinct global variables. Once side effects are allowed in expressions, the situation worsens still, with variables (and expressions) having such wide ranges of dynamic behaviour that most equivalences relying on arithmetical-logical properties, or on the very concept of *value*, no longer hold:

$$e : \mathbf{expint} \vdash \mathbf{if}\ e + e = 2 \times e\ \mathbf{then\ diverge\ else\ skip} \not\equiv \mathbf{diverge}$$
$$e : \mathbf{expint} \vdash \mathbf{if}\ e = e\ \mathbf{then\ diverge\ else\ skip} \not\equiv \mathbf{diverge}.$$

A possible discriminating context for both is $x := !x + 1; !x$, with x a local variable.

Both these equivalences hold in IA without side effects in expressions, and their failure in the presence of side effects made researchers such as Park and Reynolds express skepticism regarding the ability to reason about such programming languages.[1]

Clearly, such unrestricted behaviour makes it impossible not only to carry out any static, mathematical-logical, reasoning at the level of the assertion but also to specify any meaningful properties. The behaviour of expression-typed and variable-typed identifiers needs to be constrained. We do not want to completely eliminate the possibility of side

[1] See page 7 in the Introduction.

effects; we just want to restrict the overall behaviour of a non-local object so that we can speak of its *value* in a meaningful way.

For an expression, what we want to say is that it consistently yields the same value on repeated evaluations, in any context. For a variable, what we want to say is that unless an explicit assignment to it is carried out, it also consistently yields the same value after repeated evaluation; if an assignment is executed then the value is exactly the assigned value. This notion can be lifted to functions: if the arguments are "well behaved" then the value of the function is "well behaved,", i.e. determined by the values of the arguments.

For variables, we notice that the desired property is exactly the *stability* property introduced in Definition 4.17 on page 63. For expressions, the desired property can also be easily captured as a regular language:

Definition 5.1 (Expression stability)

$$\gamma^{x}_{\mathbf{exp}\tau} = \sum_{\alpha \in \mathcal{A}[\![\tau]\!]} (q^{\langle x \rangle} \cdot \alpha^{\langle x \rangle})^{*}.$$

We note that for commands the concept of stability is trivial; all commands produce the same (dummy) value.

Definition 5.2 (Command stability)

$$\gamma^{x}_{\mathbf{comm}} = \epsilon.$$

For stability at first-order types, we require the function to behave in a stable manner if the arguments are also stable.

Definition 5.3 (Expression-like function stability)

$$\gamma^{f}_{\overline{\sigma} \to \mathbf{exp}\tau} = \Big\{ \omega \in \left(\mathcal{K}^{f}_{\overline{\sigma} \to \mathbf{exp}\tau}\right)^{*} \;\Big|\; \alpha, \alpha' \in \mathcal{A}[\![\tau]\!]$$
$$\textit{for all substrings } q^{\langle f \rangle} \cdot \omega_1 \cdot \alpha_1^{\langle f \rangle} \cdot \omega' \cdot q^{\langle f \rangle} \cdot \omega_2 \cdot \alpha_2^{\langle f \rangle} \sqsubseteq \omega,$$
$$\textit{if} \quad \omega_1 \cdot \omega_2 \in \bigcap_{1 \le i \le k} \widetilde{\gamma^{fi}_{\sigma_i}} \quad \textit{then} \quad \alpha_1 = \alpha_2 \Big\},$$

where $\overline{\sigma} \to \mathbf{exp}\tau = \sigma_1 \to \cdots \to \sigma_k \to \mathbf{exp}\tau$.

Informally, this definition is interpreted as follows. The substrings $q^{\langle f \rangle} \cdot \omega_i \cdot \alpha_i^{\langle f \rangle}$ represent two function calls to f with ω_i the actions of the arguments. The condition of stability is that if the actions of the arguments in the two calls are stable then the actions of the results, i.e. the values returned by the function, are also stable. The result being an expression, this means $\alpha_1 = \alpha_2$.

Proposition 5.1 $\gamma^{f}_{\sigma_1 \to \cdots \to \sigma_k \to \mathbf{exp}\tau}$ *is a regular language.*

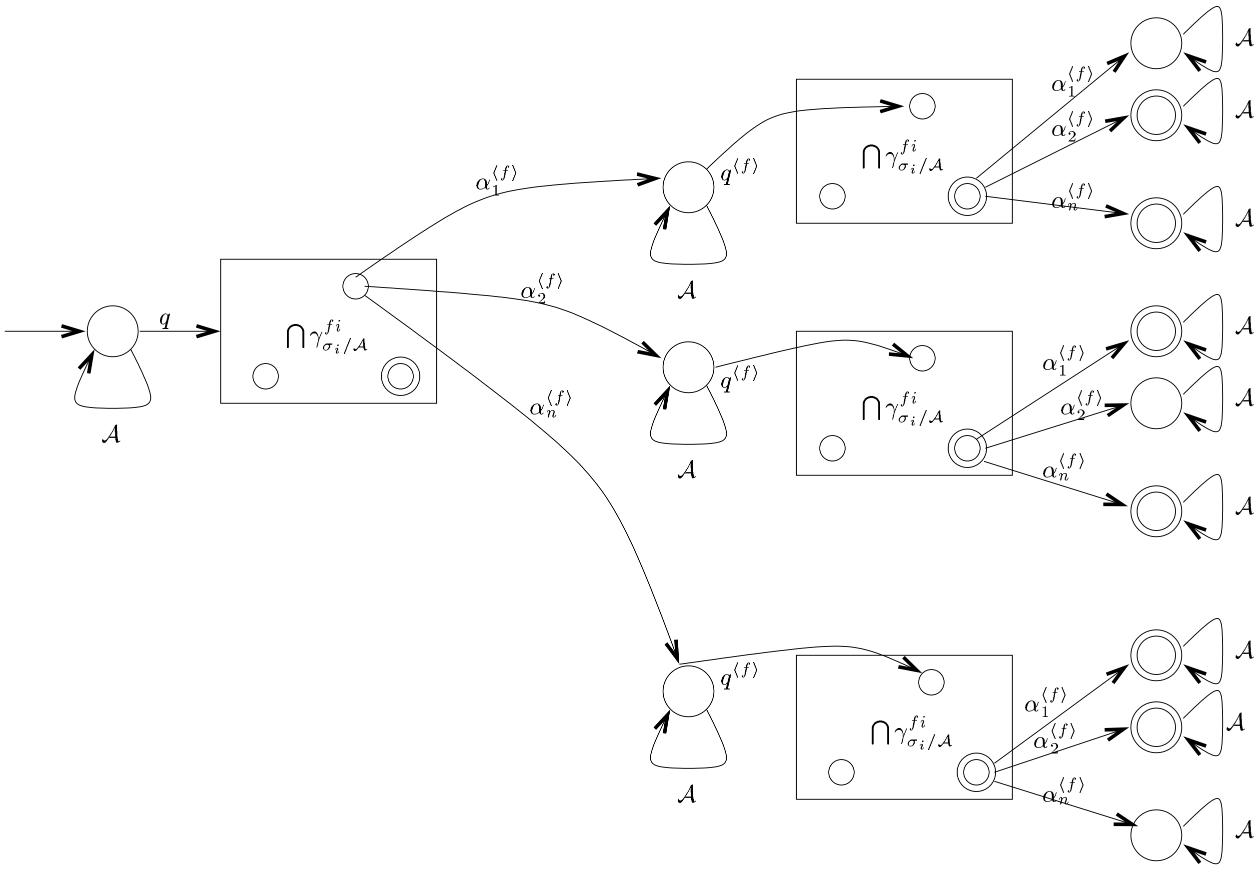


Figure 5.1: Finite state machine for $\mathcal{L}$ in Proposition 5.1.

PROOF:
We show that the language

$$\mathcal{L} = \left\{ \omega \in \left(\mathcal{K}^{f}_{\overline{\sigma}\to\mathbf{exp}\tau}\right)^* \;\middle|\; \omega \text{ has substrings } q^{\langle f\rangle}\cdot\omega_1\cdot\alpha_1^{\langle f\rangle}\cdot\omega'\cdot q^{\langle f\rangle}\cdot\omega_2\cdot\alpha_2^{\langle f\rangle} \text{ such that } \omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k} \widetilde{\gamma^{fi}_{\sigma_i}} \text{ and } \alpha_1 \neq \alpha_2 \right\}$$

is regular. $\gamma^{f}_{\overline{\sigma}\to\mathbf{exp}\tau}$ is the complement of $\mathcal{L}$ relative to $\left(\mathcal{K}^{f}_{\overline{\sigma}\to\mathbf{exp}\tau}\right)^*$, so it is regular.

We sketch the finite-state automaton that accepts $\mathcal{L}$ in Figure 5.1, for $\mathcal{A}[\![\tau]\!] = \{\alpha_1,\ldots,\alpha_n\}$ and $\mathcal{A} = \mathcal{A}[\![f : \overline{\sigma} \to \mathbf{exp}\tau]\!]$.

The boxes represent $n+1$ instances of the automaton accepting the regular language $\bigcap_{1\le i\le k} \widetilde{\gamma^{fi}_{\sigma_i}}$.

Whenever an $\alpha_k^{\langle f\rangle}$ is encountered there is a transition outside of the automaton, where the ω' part of the string is consumed.

When $q^{\langle f\rangle}$ is encountered there is a transition back into a copy of the automaton for $\bigcap_{1\le i\le k}\widetilde{\gamma_{\sigma_i}^{fi}}$, in the same state as the one which was left earlier.

The accepting state of this automaton makes a transition to a global accepting state if and only if $\alpha_1 \neq \alpha_2$. END OF PROOF.

We define stability for variable-returning functions similarly.

Definition 5.4 (Variable-like function stability)

$$\begin{aligned}
\gamma^f_{\overline{\sigma}\to\mathbf{var}\tau} = \Big\{\omega \in \left(\mathcal{K}^f_{\overline{\sigma}\to\mathbf{var}\tau}\right)^* \ \Big|\
& \textit{for all substrings } \textit{read}^{\langle f\rangle}\cdot\omega_1\cdot\alpha_1^{\langle f\rangle}\cdot\omega'\cdot\textit{read}^{\langle f\rangle}\cdot\omega_2\cdot\alpha_2^{\langle f\rangle}\sqsubseteq\omega,\\
& \quad \textit{if}\quad \omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k}\widetilde{\gamma_{\sigma_i}^{fi}} \ \textit{and}\\
& \quad \textit{for all } \textit{write}(\alpha)^{\langle f\rangle}\cdot\omega''\cdot ok^{\langle f\rangle} \textit{ in } \omega',\ \omega_1\cdot\omega''\cdot\omega_2 \notin \bigcap_{1\le i\le k}\widetilde{\gamma_{\sigma_i}^{fi}}\\
& \quad \textit{then } \alpha_1 = \alpha_2\\
\textit{and}\\
& \textit{for all substrings } \textit{write}(\alpha_1)^{\langle f\rangle}\cdot\omega_1\cdot ok^{\langle f\rangle}\cdot\omega'\cdot\textit{read}^{\langle f\rangle}\cdot\omega_2\cdot\alpha_2^{\langle f\rangle}\sqsubseteq\omega,\\
& \quad \textit{if}\quad \omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k}\widetilde{\gamma_{\sigma_i}^{fi}} \ \textit{and}\\
& \quad \textit{for all } \textit{write}(\alpha)^{\langle f\rangle}\cdot\omega''\cdot ok^{\langle f\rangle} \textit{ in } \omega',\ \omega_1\cdot\omega''\cdot\omega_2 \notin \bigcap_{1\le i\le k}\widetilde{\gamma_{\sigma_i}^{fi}},\quad \alpha\neq\alpha_2\\
& \quad \textit{then}\quad \alpha_1 = \alpha_2\Big\},
\end{aligned}$$

where $\overline{\sigma}\to\mathbf{var}\tau = \sigma_1\to\cdots\to\sigma_k\to\mathbf{var}\tau$.

The strings ω_1, ω_2, ω'' above correspond to actions of the parameters of the function. Informally, $\omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k}\widetilde{\gamma_{\sigma_i}^{fi}}$ means that the arguments behave in a stable way throughout the two function calls. The definition establishes the connection between the *read*s and the *write*s of a stable variable. The required condition for the function call to behave in a stable way is that the arguments behave in a stable manner and there are no intervening *write*s with the same arguments.

The definition of variable-like function stability ensures that, if $f : \mathbf{expint}\to\mathbf{varint}$ is stable, the phrase

$$f(1) := 1; f(2) := 2; !f(1)$$

produces the value 1 but the phrase

$$f(1) := 1; f(1) := 2; !f(1)$$

produces the value 2.

Proposition 5.2 $\gamma^{f}_{\sigma_1\to\cdots\to\sigma_k\to\mathbf{var}\tau}$ *is a regular language.*

PROOF: Similar to that for Proposition 5.1 on page 84.

Consider the following languages:

$$\mathcal{L}_1 = \Big\{\omega \in \left(\mathcal{K}^{f}_{\overline{\sigma}\to\mathbf{var}\tau}\right)^* \;\Big|\; \omega \text{ has a substring } \mathit{read}^{\langle f\rangle}\cdot\omega_1\cdot\alpha_1^{\langle f\rangle}\cdot\omega'\cdot\mathit{read}^{\langle f\rangle}\cdot\omega_2\cdot\alpha_2^{\langle f\rangle}$$
$$\text{such that } \omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k}\widetilde{\gamma^{fi}_{\sigma_i}} \text{ and } \mathit{write}(\alpha)^{\langle f\rangle},\ \mathit{ok}^{\langle f\rangle} \text{ do not occur in } \omega' \text{ and } \alpha_1\neq\alpha_2\Big\}$$

$$\mathcal{L}_2 = \Big\{\omega \in \left(\mathcal{K}^{f}_{\overline{\sigma}\to\mathbf{var}\tau}\right)^* \;\Big|\; \omega \text{ has a substring}$$
$$\mathit{read}^{\langle f\rangle}\cdot\omega_1\cdot\alpha_1^{\langle f\rangle}\cdot\omega'\cdot\mathit{write}(\alpha)^{\langle f\rangle}\cdot\omega''\cdot\mathit{ok}^{\langle f\rangle}\cdot\mathit{read}^{\langle f\rangle}\cdot\omega_2\cdot\alpha_2^{\langle f\rangle}$$
$$\text{such that } \omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k}\widetilde{\gamma^{fi}_{\sigma_i}} \text{ and } \omega_1\cdot\omega''\cdot\omega_2 \notin \bigcap_{1\le i\le k}\widetilde{\gamma^{fi}_{\sigma_i}} \text{ and } \alpha_1\neq\alpha_2\Big\}$$

$$\mathcal{L}_3 = \Big\{\omega \in \left(\mathcal{K}^{f}_{\overline{\sigma}\to\mathbf{var}\tau}\right)^* \;\Big|\; \omega \text{ has a substring } \mathit{write}(\alpha_1)^{\langle f\rangle}\cdot\omega_1\cdot\mathit{ok}^{\langle f\rangle}\cdot\omega'\cdot\mathit{read}^{\langle f\rangle}\cdot\omega_2\cdot\alpha_2^{\langle f\rangle}$$
$$\text{such that } \omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k}\widetilde{\gamma^{fi}_{\sigma_i}} \text{ and } \mathit{write}(\alpha)^{\langle f\rangle},\ \mathit{ok}^{\langle f\rangle} \text{ do not occur in } \omega' \text{ and } \alpha_1\neq\alpha_2\Big\}$$

$$\mathcal{L}_4 = \Big\{\omega \in \left(\mathcal{K}^{f}_{\overline{\sigma}\to\mathbf{var}\tau}\right)^* \;\Big|\; \omega \text{ has a substring}$$
$$\mathit{write}(\alpha_1)^{\langle f\rangle}\cdot\omega_1\cdot\mathit{ok}^{\langle f\rangle}\cdot\omega'\cdot\mathit{write}(\alpha)^{\langle f\rangle}\cdot\omega''\cdot\mathit{ok}^{\langle f\rangle}\cdot\mathit{read}^{\langle f\rangle}\cdot\omega_2\cdot\alpha_2^{\langle f\rangle}$$
$$\text{such that } \omega_1\cdot\omega_2 \in \bigcap_{1\le i\le k}\widetilde{\gamma^{fi}_{\sigma_i}} \text{ and } \omega_1\cdot\omega''\cdot\omega_2 \notin \bigcap_{1\le i\le k}\widetilde{\gamma^{fi}_{\sigma_i}} \text{ and } \alpha_1\neq\alpha_2\Big\}$$

Languages $\mathcal{L}_1, \mathcal{L}_2$ correspond to non-strict functions and $\mathcal{L}_3, \mathcal{L}_4$ to strict functions.

These languages are regular; the automata that accept them are constructed similarly to the one in Figure 5.1 on page 85.

Then, from the definition,

$$\gamma^{f}_{\sigma_1\to\cdots\to\sigma_k\to\mathbf{var}\tau} = \left(\mathcal{K}^{f}_{\overline{\sigma}\to\mathbf{var}\tau}\right)^* \setminus (\mathcal{L}_1+\mathcal{L}_2+\mathcal{L}_3+\mathcal{L}_4).$$

END OF PROOF.

Command-like function (procedure) stability is trivial; all procedures are stable.

Definition 5.5 (Procedure stability) $\gamma^{f}_{\overline{\sigma}\to\mathbf{comm}} = \epsilon$.

Meaningful assertions are formulated using identifiers that only denote stable objects. Although they might have side effects, stable objects have a behaviour consistent with that of persistent state, so they have *values* and they have the proper arithmetic-logical properties. In assertions, non-local stable objects are introduced using a generalized *stability quantifier*. We will discuss in Section 5.4 why generalized quantifiers are the natural way of introducing stable objects.

5.3 Assertions

We can now give the syntax of the language of assertions, as an extension to the syntax of IA (all other syntax rules of IA are kept the same):

TYPING RULES

$$\sigma ::= \cdots \mid \mathbf{assert},$$

$$\frac{\Gamma \vdash A_0 : \mathbf{assert} \qquad \Gamma \vdash A_1 : \mathbf{assert}}{\Gamma \vdash A_0 \ \mathbf{and}\ A_1 : \mathbf{assert}} \qquad \frac{\Gamma \vdash A_0 : \mathbf{assert} \qquad \Gamma \vdash A_1 : \mathbf{assert}}{\Gamma \vdash A_0 \ \mathbf{or}\ A_1 : \mathbf{assert}}$$

$$\frac{\Gamma \vdash B : \mathbf{expbool}}{\Gamma \vdash B : \mathbf{assert}} \qquad \frac{\Gamma \vdash A : \mathbf{assert}}{\Gamma \vdash \mathbf{not}\ A : \mathbf{assert}} \qquad \frac{\Gamma, x : \theta \vdash A : \mathbf{assert}}{\Gamma \vdash \nabla x : \theta.A : \mathbf{assert}}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash A : \mathbf{assert}}{\Gamma \vdash M; A : \mathbf{assert}}$$

Assertions are quite similar to boolean expressions, but are augmented with the new quantifier ∇ and they can be sequenced with any other first-order phrase. They may be used in abstraction, application and **let** constructs just like any other first-order IA phrase.

Implication and an existential counterpart of stability can be defined as syntactic sugar:

$$A \ \mathbf{implies}\ A' \stackrel{\text{def}}{=} \mathbf{not}\ A \ \mathbf{or}\ A',$$

respectively

$$\Delta x : \theta.A \stackrel{\text{def}}{=} \mathbf{not}\ (\nabla x : \theta.\mathbf{not}\ A).$$

Semantically, the definition of assertion connectives **and**, **or**, **not** is identical to that of the corresponding boolean operators (Definition 4.10 on page 59). The new terms are defined as follows:

Definition 5.6 (Semantics of assertions)

$$\begin{aligned}
[\![\Gamma \vdash B : \mathbf{assert}]\!]\, u &= [\![\Gamma \vdash B : \mathbf{expbool}]\!]\, u \\
(\!|\Gamma \vdash M; A : \mathbf{assert}|\!)_\alpha u &= (\!|\Gamma \vdash M : \sigma|\!) u \cdot (\!|\Gamma \vdash A : \mathbf{assert}|\!)_\alpha u, \quad \alpha \in \{tt, ff\} \\
[\![\Gamma \vdash \nabla x : \theta.A : \mathbf{assert}]\!]\, u &= \left([\![\Gamma, x : \theta \vdash A : \mathbf{assert}]\!]\, u \cap \widetilde{\gamma^x_\theta}\right) \restriction \mathcal{A}^{\overline{x}},
\end{aligned}$$

where the broadening context alphabet is $\mathcal{A}\,[\![\Gamma, x : \theta \vdash A : \mathbf{assert}]\!]$.

In the above, we use the notation

$$(\!|\Gamma \vdash M : \sigma|\!) u = \sum_{\mathsf{a} \in \mathsf{A}[\![\sigma]\!]} (\!|\Gamma \vdash M : \sigma|\!)_{\mathsf{a}} u,$$

to denote the set of all $(\!|M|\!)_\alpha u$, for all possible indices α.

The definition of stability is inspired by the definition of the block variable (Definition 4.18 on page 64): stability is a *global* constraint on the behaviour of x and scope is imposed through restriction to the alphabet of symbols not tagged by x. Stability, expressed as a regular language, has been defined for expressions (Definition 5.1 on page 84) and variables (Definition 4.17 on page 63).

Non-local objects introduced by stability quantifiers are well-behaved: they always have the arithmetic-logical properties necessary to support static reasoning, their behaviour is consistent with a notion of state and, as a result, they can participate in the formulation of meaningful assertions.

Looking back at the troublesome examples we have used in this section, it can be seen that they are no longer problematic in the presence of stability.

Example 5.1 (True assertions)

$$\begin{aligned}
&\nabla e : \mathbf{expint}.e = e \\
&\nabla e : \mathbf{expint}.e + e = 2 \times e \\
&\nabla x : \mathbf{var}\tau.\nabla e : \mathbf{exp}\tau.x := e; !x = e.
\end{aligned}$$

PROOF: Immediate from the definition of stability:

$$\begin{aligned}
[\![\nabla e : \mathbf{expint}.e = e]\!] &= ([\![e = e]\!] \cap \gamma^e_{\mathbf{expint}}) \restriction \mathcal{A}^{\overline{e}} \\
&= \Big((\sum_{\alpha \in \mathcal{A}[\![\tau]\!]} q \cdot q^{\langle e \rangle} \cdot \alpha^{\langle e \rangle} \cdot q^{\langle e \rangle} \cdot \alpha^{\langle e \rangle} \cdot tt \\
&\qquad + \sum_{\alpha_0 \neq \alpha_1 \in \mathcal{A}[\![\tau]\!]} q \cdot q^{\langle e \rangle} \cdot \alpha_0^{\langle e \rangle} \cdot q^{\langle e \rangle} \cdot \alpha_1^{\langle e \rangle} \cdot f\!f) \cap \gamma^e_{\mathbf{expint}} \Big) \restriction \mathcal{A}^{\overline{e}} \\
&= \big(\sum_{\alpha \in \mathcal{A}[\![\tau]\!]} q \cdot q^{\langle e \rangle} \cdot \alpha^{\langle e \rangle} \cdot q^{\langle e \rangle} \cdot \alpha^{\langle e \rangle} \cdot tt \big) \restriction \mathcal{A}^{\overline{e}} \\
&= q \cdot tt.
\end{aligned}$$

The other proofs are similar. END OF PROOF.

Stability is one of the key concepts of this approach, just as non-interference is the key concept introduced by Reynolds in his specification logic. The two are remarkably different, complementary in their approach to regulating the behaviour of non-local entities. Non-interference is a *local* condition which in some sense prevents the *causes* of unwanted behaviours. Stability is a *global* condition which excludes *effects* amounting to unwanted behaviour.

5.4 Specifications

Let us review the main criteria that the specification language must satisfy:

model checking: the semantic model of a specification needs to be compatible with algorithmic verification;

compositionality: the specification language must have good logical properties;

usefulness: a large enough class of programs can be specified and validated.

These three criteria are somewhat incompatible. We can reasonably anticipate that in order to satisfy the first one, significant restrictions must be imposed; but in order to satisfy the last two, we require generality and expressiveness.

Traditional semantic models of logic, including programming logics, interpret free identifiers using universal quantifiers in the meta-language, ranging either over all phrases of the same type as the identifiers or over all elements of its semantic domain (of course, the two approaches are equivalent if the semantic model is fully abstract). The order of the semantic quantifiers is one-higher than the order of the type of the free identifier. For example, a ground-type free identifier is interpreted using a first-order semantic quantifier; a first-order free identifier is interpreted using a second-order semantic quantifier. In regular-language terms, a first-order quantifier would correspond to quantifying over traces and a second-order quantifier to quantifying over languages of traces. This need to use metalanguages which are higher-order than the object languages has been sometimes referred to as "Tarski's curse," because it entails that no language can serve as its own metalanguage. The magic of game semantics is that it defeats this curse. Jaakko Hintikka [Hin96] shows how a first-order logic with generalized (branching) quantifiers can be given a complete (non-trivial) semantics using itself as a metalanguage.

Philosophical considerations aside, the practical importance, for model checking, is obvious, as logics with higher-order quantifiers are less well-behaved and less understood. So, for model-checking purposes it is preferable to use an interpretation of free identifiers that does not involve such second-order quantifiers at the semantic level. This is true especially in our case, because in the semantic model the first-order quantifiers are finitary while second-order ones are not.[2]

Notice that this "flat" interpretation of free identifiers in game semantics does not correspond exactly to a universal quantifier, as is the case in a traditional semantics. So it is intuitively incompatible with traditional specification idioms such as Reynolds's specification logic. In this traditional idiom, non-local objects, denoted by free or universally quantified identifiers, are specified using *implication*:

$$\forall x : \theta . S(x) \Rightarrow S'(x). \tag{5.1}$$

In the above, x is the identifier denoting the non-local object, $S(x)$ a specification in which x occurs freely and $S'(x)$ some other specification. S' is a specification about a

[2] This argument can be made much stronger by showing that the a specification language relying on universal quantifiers is, even at a very restricted fragment, undecidable. The author has been recently pursuing this research topic [Ghi02].

program fragment in which x occurs non-locally and S, used as an assumption for S', is in fact the specification of x. Informally, the semantics of the above is: *For all objects of type θ, if S is true of the object then S' is also true.* Even more informally, the universal quantifier introduces all possible objects of type θ; then the implication works like a filter, allowing only those objects that satisfy the assumption S. This makes S a *specification* for x in S'. This traditional style of specification is fundamentally reliant on the universal quantifier, a quantifier which we want to avoid because its semantics are higher order.

The solution we propose to this quandary is to employ a specification language which uses a semantically weakened universal quantifier and a collection of generalized quantifiers dealing with several properties we deem to be interesting. This specification language is more like a *modeling* language than a traditional logic and the inference rules for the quantifiers bear an intuitive resemblance to program refinement rules. Semantically, the ideas used here are akin to de Alfaro and Henzinger's *interface automata* [dAH01] for software model checking, based on the earlier ideas in [AG94, YS97, Jac00].

Whenever a non-local object is introduced, its behaviour is specified *explicitly* rather than *implicitly*, using a generalized quantifier. So equation 5.1 on the preceding page would be written as:

$$\mathbb{Q}_S x.S'(x),$$

where $\mathbb{Q}_S$ is a generalized quantifier binding x. The informal interpretation is: *for all objects such as S, S' is true of the object.* The difference between this and the interpretation of equation 5.1 is subtle but significant. Even more informally, only those objects satisfying S are generated in the first place. The fact that we do not use a universal quantifier greatly reduces the semantic space over which specification S' must be validated.

If the specification language contains a genuine universal quantifier then the quantifier $\mathbb{Q}_S$ is a *relativized quantifier*, i.e. syntactic sugar; but if the specification language does not contain such a quantifier then the quantifier $\mathbb{Q}_S$ is a generalized quantifier, a genuine extension of the language.

Generalized quantifiers were first introduced by Mostowski [Mos57] and developed further by Lindström [Lin66] to deal with concepts such as "for finitely many" or "for uncountably many," which cannot be handled by the ordinary quantifiers. Recent interest in generalized quantifier theory in computer science stems from the fact that they can effectively increase the expressiveness of logics without increasing their computational complexity; [Vää00] is a very informative introduction and survey of main results, especially concerning logics for PTIME.

There is an important caveat regarding generalized quantifiers. They represent *minimal* extensions of a logic in order to improve its expressiveness. So they are, in a sense, necessarily *ad-hoc.* As a methodology, we identify a concept we want to express, and which is not expressible in the current logic, and we model it using a generalized quantifier (the name *specialized quantifiers* would have been more fortunate but we stick to the standard terminology). The driving force is semantic, but because an important concern of ours is compositionality we will provide elimination rules for the quantifier as well.

This style of specification is more restrictive than the traditional one, but this very restrictiveness makes the approach so compatible with model checking. We will also see in Chapter 7 that such a restriction is not severe, and allows a large enough class of specifications which covers many interesting applications.

5.5 Specification syntax and semantics

The key difference between assertions and specifications is that assertions are "computational" (may have side effects, are dependent on state) whereas specifications are "logical" (have no side effects, are only true or false, are about the programs only).

Specifications are of two kinds: effects specifications and dynamic specifications, which describe the global behaviour of a phrase. The effects specifications are similar in form and purpose to the Hoare triples. Their syntax and typing is given by:

TYPING RULE

$$\frac{\Gamma \vdash A_1 : \mathbf{assert} \qquad \Gamma \vdash A_2 : \mathbf{assert} \qquad \Gamma \vdash M : \sigma}{\Gamma \vdash \{A_1\}\ M\ \{A_2\} : \mathbf{spec}}$$

Assertion A_1 is called the *pre-condition* and A_2 the *post-condition* of the specification.

We define the following abbreviation:

$$\{A\} \stackrel{\text{def}}{=} \{\mathbf{true}\}\ \mathbf{skip}\ \{A\},$$

where A is an assertion.

The dynamic specifications identify phrases which behave in a stable way:

TYPING RULE

$$\frac{\Gamma \vdash M : \theta}{\Gamma \vdash \boldsymbol{\nabla}_\theta(M) : \mathbf{spec}}$$

The other elements of the specification language are conjunction, implication, universal quantifier, the various generalized quantifiers as well as the constant specification **absurd**, which is always false.

TYPING RULES

$$\frac{\Gamma \vdash S_1 : \mathbf{spec} \qquad \Gamma \vdash S_2 : \mathbf{spec}}{\Gamma \vdash S_1 \wedge S_2 : \mathbf{spec}} \qquad \frac{\Gamma \vdash S_1 : \mathbf{spec} \qquad \Gamma \vdash S_2 : \mathbf{spec}}{\Gamma \vdash S_1 \Rightarrow S_2 : \mathbf{spec}}$$

$$\frac{\Gamma, x : \theta \vdash S : \mathbf{spec}}{\Gamma \vdash \forall x : \theta . S : \mathbf{spec}} \qquad \frac{}{\vdash \mathbf{absurd} : \mathbf{spec}}$$

Other connectives can be added as well but this logical fragment, also known as the *negative* fragment of predicate logic, is usually enough.

Much of the work in the specification language presented here is done by generalized quantifiers. The first one which we introduce is the stability quantifier:

TYPING RULE

$$\frac{\Gamma, x : \theta \vdash S : \mathbf{spec}}{\Gamma \vdash \nabla x : \theta . S : \mathbf{spec}}$$

Other generalized quantifiers will be introduced later.

Finally, equivalence is a specification.

TYPING RULE

$$\frac{\Gamma \vdash P : \theta \qquad \Gamma \vdash P' : \theta}{\Gamma \vdash P \equiv_\theta P' : \mathbf{spec}}$$

The semantics of specifications is given using a model consisting of two parameters. The first parameter is called the *environment*, and the second is called the *frame*. The environment captures the static, extensional properties of a non-local identifier while the frame its dynamic, intensional properties.

Definition 5.7 (Model, Standard model)

- *A model $\mathfrak{M}$ is a pair $\langle u, v\rangle$, where u, v are functions mapping identifiers to regular languages, such that $dom(u) = dom(v) = dom(\Gamma)$ and $u(x), v(x) \in \mathcal{R}_{\mathcal{A}[\![\Gamma]\!]}$ for all x. We call u the* environment *and v the* frame.
- *A standard model for environment Γ is denoted by $\mathfrak{M}_\Gamma = \langle u_\Gamma, v_\Gamma\rangle$ such that:*
  - *$u_\Gamma = (x_1 \mapsto \mathcal{K}_{\theta_1}^{x_1} \mid \cdots \mid x_k \mapsto \mathcal{K}_{\theta_k}^{x_k})$ for all $x_k : \theta_k \in \Gamma$ (same as in Lemma 4.4 on page 75);*

– $v_\Gamma(x) = \epsilon$, *for all* $x_k : \theta_k \in \Gamma$.

Definition 5.8 (Truth and validity)

- *We say that a specification* $\Gamma \vdash S : \mathbf{spec}$ *is* true in a model $\mathfrak{M}$, *and we denote it by* $\mathfrak{M} \models_\Gamma S$.
- *We say that a specification* $\Gamma \vdash S : \mathbf{spec}$ *is* valid *if it is true in the standard model,* $\mathfrak{M}_\Gamma$. *We denote a valid specification by* $\Gamma \models S$.

Whenever it does not cause confusion we may omit Γ from the above notations.

Remark 5.1 $\Gamma \models S$ *if and only if* $\Gamma_\emptyset \models \forall x_1 : \theta_1 \ldots \forall x_k : \theta_k.S$, *where* $\Gamma_\emptyset$ *is the empty environment,* $dom(\Gamma) = \{x_1, \ldots, x_k\}$ *and* $x_j : \theta_j \in \Gamma,\ 1 \leq j \leq k$.

The connectives and the quantifiers have the following interpretations:

Definition 5.9 (Specification semantics)

$$
\begin{array}{rll}
\mathfrak{M} \models \mathbf{absurd} & \textit{is always false} & \\
\mathfrak{M} \models S_1 \wedge S_2 & \textit{if and only if} & \mathfrak{M} \models S_1 \textit{ and } \mathfrak{M} \models S_2 \\
\mathfrak{M} \models S_1 \Rightarrow S_2 & \textit{if and only if} & \mathfrak{M} \models S_1 \textit{ implies } \mathfrak{M} \models S_2 \\
\langle u, v\rangle \models \forall x : \theta.S & \textit{if and only if} & \big\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \epsilon)\big\rangle \models S \\
\langle u, v\rangle \models \nabla x : \theta.S & \textit{if and only if} & \big\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \gamma^x_\theta)\big\rangle \models S.
\end{array}
$$

The interpretation of the connectives is classical.

The universal quantifier in a specification is interpreted using a copy-cat regular language, just like free variables in the game semantics of IA. The term "universal quantifier" as used in this logical framework is unconventional. Although the quantifier has binding and logical properties similar to the "standard" universal quantifier, it is nevertheless semantically weaker. In this framework, the quantifier does not range, semantically, over all terms of a certain type; the copy-cat strategy represents, in some informal sense, a "universal" object. The specification $\forall x.S$ says that S is true of the universal object, in contrast to the classical universal quantifier $\forall x.S$ which says that S is true of *any* object. The copy-cat strategies and empty dynamic constraint languages in the environment, respectively the frame, of the standard model $\mathfrak{M}_\Gamma$ are the representations of this universal object. From a model-checking point of view, the standard model is quite similar to the notion of "most general environment" used to provide closure to open systems [CGJ98].

This difference is subtle, but important. The $\forall$ quantifier is, semantically, also a generalized quantifier. As a philosophical aside, our $\forall$ quantifier is a *categorematic* quantifier while the classical $\forall$ quantifier is *syncategorematic*, i.e. the former has independent meaning and denotes (the universal object associated with the copy-cat strategy) while the latter has no independent meaning and does not denote (its meaning is given by substitution). The distinction between categorematic and syncategorematic words was made by the medieval logician William of Ockham in *Summa Logicae*, 1320. Until the introduction of generalized quantifiers it has been assumed that quantifiers must be syncategorematic (cf. standard semantics of predicate logic, such as [vD83]).

For the stability quantifier, the stability constraint is also imposed in the frame. The term "frame" is an allusion to the well known *frame problem* of AI, which has important consequences in program specification and verification [BMR93]. The set of dynamic constraints in v specify a frame because variables which are stable can only change if they are explicitly assigned to.

To interpret a Hoare-triple we need a game-semantic analogue of state, expressible in a regular language. In [AM96], Abramsky and McCusker use such a notion of "explicit state" in the proof of soundness, expressing state as a sequence of variable initializations. Our notion of state needs to be more general than that, because it must include state information about all the non-local entities of a term, not only variables. Our state needs to contain information about values of expressions and of functions as well.

Given any frame v, a computation consistent with it is any trace which is included in $\widetilde{v(x)}$, for all x in the domain of v. A state defined by a frame is any computation consistent with the frame containing at least one action tagged by x, for all x in the domain of v. The idea is that an action tagged by x is also an *observation* of the "state" of the non-local object denoted by identifier x. Such a computation has information about the state of every non-local object, so it has enough information to recover the state of the program.

Actually, this definition of state has *more* information than necessary, because it might contain redundant information. However, this raises no technical problems so we adopt this definition on grounds of its simplicity.

Definition 5.10 (State) *Given a frame v consistent with a type assignment Γ, we define the set of states $\eta(v)$ as*

$$\eta(v) = \{\omega \in \gamma(v) \mid \omega \restriction \alpha \neq \epsilon \ \text{ for all } x : \theta \in \Gamma \text{ and } \alpha \in \mathcal{A}\,[\![x : \theta]\!]\}.$$

where $\gamma(v) \stackrel{def}{=} \bigcap_{x \in dom(v)} \widetilde{v(x)}$, with broadening context $\mathcal{A} = \mathcal{A}\,[\![\Gamma]\!]$. (See Definition 4.5 on page 57.)

It is straightforward to show that:

Proposition 5.3 *For any frame v, the set of states $\eta(v)$ is regular.*

Definition 5.11 (Hoare triple, semantics)

$$\begin{aligned}\langle u, v\rangle \models_\Gamma \{A\}\ M\ \{A'\} \ &\text{if and only if for all } \omega \in \eta(v),\\ &(\omega \cdot [\![A]\!]\,u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt \text{ implies } (\omega \cdot [\![M; A']\!]\,u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt.\end{aligned}$$

This definition is similar to the standard definition of a Hoare triple, only that the representation of the *state* is the kind of *computation sequence* defined earlier. The constraints collected in the frame are applied simultaneously to the program phrase M and the assertions A and A'.

Informally, the definition is read as follows. An assertion is true of an arbitrary computation ω and in a frame v, if it evaluates only to tt immediately after ω and subject to the dynamic constraints imposed by v. A Hoare triple $\{A\}\ M\ \{A'\}$ is true in a frame v if for any computation ω for which the pre-condition A is true, the sequencing of M and post-condition A' is also true, both in frame v.

This is a partial correctness interpretation, because the empty set of traces, denoting nontermination, is included in the set of all strings terminating in tt.

A useful alternative formulation of the semantics of the Hoare triple is:

Proposition 5.4

$$\begin{aligned}&\langle u, v\rangle \models_\Gamma \{A\}\ M\ \{A'\} \quad \textit{if and only if}\\ &\qquad \textit{for all } \omega \in \eta(v), (\omega \cdot (\!|A|\!)_{\mathit{ff}} u) \cap \gamma(v) = \emptyset \quad \textit{implies} \quad (\omega \cdot (\!|M; A'|\!)_{\mathit{ff}} u) \cap \gamma(v) = \emptyset.\end{aligned}$$

Proof: Immediate from definition 5.11 on the preceding page. End Of Proof.

Finally, we interpret the stability specifications. The idea is the same as before: an expression is stable if it returns the same value, a variable is stable if it keeps the same value unless it is written to, case in which it returns the new value, and a function is stable if whenever its arguments behave in a stable manner it also behaves in a stable manner. The stability of commands and procedures is always trivially true.

In order to interpret stability, we introduce a notion of passive traces, intuitively akin to Reynolds's notion of *symmetric* non-interference [Rey78, OP+95]. A passive trace is not merely one that does not change the value of a variable relative to what the value was before the computation, but one that does not write to it at all; a computation that changes then restores a variable is not a passive trace. Moreover, a computation that writes to a variable the same value that the variable already holds is also not a passive trace.

Formally, a passive trace is a trace lacking active actions, where an active action is a *write* to one of the stable variables, given by the frame.

Definition 5.12 (Active actions) *For any term $\Gamma \vdash M : \theta$, the set of active actions in environment u and frame v is defined as*

$$\mathcal{A}_{M,u,v} = \{\mathit{write}(\alpha)^{\langle x\rangle}, ok^{\langle x\rangle} \mid v(x) = \gamma^x_{\mathbf{var}\tau} \textit{ or } v(x) = \gamma^x_{\vec{\sigma}\to\mathbf{var}\tau},\ x \in \mathit{Free}(M)\}.$$

Definition 5.13 (Passive traces) *For any term $\Gamma \vdash M : \theta$, the set of passive traces in environment u and frame v is defined as*

$$\mathcal{P}_{M,u,v} = \Big(\bigcup_{x\in \operatorname{dom} u} \lfloor u(x)\rfloor \setminus \mathcal{A}_{M,u,v}\Big)^*.$$

Definition 5.14 (Expression stability)

$$\begin{aligned}&\langle u, v\rangle \models_\Gamma \boldsymbol{\nabla}_{\mathbf{exp}\tau}(E) \textit{ if and only if}\\ &\qquad \big((\!|E|\!)_\alpha u \cdot \mathcal{P}_{E,u,v} \cdot (\!|E|\!)_{\alpha'} u\big) \cap \gamma(v) = \emptyset \textit{ for all } \alpha, \alpha' \in \mathcal{A}[\![\tau]\!], \alpha \neq \alpha'.\end{aligned}$$

Definition 5.15 (Variable stability)

$$\langle u, v\rangle \models_\Gamma \boldsymbol{\nabla}_{\mathbf{var}\tau}(V) \text{ if and only if } \big(\llparenthesis V \rrparenthesis_{\mathsf{r}\alpha} u \cdot \mathcal{P}_{V,u,v} \cdot \llparenthesis V \rrparenthesis_{\mathsf{r}\alpha'} u\big) \cap \gamma(v) = \emptyset$$
$$\text{and } \big(\llparenthesis V \rrparenthesis_{\mathsf{w}\alpha} u \cdot \mathcal{P}_{V,u,v} \cdot \llparenthesis V \rrparenthesis_{\mathsf{r}\alpha'} u\big) \cap \gamma(v) = \emptyset, \text{ for all } \alpha, \alpha' \in \mathcal{A}\,[\![\tau]\!]\,, \alpha \neq \alpha'.$$

Function stability specifications are interpreted by the following abbreviation:

Definition 5.16 (Function stability)

$$\mathfrak{M} \models \boldsymbol{\nabla}_{\overline{\sigma}\to\sigma}(F) \text{ if and only if } \mathfrak{M} \models \nabla x_1 : \sigma_1 \ldots \nabla x_k : \sigma_k . \boldsymbol{\nabla}_\sigma(F x_1 \cdots x_k),$$

where $\overline{\sigma} \to \sigma = \sigma_1 \to \cdots \to \sigma_k \to \sigma$ *and* $x_i : \sigma_i$ *are distinct identifiers not free in* F.

In other words, a function is stable if and only if its application to any stable arguments is itself stable.

Using a stability predicate we can indicate that even though certain phrases are instrumented with side-effects, they can still be used in statical reasoning.

Example 5.2
$\models \boldsymbol{\nabla}_{\mathbf{expint}\to\mathbf{expint}}(\lambda x{:}\mathbf{expint}.\, v := !v + 1; x)$
$\not\models \boldsymbol{\nabla}_{\mathbf{varint}\to\mathbf{expint}}(\lambda v{:}\mathbf{varint}.\, v := !v + 1; !v)$

The equivalence specification is interpreted by semantic equality.

Definition 5.17 (Equivalence)

$$\langle u, v\rangle \models_\Gamma P \equiv_\theta P' \text{ if and only if } [\![P]\!]\, u \cap \gamma(v) = [\![P']\!] u \cap \gamma(v).$$

Example 5.3 $\nabla x{:}\mathbf{varint}.\, x := 0; \mathbf{if}\ !x = 0\ \mathbf{then\ diverge\ else\ skip} \equiv_{\mathbf{comm}} \mathbf{diverge}$.

The following theorem shows that the specifications formulated using the language presented here can be verified using model checking.

Theorem 5.1 (Decidability of verification) *Validity of specifications is decidable: For any* $\Gamma \vdash S : \mathbf{spec}$, $\Gamma \models S$ *is decidable.*

We first prove the following intermediate result:

Lemma 5.1 (Specification truth decidability) *For any model* $\mathfrak{M}$ *and* $\Gamma \vdash S : \mathbf{spec}$, $\mathfrak{M} \models_\Gamma S$ *is decidable.*

PROOF:
The proof is by induction on the syntax of S.

Base cases:

- **absurd**: trivial;

- **Hoare triple**: by definition, for any Hoare triple, $\langle u, v\rangle \models \{A\}\ M\ \{A'\}$ if and only if for all $\omega \in \mathcal{A}^*$, $(\omega \cdot [\![A]\!]\, u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt$ implies $(\omega \cdot [\![M; A']\!]\, u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt$, where $\mathcal{A} = \mathcal{A}\,[\![\Gamma]\!]$.

  The languages $[\![A]\!]\, u$, $\gamma(v)$ and $\mathcal{A}^*$ are regular. $\gamma(v)$ is regular because it is an intersection of regular languages:

  - every $\gamma(x)$ is regular, according to definitions 4.17, 5.1 and 5.2 and propositions 5.1 and 5.2.
  - every $\widetilde{\gamma(x)}$ is regular, because regular languages are closed under shuffle.

  Therefore, the language

  $$\mathcal{L} = \{\omega \in \mathcal{A}^* \mid (\omega \cdot [\![A]\!]\, u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt\} = (\mathcal{A}^* \cdot [\![A]\!]\, u) \cap \gamma(v) \cap \mathcal{A}^* \cdot tt$$

  is also regular. Finding $\mathcal{L}$ is obviously decidable.

  The truth of the Hoare triple reduces then to verifying that

  $$(\mathcal{L} \cdot [\![M; A']\!]\, u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt,$$

  which is decidable because all languages involved are regular.

- **Stability**: from definition (5.14 and 5.15), verifying a stability specification reduces to checking regular-language emptiness, which is a decidable problem.

- **Equivalence**: from definition (5.17), checking an equivalence specification reduces to checking regular language equality, which is a decidable problem.

Inductive step:

- **Connectives**: verifying a specification with a connective requires verifying each of the components. Verifying the connectives is a decidable problem, by induction hypothesis.

- **Universal quantifier:** verifying that $\langle u, v\rangle \models \forall x : \theta.S$ reduces to verifying: $\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \epsilon)\rangle \models S$. Both are decidable, by induction hypothesis.

- **Stability quantifier:** verifying that $\langle u, v\rangle \models \nabla x : \theta.S$ reduces to verifying: $\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \gamma^x_\theta)\rangle \models S$. Both are decidable, by induction hypothesis.

END OF PROOF.

Notice that the absence of quantifiers in the semantics and the "flat" interpretation of specification quantifiers are essential in making this decidability result quite obvious.

PROOF: (*of Theorem 5.1 on the page before*)

By definition, validity of specification S reduces to verifying that for any model $\mathfrak{M}_\Gamma = \langle u_\Gamma, v_\Gamma\rangle$, $\mathfrak{M}_\Gamma \models S$, which is decidable, from Lemma 5.1 on the preceding page (Specification truth decidability). END OF PROOF.

```
x:expint, ovf:varbool ⊢
    newint fn1 in
    newint fn2 in
    newint n in
    fn1:=1; fn2:=1; n:=1; ovf:=false;
    while n<x do
        if fn2 > MAX-fn1 then (
            n:=!x;
            ovf:=true;)
        else (
            newint temp in
            temp:=!fn2;
            fn2:=!fn1+!fn2;
            fn1:=!temp;
            n:=!n+1);
    !fn1
```

Figure 5.2: Active expression computing Fibonacci numbers

We have presented in this section a specification language which is suitable for model checking program fragments of IA with active expressions.

For example, the active expression in Figure 5.2 computes Fibonacci numbers efficiently. The expression returns the `x`-th Fibonacci number, where `x` is an external variable, and it also sets a global flag `ovf` in case of overflow. We assume `MAX` is a language constant representing the maximum representable integer.

Using the specification language of this section we can validate through model-checking specifications such as:

$$\nabla x : \mathbf{expint}.\nabla \mathit{ovf} : \mathbf{varbool}.\boldsymbol{\nabla}_{\mathbf{expint}}(E)$$
$$\nabla x : \mathbf{expint}.\nabla \mathit{ovf} : \mathbf{varbool}.\nabla v : \mathbf{varint}.\{\mathbf{true}\}\ v := E\ \{\mathbf{not}\ !\mathit{ovf}\ \mathbf{implies}\ !v = \mathbf{Fibonacci}(!x)\}$$
$$\nabla \mathit{ovf} : \mathbf{varint}.\{\mathbf{let}\ \mathit{fib}\ \mathbf{be}\ \lambda x : \mathbf{exp}\tau.E\ \mathbf{in}\ \mathit{fib}(6) + \mathit{fib}(7) = \mathit{fib}(8)\ \mathbf{or}\ !\mathit{ovf}\},$$

where E is the program in Figure 5.2 and $\mathbf{Fibonacci}(n)$ is the *mathematical* function which returns the n-th Fibonacci number, as opposed to the *implementation* E.

In the examples above, the convenience of implementing E as an active expression is obvious. Reformulating the last specification using a **comm** implementation of E and its effects specified using Hoare triples would be quite awkward.

Chapter 6

Logical Properties of Specifications

The want of logic annoys. Too much logic bores. André Gide

6.1 Inferential reasoning

In this chapter we will show how we can reason inferentially about specifications using axioms, inference rules and substitution rules. We will present the logical rules in natural deduction form.

Definition 6.1 (Semantic consequence) *If Σ is a list of specifications $\Gamma \vdash S_i : \mathbf{spec}$ we say that S is a* semantic consequence *of Σ, denoted by $\Sigma \models_\Gamma S$, if and only if $\mathfrak{M}_\Gamma \models S_i$, for all specifications S_i in Σ implies $\mathfrak{M}_\Gamma \models S$.*

In the above, $\mathfrak{M}_\Gamma$ is as defined in Definition 5.8 on page 94.

Definition 6.2 (Logical consequence) *If Σ is a list of specifications $\Gamma \vdash S_i : \mathbf{spec}$ we say that S is a* logical consequence *of Σ, denoted by $\Sigma \vdash_\Gamma S$, if and only if S can be derived from Σ.*

We drop the Γ index if it causes no confusion in context.

The derivation rules will be presented in this chapter. We will show that:

Theorem 6.1 (Soundness) *$\Sigma \vdash_\Gamma S$ implies $\Sigma \models_\Gamma S$.*

PROOF: By induction on the derivation: we will show, throughout this chapter, that every axiom is sound and all inference rules preserve soundness. END OF PROOF.

We will study the logical properties of: specification connectives, quantifiers, stability specifications, assertions, and programming constructs.

The axioms and inference rules that we use are essentially the axioms and inference rules of Hoare logic, extended to procedures and adapted to deal with the phenomena of side effects and interference. Our formalism uses stability, expressed as a quantifier, to deal with both these troublesome phenomena. A stable identifier represents both a guarantee that the non-local entity denoted by the identifier is "well behaved," as discussed in more detail in the previous chapter, as well as a promise that no non-local entities interfere with it.

We also use a concept of non-interference. Computationally, two phrases do not interfere if they do not assign to each other's stable variables. The regular-language semantic interpretation of this notion is immediate, as is a correct, but not complete, syntactic characterization of non-interfering phrases. However, in our specification language non-interference is not a first-class citizen, but only a side-condition used in inferences.

Another side-condition used in inferences is normal termination. Our assertions can have computational effects, which include non-termination. In some cases, diverging assertions may have anomalous logical properties; the simplest example is the fact that assertion-level conjunction does not correspond to specification-level conjunctions:

$$\{\textbf{diverge and false}\} \not\Leftrightarrow \{\textbf{diverge}\} \wedge \{\textbf{false}\}.$$

We use stability quantifiers much as Reynolds's specification logic uses non-interference assumptions to rule out objects with behaviours incompatible with the Hoare-like axioms we want to use. This makes the inference rules rather peculiar, because the specifications must always be prefixed by the stability quantifiers.

Consider the following simple example. In Hoare logic we can easily prove that

$$\{\textbf{true}\}\ x := !y \times !y\ \{!x = !y \times !y\}.$$

In a procedural language we might want to formulate a similar specification:

$$\{\textbf{true}\}\ x := \mathit{sqr}(!y)\ \{x = \mathit{sqr}(!y)\}.$$

Of course, this cannot be done directly because of interference and side effects. In Reynolds's specification logic we must "protect" all identifiers involved using non-interference assumptions:

$$y \# \mathit{sqr} \wedge x \# y \wedge x \# \mathit{sqr} \Rightarrow \{\textbf{true}\}\ x := \mathit{sqr}(!y)\ \{x = \mathit{sqr}(!y)\}.$$

This is not enough in the presence of side effects, because expression-like functions such as *sqr* may produce interference and may even interfere with themselves through side-effects. Our formulation is (omitting type information for conciseness):

$$\nabla x.\nabla y.\nabla \mathit{sqr} \bullet \{\textbf{true}\}\ x := \mathit{sqr}(!y)\ \{!x = \mathit{sqr}(!y)\}.$$

This, indeed, can be semantically validated. But how do we *derive* this valid specification? Remember the original Hoare axiom schema for assignment:

$$\{A[V/E]\}\ V := E\ \{A\}.$$

Since substitution is the same as function application and since assertions are programming phrases, it is more convenient to write the rule using application. Also, the stability of the variable being assigned to, of the expression being assigned and of the assertion itself are important. The *instance* of the assignment axiom schema we must use is:

$$\frac{\nabla x.\nabla y.\nabla sqr \bullet \boldsymbol{\nabla}\big(sqr(!y)\big) \quad \nabla x.\nabla y.\nabla sqr \bullet \boldsymbol{\nabla}(x) \quad \nabla x.\nabla y.\nabla sqr \bullet \boldsymbol{\nabla}\big(\lambda e.e = sqr(!y)\big)}{\nabla x.\nabla y.\nabla sqr \bullet \{sqr(!y) = sqr(!y)\}\ x := sqr(!y)\ \{!x = sqr(!y)\}}$$

This is a correct instance of the assignment axiom schema because the assertion, the expression and the variable phrases do not interfere. In the premises, there are no overt assignments to the stable variables in any of the phrases above. The pre-condition can be strengthened to **true** because stable phrases have the "normal" mathematical and logical properties, including reflexivity of equality.

Suppose that we have proved that:

$$\nabla x.\nabla y.\nabla z.\nabla sqr \bullet \{\mathbf{true}\}\ x := sqr(!y);\ z := sqr(!y)\ \{!x = !z\},$$

using the assignment axiom twice, the Hoare-like axiom for sequencing and mathematical reasoning in the post-condition, and now we want to eliminate the stability quantifier ∇sqr, by actually defining the function *sqr*. The definition of the function must be stable. To make the example more interesting, let us look at a function with side effects, such as:

$$SQR \stackrel{\text{def}}{=} \lambda t.\mathbf{if}\ t > \mathbf{N}\ \mathbf{then}\ err := 1;\ 0\ \mathbf{else}\ err := 0;\ t \times t.$$

This is a simple mechanism for overflow-control. If the argument is larger than some language-specific constant $\mathbf{N}$ then an error flag is set to 1 and the default value 0 is returned. Otherwise the error flag is set to 0 and the square is actually computed. The phrase *SQR* is stable: $\boldsymbol{\nabla}(SQR)$. In order to proceed with the substitution, the prefixing quantifiers of the two specifications must match. We can introduce $\nabla x.\nabla y.\nabla z$ in the stability specification of *SQR* because none of these variables occur freely in it.

Since the sole variable assigned to, *err*, is not among the stable variables of the Hoare triple, substitution can be done safely, resulting in:

$$\nabla x.\nabla y.\nabla z. \bullet \{\mathbf{true}\}\ \mathbf{let}\ sqr\ \mathbf{be}\ SQR\ \mathbf{in}\ x := sqr(!y);\ z := sqr(!y)\ \{!x = !z\}.$$

If the implementation of *sqr* was

$$SQR \stackrel{\text{def}}{=} \lambda t.\mathbf{if}\ t > \mathbf{N}\ \mathbf{then}\ y := \mathbf{true}; 0\ \mathbf{else}\ y := \mathbf{false}; t \times t,$$

that is, variable y was used as the overflow flag, substitution is impossible, because *SQR* now interferes with the Hoare triple by writing to y, one of its stable variables.

6.2 Specification connectives and quantifiers

The inference rules for specification connectives are the usual ones:

INFERENCE RULES

$$\wedge\mathrm{I}\ \frac{S \qquad S'}{S \wedge S'} \qquad \wedge\mathrm{E1}\ \frac{S \wedge S'}{S} \qquad \wedge\mathrm{E2}\ \frac{S \wedge S'}{S'}$$

$$\Rightarrow\mathrm{I}\ \frac{\begin{array}{c}[S]\\ \vdots \\ S'\end{array}}{S \Rightarrow S'} \qquad \Rightarrow\mathrm{E}\ \frac{S \Rightarrow S' \qquad S}{S'} \qquad \text{absurd}\ \frac{\textbf{absurd}}{S}$$

PROOF OF SOUNDNESS:
These proofs are standard first-order logic soundness proofs. We only give two such proofs, for illustration.

- $\wedge$*-introduction*: If $\Sigma \vdash S$ and $\Sigma \vdash S'$ then, according to the rule $\Sigma \vdash S \wedge S'$. By induction hypothesis, $\Sigma \models S$ and $\Sigma \models S'$. By definition, it follows that if $\mathfrak{M}_\Gamma \models S_i$ for all S_i in Σ, $\mathfrak{M}_\Gamma \models S$ and $\mathfrak{M}_\Gamma \models S'$. By semantic definition of $\wedge$, this implies that $\mathfrak{M}_\Gamma \models S \wedge S'$, so $\Sigma \models S \wedge S'$.

- $\Rightarrow$*-introduction*: If $\Sigma, S \vdash S'$ then, according to the rule, $\Sigma \vdash S \Rightarrow S'$. By induction hypothesis, $\Sigma, S \models S'$. By definition, it follows that if $\mathfrak{M}_\Gamma \models S_i$ for all S_i in Σ and $\mathfrak{M}_\Gamma \models S$, then $\mathfrak{M}_\Gamma \models S'$. Therefore, if $\mathfrak{M}_\Gamma \models S$, then $\mathfrak{M}_\Gamma \models S'$, which means, according to the semantics of $\Rightarrow$, $\mathfrak{M}_\Gamma \models S \Rightarrow S'$. So $\Sigma \models S \Rightarrow S'$.

END OF PROOF.

Although the interpretation of the universal quantifier in this specification logic is not standard, it still has introduction and elimination rules similar to the usual ones:

INFERENCE RULES

$$\forall I\ \frac{S}{\forall x : \theta.S} \qquad \forall E\ \frac{\forall x : \theta.S}{S[x/P]}\ (\Gamma \vdash P : \theta)$$

In $\forall$ introduction, x must not be free in the hypotheses of S; in $\forall$ elimination the substitution of P for x must be free, i.e. the free identifiers of P are not captured by quantifiers or binding programming structures (lambda, local variable definition).

PROOF OF SOUNDNESS:

- $\forall$–*introduction*: By induction hypothesis, $\Sigma \models_{\Gamma,x:\theta} S$, i.e. if $\mathfrak{M}_{\Gamma,x:\theta} \models S_i$, for all S_i in Σ, then $\mathfrak{M}_{\Gamma,x:\theta} \models S$. $\mathfrak{M}_{\Gamma,x:\theta} = \langle u_{\Gamma,x:\theta}, v_{\Gamma,x:\theta}\rangle$ (as defined in Definition 5.8 on page 94), $\mathrm{dom}(u) = \mathrm{dom}(v) = \mathrm{dom}(\Gamma, x : \theta)$. But $\langle u, v\rangle \models S_i$ implies that for any regular languages $\mathcal{L}, \mathcal{L}'$, we have that $\big\langle (u \mid x \mapsto \mathcal{L}), (v \mid x \mapsto \mathcal{L}')\big\rangle \models S_i$ because x is not free in S_i so $\big\langle (u \mid x \mapsto \mathcal{L}), (v \mid x \mapsto \mathcal{L}')\big\rangle \models S$, according to the induction hypothesis. In particular, this is the case for $\mathcal{L} = \mathcal{K}^x_\theta, \mathcal{L}' = \epsilon$. Therefore, $\langle u, v\rangle \models \forall x : \theta.S$.
- $\forall$–*elimination*: We use Lemma 6.1, below. Since the model is $\mathfrak{M}_\Gamma = \langle u_\Gamma, v_\Gamma\rangle$. By definition, $v_\Gamma(x) = \epsilon$ for all free variables of P.

END OF PROOF.

Lemma 6.1 (Semantic substitution in specifications) *For any model $\mathfrak{M} = \langle u, v\rangle$ and specification S, if $\mathfrak{M} \models_\Gamma \forall x : \theta.S$ then $\mathfrak{M} \models_\Gamma S[x/P]$ for any $\Gamma \vdash P : \theta$ such that for all $x' \in$ Free(P), $v(x') = \epsilon$.*

So terms can be freely substituted for a universally quantified identifier so long as their own free identifiers are not subject to any dynamic constraints. Before we prove this, we prove a similar lemma at the level of assertions:

Lemma 6.2 (Semantic substitution in assertions) *Consider assertion $\Gamma, x : \theta \vdash A :$ **assert**, boolean value $\alpha \in \{tt, f\!f\}$, model $\mathfrak{M} = \langle u, v\rangle$ and state $\omega \in \eta(v)$ such that $u(x) = \mathcal{K}^x_\theta$ and $v(x) = \epsilon$.*

If $\omega \cdot (\!|A|\!)_\alpha u \cap \gamma(v) = \emptyset$ then for any term $\Gamma \vdash P : \theta$, such that for all $x' \in$ Free(P), $v(x') = \epsilon$ and $u(x') = \mathcal{K}^{x'}_{\theta'}$, we also have that $\omega \cdot (\!|A[x/P]|\!)_\alpha u \cap \gamma(v) = \emptyset$.

Informally, this lemma guarantees that we can substitute any term for any free variable, without changing the value produced by an assertion, as long as the free identifiers of the term are not subject to any stability constraints. In other words, free or universally quantified variables are, in some sense, irrelevant: they may be arbitrarily substituted with no effect on the truth value of the assertion.

PROOF: *(Of Lemma 6.2)*
We consider two cases depending on the type of x, the variable to be substituted:

- If x is of ground type, $(x : \sigma)$:

  The semantics of function application in IA is defined operationally using substitution (Figure 2.2 on page 16). Semantically, function application is defined using substitution at the level of traces (Definition 4.15 on page 62). Since the semantics is fully abstract is follows that source-level and trace-level substitution are equivalent, in the following sense: $(\!|A[x/P]|\!)_\alpha u = ((\!|A|\!)_\alpha u)[\kappa]$,
  where $\kappa : \left(\sum_{\mathsf{q}\in\mathsf{Q}[\![\sigma]\!]} \mathsf{q} \cdot \mathsf{A}_\mathsf{q}\, [\![\sigma]\!]\right)^{\langle x\rangle} \to [\![P]\!]\, u$, with $\kappa(\mathsf{q}^{\langle x\rangle} \cdot \mathsf{a}^{\langle x\rangle}) = (\!|P|\!)_\mathsf{a} u$.

If for all $\omega \in \eta(v)$ $\omega \cdot (\!|A|\!)_\alpha u \cap \gamma(v) = \emptyset$ then for all $\omega_A \in (\!|A|\!)_\alpha u$, $\omega \cdot \omega_A \notin \gamma(v)$. Therefore, $\omega \cdot \omega_A[\mathsf{q}^{\langle x \rangle} \cdot \mathsf{a}^{\langle x \rangle} / \omega_P] \notin \gamma(v)$ for all $\omega_P \in (\!|P|\!)_\mathsf{a} u$. The proof is by contradiction: if $\omega \cdot \omega_A[\mathsf{q}^{\langle x \rangle} \cdot \mathsf{a}^{\langle x \rangle} / \omega_P] \in \gamma(v)$ then $\omega \cdot \omega_A \in \gamma(v)$ because $\left(\{\mathsf{q}^{\langle x \rangle}, \mathsf{a}^{\langle x \rangle}\} \cup \lfloor \omega_P \rfloor\right) \cap \lfloor v(x') \rfloor = \emptyset$ for all $x' \in \mathrm{dom}(v)$. More informally, we are only substituting traces of unconstrained actions for traces of unconstrained actions.

- if x is a first-order function ($x : \sigma_1 \to \cdots \to \sigma_k \to \sigma$):

  In $A[x/P]$ we can assume without loss of generality, according to the Reduction Lemma (4.4 on page 75), that A is in let-free, β-normal form. Using the substitution lemma and the semantics of application it follows that:

$$(\!|A[x/P]|\!)_\alpha u = \left((\!|A|\!)_\alpha u\right)[\kappa], \tag{6.1}$$

  where $\kappa : [\![x(A_1, \ldots, A_k)]\!]\, u \to [\![P(A_1, \ldots, A_k)]\!]\, u$, such that $\kappa(\omega) = (\!|P(A_1, \ldots, A_k)|\!)_\mathsf{a} u$, for all $\omega \in (\!|x(A_1, \ldots, A_k)|\!)_\mathsf{a}$.

  It is important that A is in normal form, so it lacks function definitions, because only then are all occurrences of x in applications $x(A_1, \ldots, A_k)$.

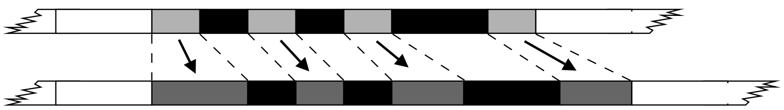
  By $\mathcal{K}^x_{\theta;\mathsf{a}}$ let us denote the regular language such that $\mathsf{q} \cdot \mathcal{K}^x_{\theta;\mathsf{a}} \cdot \mathsf{a} = \mathcal{K}^x_\theta$, for $\mathsf{q} \in \mathsf{Q}\,[\![\sigma]\!]$, $\mathsf{a} \in \mathsf{A}_\mathsf{q}\,[\![\sigma]\!]$; the relation between $\mathcal{K}^x_{\theta;\mathsf{a}}$ and $\mathcal{K}^x_\theta$ is the same as between $[\![-]\!]$ and $(\!|-|\!)_\mathsf{a}$.

  Then, using the definition of function application, substitution κ in equation 6.1 has the property that

$$\kappa(\omega) = (\!|P|\!)_\mathsf{a} u[\kappa'], \text{ for all } \omega \in \mathcal{K}^x_{\theta;\mathsf{a}_i}[\kappa']. \tag{6.2}$$

  where $\kappa'\left(\mathsf{q}_i^{\langle xi \rangle} \cdot \mathsf{a}_i^{\langle xi \rangle}\right) = (\!|A_i|\!)_{\mathsf{a}_i}$.

  The result of the substitution is illustrated as follows:



  Substring from $\mathcal{K}^x_\theta$ Substring from A

  Substring from P Substring from A_i

  The picture above shows a segment of a string in $(\!|A|\!)_\alpha u$ where the double substitution has occurred. The arrows indicate the substrings being replaced.

  Neither the substrings being replaced nor their replacements contain symbols in $\lfloor \gamma(x) \rfloor$, for any x in the environment, so they are not constrained by $\gamma(v)$. The string prior to substitution is in $\gamma(v)$ if and only if the string after replacement is in $\gamma(v)$. END OF PROOF

Now we can prove Lemma 6.1 on the preceding page.
PROOF:
By induction on the syntax of S

($\wedge$):

$$\begin{array}{ll} & \langle u,v\rangle \models \forall x:\theta.S_1 \wedge S_2 \\ \therefore & \big\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \epsilon)\big\rangle \models S_1 \wedge S_2, \quad \text{from semantic definition of } \forall \\ \therefore & \big\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \epsilon)\big\rangle \models S_i, \quad \text{for } i=1,2, \text{ semantic definition of } \wedge \\ \therefore & \langle u,v\rangle \models \forall x:\theta.S_i, \quad \text{semantic definition of } \forall \\ \therefore & \langle u,v\rangle \models S_i[x/P], \quad \text{from induction (structural on } S\text{) hypothesis} \\ \therefore & \langle u,v\rangle \models S_1[x/P] \wedge S_2[x/P], \quad \text{semantic definition of } \wedge \\ \therefore & \langle u,v\rangle \models (S_1 \wedge S_2)[x/P], \quad \text{definition of substitution.} \end{array}$$

($\Rightarrow$): similar to the above.

(absurd): trivial.

($\forall$): For $\forall x:\theta.\forall x':\theta'.S$, if $x = x'$ then $(\forall x:\theta.\forall x:\theta.S)[x/P] = \forall x:\theta.S$.
So $\mathfrak{M} \models \forall x:\theta.\forall x:\theta.S$ implies trivially $\mathfrak{M} \models \forall x:\theta.S$.
If $x \neq x'$ then:

$$\begin{array}{ll} & \langle u,v\rangle \models \forall x:\theta.\forall x':\theta'.S \\ \therefore & \big\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \epsilon)\big\rangle \models \forall x':\theta'.S, \quad \text{semantic definition of } \forall \\ \therefore & \big\langle (u \mid x \mapsto \mathcal{K}^x_\theta \mid x' \mapsto \mathcal{K}^{x'}_{\theta'}), (v \mid x \mapsto \epsilon \mid x' \mapsto \epsilon)\big\rangle \models S, \quad \text{semantic definition of } \forall \\ \therefore & \big\langle (u \mid x' \mapsto \mathcal{K}^{x'}_{\theta'} \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x' \mapsto \epsilon \mid x \mapsto \epsilon)\big\rangle \models S, \quad \text{because } x \neq x' \\ \therefore & \big\langle (u \mid x' \mapsto \mathcal{K}^{x'}_{\theta'}), (v \mid x' \mapsto \epsilon)\big\rangle \models \forall x:\theta.S, \quad \text{semantic definition of } \forall \\ \therefore & \big\langle (u \mid x' \mapsto \mathcal{K}^{x'}_{\theta'}), (v \mid x' \mapsto \epsilon)\big\rangle \models S[x/P], \quad \text{induction hypothesis} \\ \therefore & \big\langle u,v\big\rangle \models \forall x':\theta'.\big(S[x/P]\big), \quad \text{semantic definition of } \forall \\ \therefore & \big\langle u,v\big\rangle \models (\forall x':\theta'.S)[x/P], \quad P \text{ free for } x, \text{ definition of substitution} \end{array}$$

(∇): similar to the above.

(Hoare triple): immediately from Lemma 6.2 on page 105, because substituting P for x does not change the value of either the precondition or the postcondition. For all $\omega \in \eta(v)$, $u(x) = \mathcal{K}^x_\theta$, $v(x) = \epsilon$:

$$\begin{array}{l} (\omega \cdot (\!|A|\!)_{\mathit{ff}}\, u) \cap \gamma(v) = \emptyset \text{ implies } (\omega \cdot (\!|A[x/P]|\!)_{\mathit{ff}}\, u) \cap \gamma(v) = \emptyset \\ (\omega \cdot (\!|M;A'|\!)_{\mathit{ff}}\, u) \cap \gamma(v) = \emptyset \text{ implies } (\omega \cdot (\!|M;A'[x/P]|\!)_{\mathit{ff}}\, u) \cap \gamma(v) = \emptyset \end{array}$$

(Stability specifications): if

$$\big((\!|E|\!)_\alpha u \cdot \mathcal{P}_{E,u,v} \cdot (\!|E|\!)_{\alpha'} u\big) \cap \gamma(v) = \emptyset$$

then

$$\big((\!|E[x/P]|\!)_\alpha u \cdot \mathcal{P}_{E[x/P],u,v} \cdot (\!|E[x/P]|\!)_{\alpha'} u\big) \cap \gamma(v) = \emptyset,$$

by the same argument: the unconstrained traces of x are replaced by the unconstrained traces of P. In fact, Lemma 6.2 on page 105 can be generalized immediately to all ground types. Variable stability is similar.

First-order stability is an abbreviation defined using stability quantifiers and ground-type stability.

(Equivalence): Similar argument.

END OF PROOF.

In our specification language, universally quantified variables are not really useful. The workhorses of this specification language are the generalized quantifiers, so the first truly interesting rules are the rules for introducing or eliminating the stability quantifier. The introduction rule is similar to that for the universal quantifier:

INFERENCE RULE

$$\nabla I \, \frac{S}{\nabla x : \theta . S} \; (x \text{ not free in } \Sigma)$$

But the elimination rule is quite different:

INFERENCE RULE

$$\nabla E \, \frac{\Psi \bullet \nabla x : \theta . S \qquad \Psi \bullet \boldsymbol{\nabla}_\theta(P)}{\Psi \bullet S[x/P]} \; (x \text{ not in } \Psi \text{ and } P \mathrel{\#_x} \Psi \bullet S)$$

In the above, $\Psi \bullet -$ represents a sequence of quantifiers, universal or stability. The $\bullet$ sign does not play any special syntactic role, we only use it for emphasis, instead of a simple dot. The substitution of P must be free for x in S, i.e. no free identifiers of P may occur in the scope of a specification quantifier or or programming language binding construct (lambda or new), binding an identifier with the same name.

$P \mathrel{\#_x} \Psi \bullet S$ is a *non-interference condition* stipulating that P and S do not write to each other's stable variables, as declared by Ψ, which we shall define shortly (Definition 6.5 on page 110). The subscript identifier x indicates that the non-interference condition only applies to those IA terms occurring in specifications which contain x freely.

But first we must introduce some auxiliary concepts.

Definition 6.3 (Model consistent with Ψ) *Given a model $\mathfrak{M} = \langle u, v\rangle$ and a sequence of quantifiers Ψ, we define $\mathfrak{M} \diamond \Psi = \langle u \diamond \Psi, v \diamond \Psi\rangle$, which we call the* model consistent with Ψ, *such that for all specifications S, $\mathfrak{M} \models \Psi \bullet S$ if and only if $\mathfrak{M} \diamond \Psi \models S$.*

More explicitly, using the semantics of the quantifiers, this means that $u \diamond \Psi, v \diamond \Psi$ are:

$$(u \diamond \Psi)x = \begin{cases} u(x) & x \text{ not in } \Psi \\ \mathcal{K}^x_\theta & \forall x : \theta \text{ in } \Psi \\ \mathcal{K}^x_\theta & \nabla x : \theta \text{ in } \Psi \end{cases} \quad \text{and} \quad (v \diamond \Psi)x = \begin{cases} v(x) & x \text{ not in } \Psi \\ \epsilon & \forall x : \theta \text{ in } \Psi \\ \gamma^x_\theta & \nabla x : \theta \text{ in } \Psi \end{cases}$$

When Ψ is empty, then $\mathfrak{M} \diamond \Psi = \mathfrak{M}$.

Definition 6.4 (Trace non-interference) *Two sets of traces $\mathcal{L}_0, \mathcal{L}_1$ are said not to interfere in a frame v, denoted by $\mathcal{L}_0 \#_v \mathcal{L}_1$, iff $\left\lfloor \mathcal{L}_i \upharpoonright read^{\langle x \rangle} \right\rfloor = \emptyset$ or $\left\lfloor \mathcal{L}_{1-i} \upharpoonright write(\alpha)^{\langle x \rangle} \right\rfloor = \emptyset$, $i = 1, 2$, for all $x \in dom(v)$ such that $v(x) \in \{\gamma^x_{\mathbf{var}\tau}, \gamma^x_{\overline{\sigma} \to \mathbf{var}\tau}\}$, $\alpha \in \mathcal{A}[\![\tau]\!]$.*

Intuitively, two sets of traces are non-interfering if they do not contain assignments to each other's stable variable. This notion is, informally, a generalization of the notion of passive trace (Definition 5.13 on page 96).

The following rather obvious property of non-interfering languages of traces is essential, because many of the proofs hinge on the ability to insert, remove or substitute strings under the intersection with the language of constraints defined by the stability quantifiers.

Lemma 6.3 (Non-interfering string substitution) *For all sets of traces $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$ over alphabet $\mathcal{A}$ and for any frame v over the same alphabet:*

1. *if $\mathcal{L}_1 \cdot \mathcal{L}_3 \cap \gamma(v) = \emptyset$ and $\mathcal{L}_2 \#_v \mathcal{L}_3$ then $\mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \mathcal{L}_3 \cap \gamma(v) = \emptyset$,*

2. *if $\mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \mathcal{L}_3 \cap \gamma(v) = \emptyset$ and $\mathcal{L}_1 \cdot \mathcal{L}_2 \cap \gamma(v) \neq \emptyset$ and $\mathcal{L}_2 \#_v \mathcal{L}_3$ then $\mathcal{L}_1 \cdot \mathcal{L}_3 \cap \gamma(v) = \emptyset$.*

PROOF:

1. If $\mathcal{L}_1 \cdot \mathcal{L}_3 \cap \gamma(v) = \emptyset$ then there must exist $x \in \text{dom}(v)$ such that for all $\omega_i \in \mathcal{L}_i$, $\omega_1 \cdot \omega_3 \notin \widetilde{\gamma(x)}$ for all $\omega_i \in \mathcal{L}_i$ **or** one of $\mathcal{L}_i$ is empty. The second case is trivial.

   We analyze the cases of possible $\gamma(x)$.

   - if $\gamma(x) = \epsilon$ then $\widetilde{\gamma(x)} = \mathcal{A}^*$ so $\omega_1 \cdot \omega_3 \notin \widetilde{\gamma(x)}$, which is not possible. This is not a valid case.
   - if $\gamma(x) = \gamma^x_{\mathbf{exp}\tau}$ then it must be the case that $\omega_1 \cdot \omega_3 \notin \gamma^x_{\mathbf{exp}\tau} \bowtie \mathcal{A}^{\overline{x}}$ or, conversely $\omega_1 \cdot \omega_3 \upharpoonright \mathcal{A}^{\langle x \rangle} \notin \gamma^x_{\mathbf{exp}\tau}$. This means that ω_i contain symbols $\alpha_i^{\langle x \rangle}$ such that $\alpha_1 \neq \alpha_3$. Then it does not matter what $\mathcal{L}_2$ contains, still $\omega_1 \cdot \mathcal{L}_2 \cdot \omega_3 \upharpoonright \mathcal{A}^{\langle x \rangle} \notin \gamma^x_{\mathbf{exp}\tau}$.
   - if $\gamma(x) = \gamma^x_{\mathbf{var}\tau}$ then it must be the case that, following the same reasoning: either the last occurrence of $\alpha_1^{\langle x \rangle}$ in ω_1 and the first occurrence of $\alpha_3^{\langle x \rangle}$ in ω_3 are different, or the first occurrence of $write(\alpha_1)^{\langle x \rangle}$ in ω_1 and the first occurrence of $\alpha_3^{\langle x \rangle}$ in ω_3 are such that $\alpha_1 \neq \alpha_3$. In either case, from the non-interference condition, string $\mathcal{L}_2$ does not contain any $write(\alpha_3)^{\langle x \rangle}$, the only possibility which could make $\omega_1 \cdot \mathcal{L}_2 \cdot \omega_3 \upharpoonright \mathcal{A}^{\langle x \rangle} \notin \gamma^x_{\mathbf{var}\tau}$ false.

- if $\gamma(x) = \gamma^{x}_{\vec{\sigma}\to\mathbf{exp}\tau}$ then $\omega_1 \cdot \omega_3 \notin \widetilde{\gamma^{x}_{\vec{\sigma}\to\mathbf{exp}\tau}}$ means that, by the definition of first-order expression-returning function stability (Definition 5.3 on page 84),

$$\omega_1 \cdot \omega_3 = \omega' \cdot q^{\langle x\rangle} \cdot \omega_a \cdot \alpha_a^{\langle x\rangle} \cdot \omega'' \cdot q^{\langle x\rangle} \cdot \omega_b \cdot \alpha_b^{\langle x\rangle} \cdot \omega''', \tag{6.3}$$

such that $\omega_a \cdot \omega_b \in \bigcap_{1\le i\le k} \widetilde{\gamma^{fi}_{\sigma_i}}$ and $\alpha_a \neq \alpha_b$. Notice that ω_a is a substring of ω_1, and ω_b is a substring of ω_2. As such, inserting ω_2 can only happen within ω'', so the validity of equation 6.3 is not affected;

- if $\gamma(x) = \gamma^{x}_{\vec{\sigma}\to\mathbf{var}\tau}$, similar.

2. similar analysis.

End Of Proof.

Definition 6.5 (Non-interference at x) *We define non-interference at x of a phrase P and a specification $\Psi \bullet S$, denoted by $P \#_x \Psi \bullet S$ by induction on S:*

$$\begin{array}{l}
P \#_x \Psi \bullet S \wedge S' \;\; \textit{if} \;\; P \#_x \Psi \bullet S \textit{ and } P \#_x \Psi \bullet S' \\
P \#_x \Psi \bullet S \Rightarrow S' \;\; \textit{if} \;\; P \#_x \Psi \bullet S \textit{ and } P \#_x \Psi \bullet S' \\
P \#_x \Psi \bullet \forall x' : \theta'.S \;\; \textit{if} \;\; x = x' \;\; \textit{or} \;\; P \#_x \Psi \bullet S \\
P \#_x \Psi \bullet \nabla x' : \theta'.S \;\; \textit{if} \;\; x = x' \;\; \textit{or} \;\; P \#_x \Psi \bullet S \\
P \#_x \Psi \bullet \boldsymbol{\nabla}_\theta(P') \;\; \textit{if} \;\; [\![P]\!](u \diamond \Psi) \cap \gamma(v \diamond \Psi) \#_{v\diamond\Psi} [\![P']\!](u \diamond \Psi) \cap \gamma(v \diamond \Psi) \\
P \#_x \Psi \bullet \{A\}\ M\ \{A'\} \;\; \textit{if} \;\; [\![P]\!](u \diamond \Psi) \cap \gamma(v \diamond \Psi) \#_{v\diamond\Psi} [\![A]\!](u \diamond \Psi) \cap \gamma(v \diamond \Psi) \\
\qquad\qquad \textit{and} \;\; [\![P]\!](u \diamond \Psi) \cap \gamma(v \diamond \Psi) \#_{v\diamond\Psi} [\![M; A']\!](u \diamond \Psi) \cap \gamma(v \diamond \Psi).
\end{array}$$

What makes the substitution rule for stability (on page 108) unusual is that the substitution is happening *under* the quantifiers Ψ, and not at the outermost level. The reason for this is that the stability of a phrase is determined by the stability of its free identifiers. If assertions $\nabla x : \theta.S$ and $\boldsymbol{\nabla}_\theta(P)$ enjoy the same set of stable identifiers then P can be substituted for x, provided that P and S do not interfere at x.

Remark 6.1 *The definition of non-interference presented here is not syntactical, but semantic. It is possible to give a syntactical notion of non-interference. We will present it and discuss several important related issues in Section 6.6 (Side-conditions and semantic cheating).*

To make an appeal to the earlier mentioned idea of interface automaton, $\nabla x : \theta$ binds identifier x in the specification to an interface automaton corresponding to stable behaviour. The specification $\Psi \bullet \boldsymbol{\nabla}_\theta(P)$ represents the *refinement condition*, that phrase P behaves in a way compatible with ∇. Therefore, the substitution can take place.
Proof of Soundness: (*For rules ∇I and ∇E, page 108.*)

- ∇–*introduction*: same as the proof for $\forall$–introduction.

- ∇–*elimination*: immediate from Lemma 6.5 (proved on page 113).

END OF PROOF.

The semantic substitution lemmas 6.1 and 6.2 on page 105 deal with substitution in the case that the substituted identifier is unconstrained. For a stable variable they need to be adequately reformulated.

Definition 6.6 (Semantic non-interference at x) *We define non-interference at x for phrase P and specification S, denoted by $P \#_{x,\mathfrak{M}} S$ inductively on the syntax of S:*

$$\begin{array}{rll}
P \#_{x,\mathfrak{M}} S \wedge S' & \textit{if} & P \#_{x,\mathfrak{M}} S \textit{ and } P \#_{x,\mathfrak{M}} S' \\
P \#_{x,\mathfrak{M}} S \Rightarrow S' & \textit{if} & P \#_{x,\mathfrak{M}} S \textit{ and } P \#_{x,\mathfrak{M}} S' \\
P \#_{x,\mathfrak{M}} (\forall x' : \theta'.S) & \textit{if} & x = x' \quad \textit{or} \quad P \#_{x,\mathfrak{M}} S \\
P \#_{x,\mathfrak{M}} (\nabla x' : \theta'.S) & \textit{if} & x = x' \quad \textit{or} \quad P \#_{x,\mathfrak{M}} S \\
P \#_{x,\mathfrak{M}} \boldsymbol{\nabla}_\theta(P') & \textit{if} & [\![P]\!]\, u \cap \gamma(v) \;\#_v\; [\![P']\!]\, u \cap \gamma(v), \\
P \#_{x,\mathfrak{M}} \{A\}\ M\ \{A'\} & \textit{if} & [\![P]\!]\, u \cap \gamma(v) \;\#_v\; [\![A]\!]\, u \cap \gamma(v), \\
 & & \quad [\![P]\!]\, u \cap \gamma(v) \;\#_v\; [\![M; A']\!]\, u \cap \gamma(v).
\end{array}$$

The two definitions of non-interference at x are equivalent in the following sense:

Lemma 6.4 *For any model $\mathfrak{M} = \langle u, v\rangle$, and term $\Gamma \vdash P : \theta$, if $\mathfrak{M} \models \Psi \bullet S$ then*

$$P \#_x \Psi \bullet S \quad \textit{if and only if} \quad P \#_{x,\mathfrak{M}'} S,$$

for model $\mathfrak{M}' = \langle u \diamond \Psi, v \diamond \Psi\rangle$.

PROOF: Immediate from definitions. END OF PROOF.

Lemma 6.5 (Stable semantic substitution in specifications) *For any model $\mathfrak{M} = \langle u, v\rangle$ with $v(x) = \gamma_\theta^x$ and specification S, if $\mathfrak{M} \models_{\Gamma,x:\theta} S$ then $\mathfrak{M} \models_\Gamma S[x/P]$ for any $\Gamma \vdash P : \theta$ such that $\mathfrak{M} \models \boldsymbol{\nabla}_\theta(P)$ and $P \#_v S$.*

So stable terms can be freely substituted for stable identifiers as long as they do not interfere with the specification at the identifier being substituted. Before we prove this lemma we prove a similar lemma at the level of assertions:

Lemma 6.6 (Stable semantic substitution in assertions) *For any $\Gamma, x : \theta \vdash A :$ **assert**, boolean value $\alpha \in \{tt, ff\}$, model $\mathfrak{M} = \langle u, v\rangle$ with $v(x) = \gamma_\theta^x$ and state $\omega \in \eta(v)$, if $\omega \cdot (\!|A|\!)_\alpha u \cap \gamma(v) = \emptyset$, $\mathfrak{M} \models \boldsymbol{\nabla}_\theta(P)$ and $[\![P]\!]\, u \;\#_v\; [\![A]\!]\, u$, then $\omega \cdot (\!|A[x/P]|\!)_\alpha u \cap \gamma(v) = \emptyset$.*

PROOF:
This proof is similar to that of Lemma 6.2 on page 105.

- If x is of ground type $\mathbf{exp}\tau$ then:

$$[\![A[x/P]]\!]_\alpha\, u = (\!|A|\!)_\alpha u[\mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle}/(\!|P|\!)_\mathsf{a} u], \tag{6.4}$$

where $\mathsf{q} \in \mathsf{Q}\,[\![\sigma]\!]$, $\mathsf{a} \in \mathsf{A}_\mathsf{q}\,[\![\sigma]\!]$ and $\alpha \in \{tt, ff\}$. We prove by contradiction. If

$$\omega \cdot (\!|A[x/P]|\!)_\alpha u \cap \gamma(v) \neq \emptyset$$

then with equation 6.4,

$$\omega \cdot (\!|A|\!)_\alpha u[\mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle}/(\!|P|\!)_\mathsf{a} u] \cap \gamma(v) \neq \emptyset.$$

Take a string $\omega' \in \omega \cdot (\!|A|\!)_\alpha u[\mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle}/(\!|P|\!)_\mathsf{a} u] \cap \gamma(v)$. It will obviously have the form:

$$\omega \cdots (\!|P|\!)_{\mathsf{a}_1} u \cdots (\!|P|\!)_{\mathsf{a}_2} u \cdots (\!|P|\!)_{\mathsf{a}_k} u \cdots$$

All substrings denoted by $\cdots$ are passive with regard to P (Definition 5.13 on page 96) from the non-interference condition. Therefore from the stability of P, we have that $\mathsf{a}_1 = \mathsf{a}_2 = \cdots = \mathsf{a}_k = \mathsf{a}$, so the string actually has the form:

$$\omega \cdots (\!|P|\!)_\mathsf{a} u \cdots (\!|P|\!)_\mathsf{a} u \cdots (\!|P|\!)_\mathsf{a} u \cdots$$

The strings in $(\!|P|\!)_\mathsf{a} u$ are also passive with regard to any stable variable in the substrings denoted by $\cdots$, so for any $x' \in \mathrm{dom}(v)$,

$$\omega \cdots (\!|P|\!)_\mathsf{a} u \cdots (\!|P|\!)_\mathsf{a} u \cdots (\!|P|\!)_\mathsf{a} u \cdots \in \widetilde{\gamma_\theta^{x'}}$$

if and only if

$$\omega \cdots \omega_0 \cdots \omega_0 \cdots \omega_0 \cdots \in \widetilde{\gamma_\theta^{x'}}$$

for any traces passive with regard to x', $\omega_0 \in \mathcal{P}_{x',u,v}$, according to the Non-interfering String Substitution Lemma (6.3 on page 109). In particular,

$$\omega \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \in \widetilde{\gamma_\theta^{x'}}, \qquad x \neq x'.$$

Since this is for all $x' \in \mathrm{dom}(v), x' \neq x$, it follows that

$$\omega \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \in \bigcap_{x' \in \mathrm{dom}(v), x' \neq x} \widetilde{\gamma_\theta^{x}}.$$

But it can be immediately seen that

$$\omega \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \in \widetilde{\gamma_{\mathbf{exp}\tau}^{x}}.$$

The last two equations taken together imply that

$$\omega \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle} \cdots \in \gamma(v).$$

So, we can "undo" the substitution $(\!|A|\!)_\alpha u[\mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle}/(\!|P|\!)_\mathsf{a} u]$ by replacing all occurrences of $(\!|P|\!)_\mathsf{a} u$ resulting from the substitution back with $\mathsf{q}^{\langle x\rangle}\cdot \mathsf{a}^{\langle x\rangle}$ and obtain A such that: $\omega \cdot \omega'' \in \gamma(v), \omega'' \in (\!|A|\!)_\alpha u \cap \gamma(v)$, which implies $\omega \cdot (\!|A|\!)_\alpha u \cap \gamma(v) \neq \emptyset$, which is a contradiction.

- if x is of ground type $\mathbf{var}\tau$ the same argument is made, only that instead of $\mathsf{a}_1 = \cdots = \mathsf{a}_k$, we have that $\mathsf{q}_1 \cdot \mathsf{a}_1 \cdots \mathsf{q}_k \cdot \mathsf{a}_k \in \gamma^x_{\mathbf{var}\tau}$.

- if x is of ground type **comm** the same argument is made. It is always the case that $\mathsf{a}_1 = \cdots = \mathsf{a}_k = done$, but the non-interference condition is still important to make the "un-doing" of the initial substitution possible in the proof by contradiction.

- if x is of first-order type then we use substitution twice, as in equation 6.2 on page 106, in the proof of Lemma 6.2, and we repeat the same argument.

END OF PROOF.

PROOF: *(of Lemma 6.5 on page 111.)*
By induction on the syntax of S.

($\wedge$): similar to that for Lemma 6.1, considering that $P \#_v S_1 \wedge S_2$ implies $P \#_v S_1$ and $P \#_v S_2$, which is immediate from the definition. Same proof for $\Rightarrow$.

($\forall$): similar to that for Lemma 6.1, considering that if $x \neq x'$, $P \#_v \forall x' : \theta.S$ implies $P \#_v S$, immediately from the definition. If $x = x'$ then $(\forall x : \theta.S)[x/P] = \forall x : \theta.S$.

Same for ∇.

(Hoare triple): Immediate from Lemma 6.6, because the substitution does not make either the pre-condition or the post-condition false if they are true. Similar for stability specifications and equivalence.

END OF PROOF.

For the sake of symmetry we show the rules for eliminating universal quantifiers when the universal quantifier is not in the outermost position:

INFERENCE RULE

$$\forall E \, \frac{\Psi \bullet \forall x : \theta.S}{\Psi \bullet S[x/P]}$$

where x not in Ψ, $\Gamma \vdash P : \theta$ and $P \#_x \Psi \bullet S$ and P is substituted free for x in S.

From the Substitution Lemma, 4.3 on page 74, we know that substitution and **let** are semantically equal. We can, therefore, give alternative formulations for $\forall$ and ∇ elimination (see Figure 6.1 on the next page).

Finally, stable variable assumptions can also be discharged using local variable declarations:

INFERENCE RULES

$$\forall E' \; \frac{\Psi \bullet \forall x : \theta.\{A\}\ M\ \{A'\}}{\Psi \bullet \{A\}\ \textbf{let}\ x\ \textbf{be}\ P\ \textbf{in}\ M\ \{A'\}}\ (\Gamma \vdash P : \theta)$$

$$\forall E'' \; \frac{\Psi \bullet \forall x : \theta.\boldsymbol{\nabla}_{\theta'}(P')}{\Psi \bullet \boldsymbol{\nabla}_{\theta'}(\textbf{let}\ x\ \textbf{be}\ P\ \textbf{in}\ P')}\ (\Gamma \vdash P : \theta)$$

$$\nabla E' \; \frac{\Psi \bullet \nabla x : \theta.\{A\}\ M\ \{A'\} \qquad \Psi \bullet \boldsymbol{\nabla}_{\theta}(P)}{\Psi \bullet \{A\}\ \textbf{let}\ x\ \textbf{be}\ P\ \textbf{in}\ M\ \{A'\}}$$

$$\nabla E'' \; \frac{\Psi \bullet \nabla x : \theta.\boldsymbol{\nabla}_{\theta'}(P') \qquad \Psi \bullet \boldsymbol{\nabla}_{\theta}(P)}{\Psi \bullet \boldsymbol{\nabla}_{\theta'}(\textbf{let}\ x\ \textbf{be}\ P\ \textbf{in}\ P')}$$

The free variables of P are not captured in M or P', $P \#_x \Psi \bullet \{A\}\ M\ \{A'\}$ respectively $P \#_x \Psi \bullet \boldsymbol{\nabla}_{\theta'}(P')$ and x must not occur in Ψ. In addition, x must not be free in A or A' in either $\forall$ or ∇ elimination for Hoare triples.

Figure 6.1: Alternative elimination rules.

INFERENCE RULES

$$\nabla E''' \; \frac{\Psi \bullet \nabla x : \textbf{var}\tau.\{A\}\ M\ \{A'\}}{\Psi \bullet \{A\}\ \textbf{new}\tau\ x\ \textbf{in}\ M\ \{A'\}} \qquad \nabla E^{IV} \; \frac{\Psi \bullet \nabla x : \textbf{var}\tau.\boldsymbol{\nabla}_{\sigma}(M)}{\Psi \bullet \boldsymbol{\nabla}_{\sigma}(\textbf{new}\tau\ x\ \textbf{in}\ M)}$$

Variable x may not be free in A and A'.

PROOF OF SOUNDNESS:
For any model $\mathfrak{M} = \langle u, v\rangle$,

$$\begin{array}{rl} \text{if} & \langle u,v\rangle \models \nabla x : \textbf{var}\tau.\{A\}\ M\ \{A'\} \\ \text{then} & \langle (u \mid x \mapsto \mathcal{K}^x_{\textbf{var}\tau}), (v \mid x \mapsto \gamma^x_{\textbf{var}\tau})\rangle \models \{A\}\ M\ \{A'\}, \\ \therefore & \text{for all } \omega \in \gamma(v \mid x \mapsto \gamma^x_{\textbf{var}\tau}),\ \omega \cdot (\!|A|\!)_{\mathit{ff}} \cap \gamma(v \mid x \mapsto \gamma^x_{\textbf{var}\tau}) = \emptyset \\ & \text{implies } \omega \cdot (\!|M; A'|\!)_{\mathit{ff}}\, u \cap \gamma(v \mid x \mapsto \gamma^x_{\textbf{var}\tau}) = \emptyset. \end{array}$$

Since x is not free in A this is equivalent to

$$\text{for all } \omega \in \eta(v), \omega \cdot (\!|A|\!)_{\mathit{ff}} \cap \gamma(v) = \emptyset \text{ implies } \omega \cdot (\!|M; A'|\!)_{\mathit{ff}}\, u \cap \gamma(v \mid x \mapsto \gamma^x_{\textbf{var}\tau}) = \emptyset$$

$$\therefore\ \omega \cdot (\!|A|\!)_{\mathit{ff}} \cap \gamma(v) = \emptyset \text{ implies } \omega \cdot (\!|M; A'|\!)_{\mathit{ff}}\, u \cap \widetilde{\gamma^x_{\textbf{var}\tau}} \cap \gamma(v) = \emptyset$$

$$\therefore\ \omega \cdot (\!|A|\!)_{\mathit{ff}} \cap \gamma(v) = \emptyset \text{ implies } (\omega \cdot (\!|M; A'|\!)_{\mathit{ff}}\, u \cap \widetilde{\gamma^{x}_{\mathbf{var}\tau}}) \restriction \mathcal{A}^{\overline{x}} \cap \gamma(v) = \emptyset$$
$$\therefore\ \omega \cdot (\!|A|\!)_{\mathit{ff}} \cap \gamma(v) = \emptyset \text{ implies } \omega \cdot (\!|\mathbf{new}\tau\ x\ \mathbf{in}\ M; A'|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset$$

Where $\mathcal{A} = \mathcal{A}[\![\Gamma]\!]$. Also,

$$(\!|\mathbf{new}\tau\ x\ \mathbf{in}\ M; A'|\!)_{\mathit{ff}}\, u = (\!|(\mathbf{new}\tau\ x\ \mathbf{in}\ M); A'|\!)_{\mathit{ff}}\, u,$$

if x not free in A. So it follows that

$$\omega \cdot (\!|A|\!)_{\mathit{ff}} \cap \gamma(v) = \emptyset \text{ implies } \omega \cdot (\!|(\mathbf{new}\tau\ x\ \mathbf{in}\ M); A'|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset$$

which means that $\mathfrak{M} \models \{A\}\ \mathbf{new}\tau\ x\ \mathbf{in}\ M\ \{A'\}$.

Similarly for the stability specification. End Of Proof.

The final specification-level inference rule we show is for equivalence:

Inference Rule

$$\frac{\Psi \bullet S[x/P_1] \qquad \Psi \bullet P_1 \equiv_\theta P_2}{\Psi \bullet S[x/P_2]}$$

Where P_i are substituted freely for x in S.

Proof of Soundness: Induction on the syntax of specifications: For connectives the substitution distributes over the connectives, so we can immediately use the inductive hypothesis. For quantifiers the same, since P_i must be substituted freely. For stability specifications and Hoare triples the proof is immediate, as equal strings are replacing equal strings in all the definitions. End Of Proof.

6.3 Inference rules for stability

From Theorem 5.1 we know that stability specifications, like the other specifications in our language, can be verified by model-checking. But it can be useful, in some situations, to determine the stability of phrases inferentially or by a combination of model-checking and inference. In other words, stability specifications can be themselves model-checked compositionally.

Some useful inference rules are given in Figure 6.2 on the following page. **K** is any language constant and **op** any language operator (all commands are trivially stable, so the non-interference condition is not necessary for the assignment operator). The condition $(*)$ in the NO-Side-eFfects rule is that all free identifiers of P are bound by ∇ and P is not self-interfering, $\Psi \bullet P \# P$.

By $\Psi \bullet P \# P'$ we mean that phrases P and P' do not interfere with each other's stable variables:

INFERENCE RULES

$$\text{CONST}\ \frac{}{\boldsymbol{\nabla}_\sigma(\mathbf{K})}\ (\mathbf{K}:\sigma) \qquad \text{AXIOM}\ \frac{}{\forall x:\theta.\boldsymbol{\nabla}_\theta(x)} \qquad \text{NO-SF}\ \frac{}{\Psi \bullet \boldsymbol{\nabla}_\theta(P)}\ (*)$$

$$\text{VAR}\ \frac{\Psi \bullet \boldsymbol{\nabla}_{\mathbf{var}\tau}(V)}{\Psi \bullet \boldsymbol{\nabla}_{\mathbf{exp}\tau}(!V)} \qquad \text{OP}\ \frac{\Psi \bullet \boldsymbol{\nabla}_{\sigma_1}(M_1) \qquad \Psi \bullet \boldsymbol{\nabla}_{\sigma_2}(M_2)}{\Psi \bullet \boldsymbol{\nabla}_{\sigma'}(M_1\ \mathbf{op}\ M_2)}\ (\Psi \bullet M_1 \# M_2)$$

$$\text{IF}\ \frac{\Psi \bullet \boldsymbol{\nabla}_{\mathbf{expbool}}(B) \qquad \Psi \bullet \boldsymbol{\nabla}_\sigma(M_1) \qquad \Psi \bullet \boldsymbol{\nabla}_\sigma(M_2)}{\Psi \bullet \boldsymbol{\nabla}_\sigma(\mathbf{if}\ B\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)}\ (\Psi \bullet M_1 \# M_2, \Psi \bullet M_i \# B)$$

$$\text{APP}\ \frac{\Psi \bullet \boldsymbol{\nabla}_{\sigma\to\theta}(P) \qquad \Psi \bullet \boldsymbol{\nabla}_\sigma(M)}{\Psi \bullet \boldsymbol{\nabla}_\theta(PM)}\ (\Psi \bullet M \# P) \qquad \text{ABS}\ \frac{\Psi \bullet \forall x:\sigma.\boldsymbol{\nabla}_\theta(P)}{\Psi \bullet \boldsymbol{\nabla}_{\sigma\to\theta}(\lambda x:\sigma.P)}$$

$$\text{COM}\ \frac{}{\boldsymbol{\nabla}_{\mathbf{comm}}(C)} \qquad \text{PROC}\ \frac{}{\boldsymbol{\nabla}_{\overline{\sigma}\to\mathbf{comm}}(P)}$$

Figure 6.2: Some inference rules for stability.

Definition 6.7 (Program non-interference) *We say that two phrases P, P' do not interfere under Ψ, denoted by $\Psi \bullet P \# P'$, if $[\![P]\!](u \diamond \Psi) \#_{v\diamond\Psi} [\![P']\!](u \diamond \Psi)$.*

This definition is virtually identical to the definition of non-interference at x (Definitions 6.5 and 6.6 on page 111) but it is between two phrases rather than between a phrase and a specification.

The reason such inference rules are useful is that for many phrases it is quite obvious whether they are stable, so model-checking the stability would be an unnecessary expense.
PROOF OF SOUNDNESS:

(CONST): For any numerical constant n, for any model $\mathfrak{M} = \langle u, v\rangle$,

$$(\!|n|\!)_m u \cdot \mathcal{P}_{n,u,v} \cdot (\!|n|\!)_{m'} u \cap \gamma(v) = \emptyset$$

for all $m, m' \in \mathcal{A}[\![\mathbf{int}]\!], m \neq m'$ because it follows that it cannot be the case that $n = m$ and $n = m'$ so at least one of $(\!|n|\!)_m u, (\!|n|\!)_{m'} u$ is empty.

Similarly for logical constants, **true** and **false**.

The only other constant is **diverge**, and $(\!|\mathbf{diverge}|\!)_\alpha u = \emptyset$, for all α.

(AXIOM): We need to prove that for $\mathfrak{M}_\emptyset = \langle u_\emptyset, v_\emptyset\rangle$, $\mathfrak{M}_\emptyset \models \forall x : \theta.\boldsymbol{\nabla}_\theta(x)$. This is equivalent to $\langle (x \mapsto \mathcal{K}^x_\theta), (x \mapsto \gamma^x_\theta)\rangle \models \boldsymbol{\nabla}_\theta(x)$. We consider the following cases:

- x is of ground type $\mathbf{exp}\tau$. We need to prove

$$(\!|x|\!)_{\alpha_1} u \cdot \mathcal{P}_{x,u,v} \cdot (\!|x|\!)_{\alpha_2} u \cap \widetilde{\gamma^x_{\mathbf{exp}\tau}} = \emptyset,$$

for all $\alpha_1, \alpha_2 \in \mathcal{A}\llbracket\sigma\rrbracket, \alpha_1 \neq \alpha_2$. This is equivalent to

$$q^{\langle x\rangle} \cdot \alpha_1^{\langle x\rangle} \cdot \mathcal{P}_{x,u,v} \cdot q^{\langle x\rangle} \cdot \alpha_2^{\langle x\rangle} \upharpoonright \mathcal{A}\llbracket x : \mathbf{exp}\tau\rrbracket \cap \sum_{\alpha\in\mathcal{A}\llbracket\tau\rrbracket} (q^{\langle x\rangle} \cdot \alpha^{\langle x\rangle})^* = \emptyset,$$

which is true, since $\alpha_1 \neq \alpha_2$.

- x is of ground type $\mathbf{var}\tau$. We need to prove that

$$(\!|x|\!)_{\mathsf{r};\alpha_1} u \cdot \mathcal{P}_{x,u,v} \cdot (\!|x|\!)_{\mathsf{r}\alpha_2} u \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} = \emptyset,$$

for all $\alpha_1, \alpha_2 \in \mathcal{A}\llbracket\sigma\rrbracket, \alpha_1 \neq \alpha_2$. This is equivalent to

$$\begin{aligned} &\mathit{read}^{\langle x\rangle} \cdot \alpha_1^{\langle x\rangle} \cdot \mathcal{P}_{x,u,v} \cdot \mathit{read}^{\langle x\rangle} \cdot \alpha_2^{\langle x\rangle} \upharpoonright \mathcal{A}\llbracket x : \mathbf{exp}\tau\rrbracket \\ \cap &\sum_{\alpha\in\mathcal{A}\llbracket\tau\rrbracket} (\mathit{read}^{\langle x\rangle} \cdot \alpha^{\langle x\rangle})^* \cdot \sum_{\alpha\in\mathcal{A}\llbracket\tau\rrbracket} \left(\mathit{write}(\alpha)^{\langle x\rangle} \cdot \mathit{ok}^{\langle x\rangle} \cdot (\mathit{read}^{\langle x\rangle} \cdot \alpha^{\langle x\rangle})^*\right)^* = \emptyset, \end{aligned}$$

This is true because $\mathcal{P}_{x,u,v}$ contains no $\mathit{write}(\alpha)^{\langle x\rangle}$ actions and $\alpha_1 \neq \alpha_2$. We also need to prove

$$(\!|x|\!)_{\mathsf{w}\alpha_1} u \cdot \mathcal{P}_{x,u,v} \cdot (\!|x|\!)_{\mathsf{r}\alpha_2} u \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} = \emptyset,$$

for all $\alpha_1, \alpha_2 \in \mathcal{A}\llbracket\sigma\rrbracket, \alpha_1 \neq \alpha_2$, which is similar.

- if x is of first-order type then, by definition, $\mathfrak{M} \models \nabla x : \theta.\boldsymbol{\nabla}_\theta(x)$ if and only if $\mathfrak{M} \models \nabla x_1 : \sigma_1 \ldots \nabla x_k : \sigma_k.\nabla x : \theta.\boldsymbol{\nabla}_\sigma(xx_1 \cdots x_k)$, where $\theta = \sigma_1 \to \cdots \to \sigma_k \to \sigma$.
  - $\sigma = \mathbf{comm}$ is always true.
  - $\sigma = \mathbf{exp}\tau$ then we need to prove that

$$\mathfrak{M} \models \nabla x_1 : \sigma_1 \ldots \nabla x_k : \sigma_k.\nabla x : \overline{\sigma} \to \mathbf{exp}\tau.\boldsymbol{\nabla}_{\mathbf{exp}\tau}(xx_1 \cdots x_k),$$

which is equivalent to

$$(\!|xx_1 \cdots x_k|\!)_\alpha u' \cdot \mathcal{P}_{xx_1\cdots x_k,u',v'} \cdot (\!|xx_1 \cdots x_k|\!)_\alpha u' \cap \gamma(v') = \emptyset, \tag{6.5}$$

where $v' = (v \mid x_i \mapsto \gamma^{x_i}_{\sigma_i} \mid x \mapsto \gamma^x_{\overline{\sigma}\to\mathbf{exp}\tau})$
and $u' = (u \mid x_1 \mapsto \mathcal{K}^{x_1}_{\sigma_1} \mid \cdots \mid x_k \mapsto \mathcal{K}^{x_k}_{\sigma_k} \mid x \mapsto \mathcal{K}^x_{\overline{\sigma}\to\mathbf{exp}\tau})$.
We define the language $\mathcal{K}^x_{\overline{\sigma}\to\mathbf{exp}\tau;\alpha}$ as in the proof of the Semantic Substitution Lemma (6.2, page 106). If $\alpha \neq \alpha'$, 6.5 becomes:

$$\mathcal{K}^x_{\overline{\sigma}\to\mathbf{exp}\tau;\alpha}[\kappa] \cdot \mathcal{P}_{xx_1\cdots x_k,u',v'} \cdot \mathcal{K}^x_{\overline{\sigma}\to\mathbf{exp}\tau;\alpha'}[\kappa] \cap \gamma(v') = \emptyset, \tag{6.6}$$

where $\kappa : \left(\sum_{1\le i\le k}\sum_{\mathsf{q}_i\in\mathsf{Q}\llbracket\sigma_i\rrbracket} \mathsf{q}_i \cdot \mathsf{A}_{\mathsf{q}_i}\llbracket\sigma_i\rrbracket\right)^{\langle i\rangle} \to \left(\sum_{1\le i\le k}\sum_{\mathsf{q}_i\in\mathsf{Q}\llbracket\sigma_i\rrbracket} \mathsf{q}_i \cdot \mathsf{A}_{\mathsf{q}_i}\llbracket\sigma_i\rrbracket\right)^{\langle x_i\rangle}$ such that $\kappa\left(\mathsf{q}^{\langle i\rangle} \cdot \mathsf{a}^{\langle i\rangle}\right) = \mathsf{q}^{\langle x_i\rangle} \cdot \mathsf{a}^{\langle x_i\rangle}$, with $\theta = \overline{\sigma} \to \sigma$.

We use the definition of expression-like function stability (Definition 5.3 on page 84). Consider a string in the language intersected with $\gamma(v')$ in equation 6.6. If it has a substring of the form

$$q^{\langle x\rangle} \cdot \omega_1 \cdot \alpha_1^{\langle x\rangle} \cdot \omega' \cdot q^{\langle x\rangle} \cdot \omega_2 \cdot \alpha_2^{\langle x\rangle}$$

then $\omega_1 \cdot \omega_2 \in \bigcap_{1\le i\le k} \widetilde{\gamma_{\sigma_i}^{xi}}$ because following the substitutions in equation 6.6 any occurrence of a $\mathsf{q}_i^{\langle xi\rangle}$ and $\mathsf{a}_i^{\langle xi\rangle}$ is in a string of the form $\mathsf{q}_i^{\langle xi\rangle} \cdot \mathsf{q}_i^{\langle x_i\rangle} \cdot \mathsf{a}_i^{\langle x_i\rangle} \cdot \mathsf{a}_i^{\langle xi\rangle}$ and all arguments $\mathsf{q}_i^{\langle x_i\rangle} \cdot \mathsf{a}_i^{\langle x_i\rangle}$ occur in a stable manner due to the intersection with $\widetilde{\gamma_{\sigma_i}^{x_i}}$, part of $\gamma(v')$, in equation 6.6. From the definition of stability for $x : \overline{\sigma} \to \mathbf{exp}\tau$, it follows that if $\alpha \neq \alpha'$ then the intersection in equation 6.6 is empty.

– $\sigma = \mathbf{var}\tau$ has a similar argument.

(NO-SF): The proof is by induction on the syntax of P. The base cases are covered by the CONST and AXIOM rules.

The cases of dereferencing, operators, branching, abstraction and application are covered by the rules VAR, OP, IF, ABS and APP using the fact that sub-terms cannot interfere since, by induction hypothesis, they do not assign to the stable variables. The cases of assignment and iteration are trivial because commands are always stable.

The remaining case is that of **new**:

$$\mathfrak{M}_\Gamma \models \Psi \bullet \nabla_\sigma(\mathbf{new}\tau\ x\ \mathbf{in}\ P)$$

for all $\mathfrak{M}_\Gamma$ if $\mathbf{new}\tau\ x\ \mathbf{in}\ P$ does not assign to any free variable in P bound by ∇ in Ψ. The case $\sigma = \mathbf{comm}$ is trivial, because all commands are stable. If $\sigma = \mathbf{exp}\tau'$ then we need to prove that

$$(\!|\mathbf{new}\tau\ x\ \mathbf{in}\ P|\!)_\alpha u \cdot \mathcal{P}_{\mathbf{new}\tau\ x\ \mathbf{in}\ P,u,v} \cdot (\!|\mathbf{new}\tau\ x\ \mathbf{in}\ P|\!)_{\alpha'} u \cap \gamma(v) = \emptyset,$$

for all $\alpha, \alpha' \in \mathcal{A}[\![\tau]\!], \alpha \neq \alpha'$, i.e.

$$\begin{aligned}\big((\!|P|\!)_\alpha(u \mid x \mapsto \mathcal{K}^x_{\mathbf{var}\tau}) \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} \restriction \mathcal{A}^{\overline{x}}\big) \\ \cdot \mathcal{P}_{\mathbf{new}\tau\ x\ \mathbf{in}\ P,u,v} \cdot \big((\!|P|\!)_{\alpha'}(u \mid x \mapsto \mathcal{K}^x_{\mathbf{var}\tau}) \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} \restriction \mathcal{A}^{\overline{x}}\big) \cap \gamma(v) = \emptyset.\end{aligned}$$

We cannot apply the induction hypothesis here, because P is not necessarily stable (it may assign to x), so we need to use induction on the syntax of P. The induction is laborious but routine. The interesting case is the base case $P = !x_0$.

We need to show that:

$$\begin{aligned}\big((\!|!x_0|\!)_\alpha(u \mid x \mapsto \mathcal{K}^x_{\mathbf{var}\tau}) \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} \restriction \mathcal{A}^{\overline{x}}\big) \\ \cdot \mathcal{P}_{\mathbf{new}\tau\ x\ \mathbf{in}\ !x_0,u,v} \cdot \big((\!|!x_0|\!)_{\alpha'}(u \mid x \mapsto \mathcal{K}^x_{\mathbf{var}\tau}) \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} \restriction \mathcal{A}^{\overline{x}}\big) \cap \gamma(v) = \emptyset.\end{aligned}$$

There are two sub-cases, depending whether x_0 is the local variable.

- $(x_0 \neq x)$: The task reduces to showing that

$$\mathit{read}^{\langle x_0 \rangle} \cdot \alpha^{\langle x_0 \rangle} \cdot \mathcal{P}_{\mathbf{new}\tau\ x\ \mathbf{in}\ !x_0,u,v} \cdot \mathit{read}^{\langle x_0 \rangle} \cdot (\alpha')^{\langle x_0 \rangle} \cap \gamma(v) = \emptyset,$$

  which is true from the stability of x_0.

- $(x_0 = x)$: In this case we need to show that

$$(\mathit{read}^{\langle x \rangle} \cdot \alpha^{\langle x \rangle} \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} \restriction \mathcal{A}^{\overline{x}}) \cdot \mathcal{P}_{\mathbf{new}\tau\ x\ \mathbf{in}\ !x,u,v} \cdot (\mathit{read}^{\langle x \rangle} \cdot (\alpha')^{\langle x \rangle} \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} \restriction \mathcal{A}^{\overline{x}}) \cap \gamma(v) = \emptyset,$$

  which is true because either $\alpha \neq \alpha_\tau$ or $\alpha' \neq \alpha_\tau$, (the initial value of the local variable), so either $\mathit{read}^{\langle x \rangle} \cdot \alpha^{\langle x \rangle} \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} = \emptyset$ or $\mathit{read}^{\langle x \rangle} \cdot (\alpha')^{\langle x \rangle} \cap \widetilde{\gamma^x_{\mathbf{var}\tau}} = \emptyset$.

(VAR): We need to show that $(\!| !V |\!)_\alpha u \cdot \mathcal{P}_{!V,u,v} \cdot (\!| !V |\!)_{\alpha'} \cap \gamma(v) = \emptyset$, which is equivalent to $(\!| V |\!)_{\mathsf{r}\alpha} u \cdot \mathcal{P}_{V,u,v} \cdot (\!| V |\!)_{\mathsf{r}\alpha'} \cap \gamma(v) = \emptyset$, which follows from the stability of variable V.

(OP): We need to show that

$$(\!| M_1\ \mathbf{op}\ M_2 |\!)_\alpha u \cdot \mathcal{P}_{M_1\ \mathbf{op}\ M_2,u,v} \cdot (\!| M_1\ \mathbf{op}\ M_2 |\!)_{\alpha'} \cap \gamma(v) = \emptyset$$

equivalent to $(\!| M_1 |\!)_{\alpha_1} u \cdot (\!| M_2 |\!)_{\alpha_2} u \cdot \mathcal{P}_{M_1\ \mathbf{op}\ M_2,u,v} \cdot (\!| M_1 |\!)_{\alpha'_1} u \cdot (\!| M_2 |\!)_{\alpha'_2} u \cap \gamma(v) = \emptyset$

for all $\alpha_1, \alpha_2, \alpha'_1, \alpha'_2$ such that $\alpha_1\ \mathit{op}\ \alpha_2 \neq \alpha'_1\ \mathit{op}\ \alpha'_2$, where op is the obvious interpretation of **op**, which requires either $\alpha_1 \neq \alpha'_1$ or $\alpha_2 \neq \alpha'_2$. But if $\alpha_1 \neq \alpha'_1$ then

$$(\!| M_1 |\!)_{\alpha_1} u \cdot (\!| M_2 |\!)_{\alpha_2} u \cdot \mathcal{P}_{M_1\ \mathbf{op}\ M_2,u,v} \cdot (\!| M_1 |\!)_{\alpha'_1} \cap \gamma(v) = \emptyset,$$

from the stability of M_1 and the fact that M_1 does not interfere with M_2, so that $(\!| M_2 |\!)_{\alpha_2} u \subseteq \mathcal{P}_{M_1\ \mathbf{op}\ M_2,u,v}$. Similarly if $\alpha_2 \neq \alpha'_2$.

(IF): Similar to the above.

(APP): This inference rule can be derived, using the definition $\mathbf{\nabla}_{\sigma\to\theta}(P) \stackrel{\text{def}}{=} \forall x : \sigma.\mathbf{\nabla}_\theta(Px)$, $x \notin \mathit{Free}(P)$:

$$\cfrac{\cfrac{\Psi \bullet \forall x : \sigma.\mathbf{\nabla}_\theta(Px) \qquad \Psi \bullet \mathbf{\nabla}_\sigma(M)}{\Psi \bullet \mathbf{\nabla}_\theta(Px)[x/M]}}{\Psi \bullet \mathbf{\nabla}_\theta(PM)}$$

because $\Psi \bullet M \# P$ implies $M \#_x \Psi \bullet \forall x : \sigma.\mathbf{\nabla}_\theta(Px)$, from definition.

(ABS): Immediately from definition of first-order stability.

END OF PROOF.

For the stability of **new** it is important that the local variables are initialized with a default value. This property would fail if the initial value were assigned non-deterministically, because the following expression would not be stable: $\mathbf{new}\tau\ v\ \mathbf{in}\ !v$.

This set of inference rules for stability is obviously incomplete. It is easy to see that phrases such as $0 \times E$ are stable, whatever expression E stands for. Also, rather strange phrases such as **if** B **then** 1 **else diverge** are also obviously stable, regardless of what boolean expression B stands for. The phrase is stable only because one of the branches consists of the empty set of traces. In the context of *total* correctness this phrase would create serious technical problems.

But giving better coverage of the semantics of stability using logical rules can only come at the expense of increased complexity of the rules. Whenever absolute precision in the determination of stability is required one can resort to model checking; the point of the logical rules is to give an easy way to handle the cases when stability is rather obvious, without resorting to potentially computationally expensive verifications of the semantical model.

6.4 Inference rules for assertions

Inference rules for assertions relate connectives and quantifiers occurring in assertions with specification level connectives. They all deal with "static" reasoning, i.e. assertions which are true in all states. These assertions are called *mathematical facts*.

The following two rules are similar to rules in Hoare logic (J stands for "join").

INFERENCE RULES

$$\textbf{and}\text{J}\ \frac{\Psi \bullet \{A_1\} \qquad \Psi \bullet \{A_2\}}{\Psi \bullet \{A_1 \textbf{ and } A_2\}} \qquad \textbf{or}\text{J1}\ \frac{\Psi \bullet \{A_1\}}{\Psi \bullet \{A_1 \textbf{ or } A_2\}} \qquad \textbf{or}\text{J2}\ \frac{\Psi \bullet \{A_2\}}{\Psi \bullet \{A_1 \textbf{ or } A_2\}}$$

PROOF OF SOUNDNESS:

- (**and**–join): For any model $\langle u, v\rangle$ consistent with Ψ we need to prove that

  $$\text{for all } \omega \in \eta(v), \quad \omega \cdot (\!|A_1 \textbf{ and } A_2|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset.$$

  Equivalently,

  $$\omega \cdot (\!|A_1|\!)_{\mathit{tt}} u \cdot (\!|A_2|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset \text{ and} \tag{6.7}$$
  $$\omega \cdot (\!|A_1|\!)_{\mathit{ff}}\, u \cdot (\!|A_2|\!)_{\mathit{tt}} u \cap \gamma(v) = \emptyset \text{ and} \tag{6.8}$$
  $$\omega \cdot (\!|A_1|\!)_{\mathit{ff}}\, u \cdot (\!|A_2|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset. \tag{6.9}$$

  But $(\!|A_i|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset$, from the truth of assertion A_i, with $i = 1, 2$.

- (**or**J1 and J2): Similar proof.

END OF PROOF.

The "splitting" counterpart of **and**J, however, is not sound:

$$\textbf{and}\text{-Split}\ \frac{\Psi \bullet \{A_1 \textbf{ and } A_2\}}{\Psi \bullet \{A_1\} \wedge \Psi \bullet \{A_2\}}$$

for reasons of possible interference or possible divergence. For example, the following inference is unsound:

$$\frac{\{\textbf{diverge and false}\}}{\{\textbf{diverge}\} \wedge \{\textbf{false}\}}$$

The premise is true because **diverge and false** $\equiv$ **diverge**, and **diverge** is true in a partial-correctness interpretation. But the conclusion is clearly false. The required side-conditions to make the rule valid is that assertions A_1 and A_2 terminate normally and do not interfere given Ψ, denoted by $\Psi \bullet \delta(A_1 \textbf{ and } A_2)$ and $\Psi \bullet A_1 \# A_2$.

Definition 6.8 (Normal termination) *A term $\Gamma \vdash P : \theta$ is said to terminate normally under stability constraints Ψ, denoted by $\Psi \bullet \delta(P)$, if for all $\omega \in \eta(v \diamond \Psi)$,*

$$\omega \cdot [\![P]\!]\,(u \diamond \Psi) \cap \gamma(v \diamond \Psi) \neq \emptyset.$$

INFERENCE RULE

$$\textbf{and}\text{-Split}\ \frac{\Psi \bullet \{A_1 \textbf{ and } A_2\}}{\Psi \bullet \{A_1\} \wedge \Psi \bullet \{A_2\}}$$

$\Psi \bullet \delta(A_1 \textbf{ and } A_2)$ and $\Psi \bullet A_1 \# A_2$.

PROOF OF SOUNDNESS: For any model $\langle u, v\rangle$ consistent with Ψ, and for all $\omega \in \eta(v)$,

$$\begin{aligned}
& \omega \cdot (\!|A_1 \textbf{ and } A_2|\!)_{f\!f}\, u \cap \gamma(v) = \emptyset, && (6.10)\\
\text{equivalently,}\quad & \omega \cdot (\!|A_1|\!)_{tt} u \cdot (\!|A_2|\!)_{f\!f}\, u \cap \gamma(v) = \emptyset \text{ and} && (6.11)\\
& \omega \cdot (\!|A_1|\!)_{f\!f}\, u \cdot (\!|A_2|\!)_{tt} u \cap \gamma(v) = \emptyset \text{ and} && (6.12)\\
& \omega \cdot (\!|A_1|\!)_{f\!f}\, u \cdot (\!|A_2|\!)_{f\!f}\, u \cap \gamma(v) = \emptyset \text{ and}
\end{aligned}$$

From equation 6.10 and the assumption that A_1 **and** A_2 terminates normally, $\Psi \bullet \delta(A_1 \textbf{ and } A_2)$ it follows also that

$$\begin{aligned}
& \omega \cdot (\!|A_1 \textbf{ and } A_2|\!)_{tt} u \cap \gamma(v) \neq \emptyset,\\
\text{equivalently,}\quad & \omega \cdot (\!|A_1|\!)_{tt} u \cdot (\!|A_2|\!)_{tt} u \cap \gamma(v) \neq \emptyset && (6.13)
\end{aligned}$$

In equation 6.13, from the non-interference condition $\Psi \bullet A_1 \# A_2$, using the non-interference substitution lemma (6.3 on page 109) it follows that

$$\omega \cdot (\!|A_1|\!)_{tt} u \cap \gamma(v) \neq \emptyset \quad\text{and}\quad \omega \cdot (\!|A_2|\!)_{tt} u \cap \gamma(v) \neq \emptyset. \tag{6.14}$$

From equations 6.11, 6.14 and the non-interference condition it follows that, according to Lemma 6.3, $\omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$. From equations 6.12, 6.14 and the non-interference condition it follows that $\omega \cdot (\!|A_2|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$. END OF PROOF.

The rule for implication is:

INFERENCE RULE

$$\textbf{imply}\ \frac{\Psi \bullet \{A_1\} \qquad \Psi \bullet \{A_1\ \textbf{implies}\ A_2\}}{\Psi \bullet \{A_1\ \textbf{and}\ A_2\}}$$

The proof of soundness is similar. Using the rule above we can also derive the more usual implication rule, where the premise A_1 is canceled using **and**S. But the rule above has the advantage that it requires no side-conditions so, in some respect, it is more basic.

The assertion language may contain assignments, which seems to clash with the idea of static reasoning. However, the following two axiom schemata give adequate support for static reasoning even in the presence of side effects:

AXIOM

$$\text{MATH}\ \frac{}{\Psi \bullet \{A\}}\ \text{if } A \text{ is a mathematical fact}$$

The rule MATH says that any arithmetical or logical fact expressed as an assertion A can be introduced as an axiom in the specification language provided that all its free identifiers are bound by stability quantifiers in Ψ. Since A is a mathematical fact it is implied that it only uses expression-identifiers and arithmetical and logical operators.

INFERENCE RULE

$$\text{EQ}\ \frac{\Psi \bullet \{E_1 = E_2\} \quad \Psi \bullet \boldsymbol{\nabla}_{\mathbf{expint}}(E_1) \quad \Psi \bullet \boldsymbol{\nabla}_{\mathbf{expint}}(E_2) \quad \Psi \bullet \boldsymbol{\nabla}_{\mathbf{expint}\to\mathbf{expint}}(F)}{\Psi \bullet \{F(E_1) = F(E_2)\}}$$

with side-condition $\Psi \bullet E_1 \mathrel{\#} E_2$ and $\Psi \bullet E_i \mathrel{\#} F$ for $i \in \{1, 2\}$.

Note that in this rule expressions E_i may have side effects. Using these inference rules, along with substitution, we can statically reason about expressions with side effects.

The EQuality rule is especially important, because it compensates for the fact that in the presence of side effects equality of expressions does not entail semantic equivalence, illustrated by the failure of Proposition 2.2 on page 27. This rule shows that despite possible semantic inequivalence, equal expressions may be substituted for equal expressions preserving equality, if the right non-interference conditions are met. For assertions, the equality rule is:

INFERENCE RULE

$$\text{EQ'}\ \frac{\Psi \bullet \{E_1 = E_2\} \quad \Psi \bullet \nabla_{\mathbf{expint}}(E_1) \quad \Psi \bullet \nabla_{\mathbf{expint}}(E_2) \quad \Psi \bullet \nabla_{\mathbf{expint}\to\mathbf{assert}}(F) \quad \Psi \bullet \{F(E_1)\}}{\Psi \bullet \{F(E_2)\}}$$

with side-condition $\Psi \bullet E_1 \# E_2$, and $\Psi \bullet \delta\big(F(E_1)\big)$, $\Psi \bullet \delta(E_1 = E_2)$, $\Psi \bullet E_i \# F$ for $i \in \{1, 2\}$.

The generalization to an arbitrary number of arguments is straightforward.

Below there is a simple example of static reasoning with active expressions.

Example 6.1

$$\nabla E\ \frac{MATH\ \dfrac{}{\nabla y : \mathbf{expint}.\nabla x : \mathbf{expint}.\{x + x = 2 \times x\}} \qquad OP\ \dfrac{COMM\ \dfrac{}{\nabla y : \mathbf{expint}.\nabla_{\mathbf{comm}}(v := !v + 1)} \qquad AXIOM\ \dfrac{}{\nabla y : \mathbf{expint}.\nabla_{\mathbf{expint}}(y)}}{\nabla y : \mathbf{expint}.\nabla_{\mathbf{expint}}(v := !v + 1; y)}}{\nabla y : \mathbf{expint}.\{(v := !v + 1; y) + (v := !v + 1; y) = 2 \times (v := !v + 1; y)\}}$$

PROOF OF SOUNDNESS:

(MATH): We will not formally prove this axiom. It would be very laborious to prove all laws of arithmetic and logic, but it is rather trivial for every particular instance; example 5.1 on page 89 provides some evidence of that. Informally, the reason every individual such equation can be proved is that the interpretation of the arithmetic-logical operators is the standard one and that in the presence of stability constraints and the absence of computational effects (assignments and divergence) expression-denoting identifiers keep their value.

Note that what exactly is an arithmetical fact also depends on the particular implementation of the arithmetic operators, as discussed on page 59.

(EQ): For any model $\langle u, v\rangle$ consistent with Ψ we need to prove that

$$\omega \cdot (\!| F E_1 |\!)_{n_1} u \cdot (\!| F E_2 |\!)_{n_2} u \cap \gamma(v) = \emptyset$$

for all $\omega \in \eta(v)$, $n_1, n_2 \in \mathcal{N}$, $n_1 \neq n_2$. This is equivalent to

$$\omega \cdot (\!| F |\!)_{n_1} u[q^{\langle 1\rangle} \cdot m_1^{\langle 1\rangle} / (\!| E_1 |\!)_{m_1} u] \cdot (\!| F |\!)_{n_2} u[q^{\langle 1\rangle} \cdot m_2^{\langle 1\rangle} / (\!| E_2 |\!)_{m_2} u] \cap \gamma(v) = \emptyset, \quad (6.15)$$

for all $m_1, m_2 \in \mathcal{N}$.

If F is non-strict (does not use its argument) then this follows immediately from the stability of F. If F is strict and $m_1 \neq m_2$ then, from the assumption that $E_1 = E_2$ in $\langle u, v\rangle$, it follows that, for any traces ω', ω'' that do not interfere with E_i:

$$\omega' \cdot (\!|E_1|\!)_{m_1} \cdot \omega'' \cdot (\!|E_2|\!)_{m_2} \cap \gamma(v) = \emptyset,$$

because of the non-interference condition $\Psi \bullet F \# E_i$. This means that equation 6.15 is true, because every string contains occurrences of E_i, and the rest of the strings do not interfere.

If $m_1 = m_2 = m$, then we prove it by contradiction, similarly to Lemma 6.6 on page 111. If there exists a string in the language of the left-hand-side of equation 6.15 then, because of the non-interference condition and the non-interference string substitution lemma (Lemma 6.3 on page 109), the substrings contributed by $(\!|E_i|\!)_{m_i} u$ can be removed and still obtain a string in $\gamma(v)$.

But in that case we have

$$\omega \cdot (\!|F|\!)_{n_1} u[q^{\langle 1\rangle} \cdot m^{\langle 1\rangle}/\epsilon] \cdot (\!|F|\!)_{n_2} u[q^{\langle 1\rangle} \cdot m^{\langle 1\rangle}/\epsilon] \cap \gamma(v) \neq \emptyset$$

which is equivalent to $\omega \cdot (\!|F(m)|\!)_{n_1} u \cdot (\!|F(m)|\!)_{n_2} \cap \gamma(v) \neq \emptyset.$

which is a contradiction to the stability of F, because applying a stable function F to a constant cannot yield different results, by definition of stability (Definition 5.3 on page 84).

(EQ'): For any model $\langle u, v\rangle$ consistent with Ψ, from the normal termination condition $\Psi \bullet \delta\big(F(E_1)\big)$ and the truth of the assertion $\Psi \bullet \{F(E_1)\}$ it follows that:

$$\omega \cdot (\!|F|\!)_{tt} u[q^{\langle 1\rangle} \cdot n^{\langle 1\rangle}/(\!|E_1|\!)_n u] \cap \gamma(v) \neq \emptyset. \tag{6.16}$$

If function F is non-strict the conclusion follows immediately from the stability of F in the premise.

If function F is strict then, from the non-interference condition it must be the case that $\omega \cdot (\!|E_1|\!)_n u \in \gamma(v)$. The equality $\Psi \bullet E_1 = E_2$, the non-interference $\Psi \bullet E_1 \# E_2$ and the normal termination $\Psi \bullet \delta(E_1 = E_2)$ together imply that it must also be the case that $\omega \cdot (\!|E_2|\!)_n u \in \gamma(v)$.

We can exchange the non-interfering, non-empty languages $(\!|E_1|\!)_n u$ and $(\!|E_2|\!)_n u$ in equation 6.16 and still obtain a non-empty language, $\omega \cdot (\!|F|\!)_{tt} u[q^{\langle 1\rangle} \cdot n^{\langle 1\rangle}/(\!|E_2|\!)_n u] \cap \gamma(v)$ which, together with the premise $\Psi \bullet F(E_1)$ implies the conclusion

$$\omega \cdot (\!|F(E_2)|\!)_{ff} u \cap \gamma(v) = \emptyset.$$

END OF PROOF.

INFERENCE RULES

$$\text{SP}\ \frac{\Psi \bullet \{A_1\} \Rightarrow \{A_2\} \qquad \Psi \bullet \{A_2\}\ M\ \{A_3\}}{\Psi \bullet \{A_1\}\ M\ \{A_3\}}$$

$$\text{WC}\ \frac{\Psi \bullet \{A_1\}\ M\ \{A_2\} \qquad \Psi \bullet \{A_2\} \Rightarrow \{A_3\}}{\Psi \bullet \{A_1\}\ M\ \{A_3\}}$$

$$\text{SQ}\ \frac{\Psi \bullet \{A_1\}\ M_1\ \{A_2\} \qquad \Psi \bullet \{A_2\}\ M_2\ \{A_3\}}{\Psi \bullet \{A_1\}\ M_1\ \mathbf{op}\ M_2\ \{A_3\}}$$

Figure 6.3: Inference rules for composition.

6.5 Inference rules for programs

We begin, in Figure 6.3, with rules for composition: strengthening precedent, weakening consequent and sequential composition. These rules are similar to the standard Hoare-logic rules, except for the presence of the quantifiers. It is perhaps a little surprising that no side-conditions are required. From the previous section we have seen that when assertions are discharged we must be careful to deal with the possibility that the assertion was true because of divergence, which sometimes makes its discharge unsound, as is illustrated by the rule **and**–SPLIT on page 121.

PROOF OF SOUNDNESS:

(Strengthening Precedent): immediate from semantic definitions.

(Weakening Consequent): For any model $\langle u, v\rangle$ consistent with Ψ, the Hoare triple in the rule's premise is true if:

$$\text{for all } \omega \in \eta(v),\ \ \omega \cdot (\!|A_1|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset$$

implies that

$$\omega \cdot (\!|M; A_2|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset,$$

$$\text{which is equivalent to}\ \ \omega \cdot \left(\sum_{\mathsf{a} \in \mathcal{A}[\![\sigma]\!]} (\!|M|\!)_{\mathsf{a}}\, u \right) \cdot (\!|A_2|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset.$$

But using the second premise it follows that

$$\text{for all } \omega \in \eta(v),\ \ \omega \cdot \big(\sum_{\mathsf{a} \in \mathcal{A}[\![\sigma]\!]} (\!|M|\!)_{\mathsf{a}}\, u \big) \cdot (\!|A_3|\!)_{\mathit{ff}}\, u \cap \gamma(v) = \emptyset,$$

taking the state $\omega' = \omega \cdot (\!|M|\!)_{\mathsf{a}} u \cap \gamma(v)$, which is ω after the execution of M. The case when $\omega' = \emptyset$ must be dealt with separately, because $\emptyset$ is not a valid state. But in this case the conclusion follows trivially.

This is equivalent to for all $\omega \in \eta(v)$, $\omega \cdot (\!|M; A_3|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$, hence the Hoare triple in the conclusion is true.

(Sequencing): similar to WC.

END OF PROOF.

The assignment axiom is exactly the kind of low-level axiom the use of which model checking avoids. However, for the sake of completeness we will give an assignment axiom for our specification language:

INFERENCE RULE

$$\text{AS}\ \frac{\Psi \bullet \boldsymbol{\nabla}_{\mathbf{exp}\,\mathcal{T}}(E) \quad \Psi \bullet \boldsymbol{\nabla}_{\mathbf{exp}\,\mathcal{T}}(V) \quad \Psi \bullet \boldsymbol{\nabla}_{\mathbf{exp}\,\mathcal{T}\rightarrow\mathbf{assert}}(F)}{\Psi \bullet \{F(E)\}\ V := E\ \{F(!V)\}} \quad \begin{array}{l}\Psi \bullet F \# E,\\ \Psi \bullet F \# V,\\ \Psi \bullet V \# E\end{array}$$

PROOF OF SOUNDNESS:

For any model $\langle u, v\rangle$ consistent with Ψ we have to prove that for all $\omega \in \eta(v)$ and for all $\alpha \in \mathcal{A}\,[\![\tau]\!]$, $\omega \cdot (\!|F(E)|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$ implies $\omega \cdot (\!|E|\!)_{\alpha} u \cdot (\!|V|\!)_{\mathsf{w}\alpha} u \cdot (\!|F(!V)|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$.

We consider the following cases:

1. function F is non-strict, immediate.
2. function F is strict and $\omega \cdot (\!|E|\!)_{\alpha} u \cap \gamma(v) = \emptyset$, i.e. expression E is not defined for value α (possibly for non-termination but not necessarily) in state ω, again, immediate.
3. function F is strict and $\omega \cdot (\!|E|\!)_{\alpha} u \cap \gamma(v) \neq \emptyset$, i.e. expression E is defined in state ω. We need to prove that

$$\omega \cdot (\!|E|\!)_{\alpha} u \cdot (\!|V|\!)_{\mathsf{w}\alpha} u \cdot (\!|F|\!)_{\mathit{ff}} u[q^{\langle 1\rangle} \cdot \alpha_0^{\langle 1\rangle} / (\!|V|\!)_{\mathsf{r}\alpha_0}] \cap \gamma(v) = \emptyset.$$

The stability of V and the non-interference conditions imply that the only interesting case is $\alpha = \alpha_0$. We need to prove

$$\omega \cdot (\!|E|\!)_{\alpha} u \cdot (\!|V|\!)_{\mathsf{w}\alpha} u \cdot (\!|F|\!)_{\mathit{ff}} u[q^{\langle 1\rangle} \cdot \alpha^{\langle 1\rangle} / (\!|V|\!)_{\mathsf{r}\alpha}] \cap \gamma(v) = \emptyset.$$

Again, we need to consider two cases. If $(\!|V|\!)_{\mathsf{w}\alpha} u \cap \gamma(v) = \emptyset$, this follows immediately.

Otherwise, we can substitute non-interfering non-empty languages for non-interfering non-empty languages (Lemma 6.3 on page 109), so this is equivalent to:

$$\omega \cdot (\!|E|\!)_\alpha u \cdot (\!|V|\!)_{\mathsf{w}\alpha} u \cdot (\!|F|\!)_{\mathit{ff}} u[q^{\langle 1\rangle} \cdot \alpha^{\langle 1\rangle} / (\!|E|\!)_\alpha] \cap \gamma(v) = \emptyset,$$

which follows from the assumption that $\omega \cdot (\!|F(E)|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$, the stability of F and E and the non-interference of E and V with F.

End Of Proof.

The other two imperative constructs are branching and iteration. But before giving rules for reasoning about them we will consider the following lemma which formalizes the idea that stable expressions may have at most one value in any state:

Lemma 6.7 (Determinacy) *If expression $\Gamma \vdash E : \mathbf{exp}\tau$ is stable in frame v then for all $\omega \in \eta(v)$, for all $\alpha_1, \alpha_2 \in \mathcal{A}[\![\tau]\!]$, and for any environment u, $\omega \cdot (\!|E|\!)_{\alpha_1} u \cap \gamma(v) \neq \emptyset$ and $\omega \cdot (\!|E|\!)_{\alpha_2} u \cap \gamma(v) \neq \emptyset$, implies $\alpha_1 = \alpha_2$.*

Proof:
For any model $\langle u, v\rangle$ consistent with Ψ, if expression E is stable then, if $\alpha_1 \neq \alpha_2$, $(\!|E|\!)_{\alpha_1} u \cdot (\!|E|\!)_{\alpha_2} u \cap \gamma(v) = \emptyset$, so also for all $\omega \in \eta(v)$,

$$\omega \cdot (\!|E|\!)_{\alpha_1} u \cdot (\!|E|\!)_{\alpha_2} u \cap \gamma(v) = \emptyset.$$

Therefore, there must be some $x \in \mathrm{dom}(v)$ such that

$$\omega \cdot (\!|E|\!)_{\alpha_1} u \cdot (\!|E|\!)_{\alpha_2} u \cap \widetilde{v(x)} = \emptyset.$$

This is only possible if $v(x) = \gamma_\theta^x$. Removing the symbols not tagged by x this implies that

$$\left(\omega \cdot (\!|E|\!)_{\alpha_1} u \cdot (\!|E|\!)_{\alpha_2} u \restriction \mathcal{A}^{\overline{x}}\right) \cap \gamma_\theta^x = \emptyset. \tag{6.17}$$

We consider the following cases depending on θ:

- $\theta = \mathbf{exp}\tau$: Equation 6.17 becomes:

$$\underbrace{(q^{\langle x\rangle} \cdot \alpha^{\langle x\rangle})^+}_{\omega} \cdot \underbrace{(q^{\langle x\rangle} \cdot \alpha_{\alpha_1}^{\langle x\rangle})^*}_{(\!|E|\!)_{\alpha_1} u} \cdot \underbrace{(q^{\langle x\rangle} \cdot \alpha_{\alpha_2}^{\langle x\rangle})^*}_{(\!|E|\!)_{\alpha_2} u} \cap \gamma_{\mathbf{exp}\tau}^x = \emptyset.$$

  This is possible only if $\alpha \neq \alpha_{\alpha_1}$ or $\alpha \neq \alpha_{\alpha_2}$. But if $\alpha \neq \alpha_{\alpha_i}$ then $\omega \cdot (\!|E|\!)_{\alpha_i} u \cap \gamma(v) = \emptyset$, which contradicts the hypothesis.

- $\theta = \mathbf{var}\tau$: similar, with the observation that when restricted to $\mathcal{A}^{\overline{x}}$, $(\!|E|\!)_{\alpha_i} u$ must have a $\mathit{read}^{\langle x\rangle}$ as the first move, otherwise the intersection with $\gamma_{\mathbf{exp}\tau}^x$ cannot be empty.

INFERENCE RULES

$$\text{IF}\ \frac{\Psi \bullet \{A_1 \textbf{ and } B\}\ M_1\ \{A_2\} \quad \Psi \bullet \{A_1 \textbf{ and not } B\}\ M_2\ \{A_2\} \quad \Psi \bullet \boldsymbol{\nabla}_{\mathbf{expbool}}(B)}{\Psi \bullet \{A_1\} \textbf{ if } B \textbf{ then } M_1 \textbf{ else } M_2\ \{A_2\}}$$

$$\text{DO}\ \frac{\Psi \bullet \{A \textbf{ and } B\}\ C\ \{A\} \quad \Psi \bullet \boldsymbol{\nabla}_{\mathbf{expbool}}(B)}{\{A\} \textbf{ while } B \textbf{ do } C\ \{A \textbf{ and not } B\}}$$

IF non-interference conditions $\Psi \bullet B \# M_1, \Psi \bullet B \# M_2, \Psi \bullet B \# A_2, \Psi \bullet A_1 \# B$
DO non-interference conditions $\Psi \bullet A \# B, \Psi \bullet B \# C$.

Figure 6.4: Inference rules for branching and iteration.

- $\theta = \mathbf{comm}$: not a valid case.
- $\theta = \overline{\sigma} \to \mathbf{exp}\tau$ similar to $\mathbf{exp}\tau$.
- $\theta = \overline{\sigma} \to \mathbf{var}\tau$ similar to $\mathbf{var}\tau$.
- $\theta = \overline{\sigma} \to \mathbf{comm}$ not a valid case.

All the cases above generate contradictions, so it must be the case that $\alpha_1 = \alpha_2$.

END OF PROOF.

The rules for branching and iteration are given in Figure 6.4.
PROOF OF SOUNDNESS:

(IF): For any model $\langle u, v\rangle$ consistent with Ψ we need to prove that if

$$\begin{aligned}
&\text{for all } \omega \in \eta(v), \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{\mathit{tt}} u \cap \gamma(v) = \emptyset\\
&\text{and } \omega \cdot (\!|A_1|\!)_{\mathit{tt}} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset\\
&\text{and } \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset\\
&\qquad\text{implies } \omega \cdot (\!|M_1|\!)_{\mathsf{a}} u \cdot (\!|A_2|\!)_{\mathit{ff}} u = \emptyset\\
\text{and } &\text{for all } \omega \in \eta(v), \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset\\
&\text{and } \omega \cdot (\!|A_1|\!)_{\mathit{tt}} u \cdot (\!|B|\!)_{\mathit{tt}} u \cap \gamma(v) = \emptyset\\
&\text{and } \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset\\
&\qquad\text{implies } \omega \cdot (\!|M_2|\!)_{\mathsf{a}} u \cdot (\!|A_2|\!)_{\mathit{ff}} u = \emptyset,
\end{aligned}$$

for all $\mathsf{a} \in \mathcal{A}[\![\sigma]\!]$, $M : \sigma$, then

for all $\omega \in \eta(v), \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$ implies $\omega \cdot (\!|(\textbf{if } B \textbf{ then } M_1 \textbf{ else } M_2); A_2|\!)_{\mathit{ff}} u = \emptyset$, assuming that $\Psi \bullet \nabla(B)$. So we need to prove that

$$\begin{aligned} &\text{for all } \omega \in \eta(v), \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \\ &\text{implies } \omega \cdot (\!|B|\!)_{tt} u \cdot (\!|M_1|\!)_{\mathsf{a}} u \cdot (\!|A_2|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \\ &\quad \text{and } \omega \cdot (\!|B|\!)_{\mathit{ff}} u \cdot (\!|M_2|\!)_{\mathsf{a}} u \cdot (\!|A_2|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset, \end{aligned}$$

for all $\mathsf{a} \in \mathcal{A}[\![\sigma]\!]$, $M : \sigma$.

If $\omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$ then it follows that $\omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$ and $\omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{tt} u \cap \gamma(v) = \emptyset$.

From Determinacy Lemma 6.7 on page 127, it follows that

$$\omega \cdot (\!|A_1|\!)_{tt} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \text{ or } \omega \cdot (\!|A_1|\!)_{tt} u \cdot (\!|B|\!)_{tt} u \cap \gamma(v) = \emptyset. \tag{6.18}$$

If the first part is true then all the semantic conditions for the first premise are met, so $\omega \cdot (\!|M_1|\!)_{\mathsf{a}} u \cdot (\!|A_2|\!)_{\mathit{ff}} u = \emptyset$ and further, from the non-interference of B with M_1, A_2 we have that $\omega \cdot (\!|B|\!)_{tt} u \cdot (\!|M_1|\!)_{\mathsf{a}} u \cdot (\!|A_2|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$. On the other hand, $\omega \cdot (\!|B|\!)_{\mathit{ff}} u \cdot (\!|M_2|\!)_{\mathsf{a}} u \cdot (\!|A_2|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$ follows directly from the non-interference of A_1 and B which gives $\omega \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$. This semantically validates the conclusion of the rule.

If the second part of Equation 6.18 is true, we reason analogously, reversing M_1 and M_2.

(DO): We need to prove that for all $\omega \in \eta(v)$,

$$\text{if } \omega \cdot (\!|A|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \tag{6.19}$$

$$\text{and } \omega \cdot (\!|A|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{tt} u \cap \gamma(v) = \emptyset \tag{6.20}$$

$$\text{and } \omega \cdot (\!|A|\!)_{tt} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \tag{6.21}$$

$$\text{then } \omega \cdot (\!|C; A|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$$

implies for all $\omega \in \eta(v)$,

$$\omega \cdot (\!|A|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \text{ implies } \omega \cdot (\!|(\textbf{while } B \textbf{ do } C); A \textbf{ and not } B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset. \tag{6.22}$$

So given that

$$\omega \cdot (\!|A|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \tag{6.23}$$

and the premises we need to prove that

$$\omega \cdot ((\!|B|\!)_{tt} u \cdot (\!|C|\!) u)^* \cdot (\!|B|\!)_{\mathit{ff}} u \cdot (\!|A|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset, \tag{6.24}$$

$$\omega \cdot ((\!|B|\!)_{tt} u \cdot (\!|C|\!) u)^* \cdot (\!|B|\!)_{\mathit{ff}} u \cdot (\!|A|\!)_{tt} u \cdot (\!|B|\!)_{tt} u \cap \gamma(v) = \emptyset, \tag{6.25}$$

$$\omega \cdot ((\!|B|\!)_{tt} u \cdot (\!|C|\!) u)^* \cdot (\!|B|\!)_{\mathit{ff}} u \cdot (\!|A|\!)_{\mathit{ff}} u \cdot (\!|B|\!)_{tt} u \cap \gamma(v) = \emptyset. \tag{6.26}$$

Given the stability of B and the non-interference condition $\Psi \bullet A \# B$, the only non-trivial equation is 6.24.

We will prove by induction on $k \in \mathbb{N}$ that for all $k \geq 0$:

$$\omega \cdot (\llparenthesis B \rrparenthesis_{tt} u \cdot \llparenthesis C \rrparenthesis u)^k \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis A \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cap \gamma(v) = \emptyset.$$

- (Base case) $k = 0$, we need to prove that $\omega \cdot \epsilon \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis A \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cap \gamma(v) = \emptyset$. This is immediate, from equation 6.23 on the preceding page and the non-interference condition $\Psi \bullet A \# B$.
- (Inductive case) assuming that

$$\omega \cdot (\llparenthesis B \rrparenthesis_{tt} u \cdot \llparenthesis C \rrparenthesis u)^k \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis A \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis B \rrparenthesis_{tt} u \cap \gamma(v) = \emptyset,$$

we prove that

$$\begin{array}{ll} & \omega \cdot (\llparenthesis B \rrparenthesis_{tt} u \cdot \llparenthesis C \rrparenthesis u)^{k+1} \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis A \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis B \rrparenthesis_{tt} u \cap \gamma(v) = \emptyset, \\ \text{i.e.} & \omega \cdot \llparenthesis B \rrparenthesis_{tt} u \cdot \llparenthesis C \rrparenthesis u \cdot (\llparenthesis B \rrparenthesis_{tt} u \cdot \llparenthesis C \rrparenthesis u)^k \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis A \rrparenthesis_{\mathit{ff}} u \cdot \llparenthesis B \rrparenthesis_{tt} u \cap \gamma(v) = \emptyset. \end{array}$$

According to the Determinacy Lemma (6.7) for B:

$$\omega \cdot \llparenthesis B \rrparenthesis_{\mathit{ff}} u \cap \gamma(v) = \emptyset \tag{6.27}$$
$$\text{or } \omega \cdot \llparenthesis B \rrparenthesis_{tt} u \cap \gamma(v) = \emptyset \tag{6.28}$$

If equation 6.28 holds, then the conclusion follows immediately.
If equation 6.27 holds then it means that conditions 6.19– 6.21 on the page before are met, so from the premise of the rule, $\omega \cdot \llparenthesis C \rrparenthesis u \cdot \llparenthesis A \rrparenthesis_{\mathit{ff}} u \cap \gamma(v) = \emptyset$. This means that for all $\omega' \in \omega \cdot \llparenthesis C \rrparenthesis u \cap \gamma(v)$, $\omega' \cdot \llparenthesis A \rrparenthesis_{\mathit{ff}} u = \emptyset$. So we can apply the induction hypothesis, which immediately proves the rule. END OF PROOF.

A weaker form of the rules for branching and iteration is:

INFERENCE RULES

$$\text{IF'}\ \frac{\Psi \bullet \{A_1\}\ M_1\ \{A_2\} \quad \Psi \bullet \{A_1\}\ M_2\ \{A_2\}}{\Psi \bullet \{A_1\}\ \textbf{if}\ B\ \textbf{then}\ M_1\ \textbf{else}\ M_2\ \{A_2\}}$$

$$\text{DO'}\ \frac{\Psi \bullet \{A\}\ C\ \{A\}}{\{A\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{A\ \textbf{and not}\ B\}}$$

IF' non-interference conditions $\Psi \bullet B \# M_1, \Psi \bullet B \# M_2, \Psi \bullet B \# A_2, \Psi \bullet A_1 \# B$
DO' non-interference conditions $\Psi \bullet A \# B, \Psi \bullet B \# C$.

These rules apply in case that the guard B is not stable, so that B cannot be used as a precondition in the specifications for M_i or C. The proofs of soundness are similar to the proofs for the IF and DO rules.

The final two rules we mention here are *Non-interference abstraction* and *Constancy*. They both provide the means to prove invariant assertions about programs and "factor out" these assertions in reasoning about the programs. The first one was first introduced in [Rey81b], and the second one in [OT93b].

INFERENCE RULES

$$\text{CON}\ \frac{\Psi \bullet \{A\} \Rightarrow \{A_1\}\ M\ \{A_2\}}{\Psi \bullet \{A \ \mathbf{and}\ A_1\}\ M\ \{A \ \mathbf{and}\ A_2\}}\ \Psi \bullet M \# A$$

$$\text{ABS}\ \frac{\Psi \bullet \{A\}\ M\ \{A\} \qquad \Psi \bullet \forall m : \sigma.\{A\} \Rightarrow \{A_1\}\ F(m)\ \{A_2\}}{\Psi \bullet \{A \ \mathbf{and}\ A_1\}\ F(M)\ \{A \ \mathbf{and}\ A_2\}}\ \begin{array}{l}\Psi \bullet F \# A \\ \Psi \bullet M \# A_2 \\ \Psi \bullet M \# F\end{array}$$

provided, in both rules, that $\Psi \bullet A \# A_1$ and $\Psi \bullet A \# A_2$.

In ABS, M and m have the same type σ. The ABS rule is less powerful than originally presented in [OT93b], because it does not use any assumptions about the argument m of function P and it requires M not to interfere with A_2 and F. We will see in the next chapter a stronger version of this rule which is closer to the original.

Informally, the CON(stancy) rule says that if a term M does not interfere with a true assertion A then the assertion may be assumed to be true *throughout* the execution of M, so it can be treated as a local mathematical fact in partial correctness reasoning about M. The ABS(traction) rule says that procedure F uses its argument *as a unit*, and if assertion A is an invariant for the argument, and it is not interfered with by the body of the function then it will also hold throughout the execution of $F(M)$, so it too may be treated as a local mathematical fact in reasoning about $F(M)$.

PROOF OF SOUNDNESS:

(CON): For any model $\langle u, v\rangle$ consistent with Ψ, if the precondition of the conclusion holds then for all $\omega \in \eta(v)$:

$$\begin{aligned}\omega \cdot (\!|A|\!)_{\mathit{ff}} u \cdot (\!|A_1|\!)_{\mathit{tt}} u \cap \gamma(v) &= \emptyset \\ \omega \cdot (\!|A|\!)_{\mathit{tt}} u \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) &= \emptyset \\ \omega \cdot (\!|A|\!)_{\mathit{ff}} u \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) &= \emptyset.\end{aligned}$$

From the non-interference of A and A_1 and Lemma 6.3 on page 109 this implies:

$$\begin{aligned}\omega \cdot (\!|A|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset &\text{ or } \omega \cdot (\!|A_1|\!)_{\mathit{tt}} u \cap \gamma(v) = \emptyset \\ \omega \cdot (\!|A|\!)_{\mathit{tt}} u \cap \gamma(v) = \emptyset &\text{ or } \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset \\ \omega \cdot (\!|A|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset &\text{ or } \omega \cdot (\!|A_1|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset.\end{aligned}$$

If $\omega \cdot (\!|A|\!)_{\mathit{ff}} u \cap \gamma(v) = \emptyset$ and $\omega \cdot (\!|A|\!)_{\mathit{tt}} u \cap \gamma(v) = \emptyset$, i.e. A does not terminate normally in ω the postcondition is satisfied trivially.

If $\omega \cdot (\!| A |\!)_{tt} u \cap \gamma(v) \neq \emptyset$ then it must be the case that $\omega \cdot (\!| A_1 |\!)_{ff} u \cap \gamma(v) = \emptyset$ and, from the Determinacy Lemma, 6.7 on page 127, $\omega \cdot (\!| A |\!)_{ff} u \cap \gamma(v) = \emptyset$. Applying the premise of the rule, we have that

$$\omega \cdot (\!| M |\!)_{\alpha} u \cdot (\!| A_2 |\!)_{ff} u \cap \gamma(v) = \emptyset. \tag{6.29}$$

We need to prove that

$$\omega \cdot (\!| M |\!)_{\alpha} u \cdot (\!| A |\!)_{ff} u \cdot (\!| A_2 |\!)_{tt} u \cap \gamma(v) = \emptyset \tag{6.30}$$
$$\omega \cdot (\!| M |\!)_{\alpha} u \cdot (\!| A |\!)_{tt} u \cdot (\!| A_2 |\!)_{ff} u \cap \gamma(v) = \emptyset \tag{6.31}$$
$$\omega \cdot (\!| M |\!)_{\alpha} u \cdot (\!| A |\!)_{ff} u \cdot (\!| A_2 |\!)_{ff} u \cap \gamma(v) = \emptyset. \tag{6.32}$$

Equations 6.30 and 6.32 follow from 6.29 and the non-interference of M with A and A with A_2. Equation 6.31 is implied directly from the premise.

The other case, $\omega \cdot (\!| A |\!)_{ff} u \cap \gamma(v) \neq \emptyset$ makes both the premise and the conclusion trivially true.

(ABS): If the precondition of the conclusion holds then for all $\omega \in \eta(v)$,

$$\omega \cdot (\!| A |\!)_{tt} u \cdot (\!| A_1 |\!)_{ff} u \cap \gamma(v) = \emptyset$$
$$\omega \cdot (\!| A |\!)_{ff} u \cdot (\!| A_1 |\!)_{ff} u \cap \gamma(v) = \emptyset$$
$$\omega \cdot (\!| A |\!)_{ff} u \cdot (\!| A_1 |\!)_{tt} u \cap \gamma(v) = \emptyset.$$

For the same reasons as before, the only interesting case is $\omega \cdot (\!| A |\!)_{ff} u \cap \gamma(v) = \emptyset$ and $\omega \cdot (\!| A_1 |\!)_{tt} u \cap \gamma(v) \neq \emptyset$. The precondition of both premises is met, therefore:

$$\omega \cdot (\!| M |\!)_{\mathsf{a}} u \cdot (\!| A |\!)_{ff} u \cap \gamma(v) = \emptyset,$$
$$\omega \cdot (\!| Fm |\!)_{\mathsf{a}'} u \cdot (\!| A_2 |\!)_{ff} u \cap \gamma(v) = \emptyset.$$

for all $\mathsf{a} \in \mathcal{A}[\![\sigma]\!]$, and all $\mathsf{a}' \in \mathcal{A}[\![\sigma \to \sigma']\!]$, where $F : \sigma \to \sigma'$. We need to prove that:

$$\omega \cdot (\!| FM |\!)_{\mathsf{a}'} u \cdot (\!| A |\!)_{tt} u \cdot (\!| A_2 |\!)_{ff} u \cap \gamma(v) = \emptyset \tag{6.33}$$
$$\omega \cdot (\!| FM |\!)_{\mathsf{a}'} u \cdot (\!| A |\!)_{ff} u \cdot (\!| A_2 |\!)_{tt} u \cap \gamma(v) = \emptyset \tag{6.34}$$
$$\omega \cdot (\!| FM |\!)_{\mathsf{a}'} u \cdot (\!| A |\!)_{ff} u \cdot (\!| A_2 |\!)_{ff} u \cap \gamma(v) = \emptyset. \tag{6.35}$$

Equations 6.33 and 6.35 follow directly by applying the trace non-interference substitution lemma (6.3), as M does not interfere with F and A_2 and the traces associated with m are non-empty.

Equation 6.34 is proved by induction on the number of times F uses its argument, using the same method as in the proof for the DO rule (page 129).

END OF PROOF.

6.6 Side-conditions and semantic cheating

The reader will have noticed that in the rules of the previous sections we have used two types of side-conditions: non-interference and normal termination. These rules are defined in terms of the *semantics* of terms rather than in terms of syntactic or logical properties; this is why they can be considered cheating.

This form of cheating, however, is not unreasonable in the context of model-checking, as they are clearly mechanically decidable properties. Also, if the premises of a rule are model-checked then one of the more computationally expensive phases of the verification, constructing the regular-language model, has already been accomplished and the additional verification of the side-conditions should come at not too great an expense.

Defining non-interference in semantic terms rather than syntactic ones has the advantage of precision. For example, the semantic definition of non-interference correctly establishes that the following two phrases do not interfere:

$$\nabla x : \mathbf{var}\tau.\nabla v : \mathbf{var}\tau.(x := 0; \mathbf{if}\ !x \neq 0\ \mathbf{then}\ v := 0\ \mathbf{else\ skip})\ \#\ !v.$$

The first phrase contains a command which writes to v, but it occurs in *dead code*, code which will never be executed. Using axioms such as Reynolds's to prove non-interference, the non-interference condition above cannot be proved, but semantic-level verification does the job.

The disadvantage of this form of cheating is that, unlike inference, it requires model checking so it is potentially expensive. As is generally the case with model checking, sometimes this is worth the effort but some other times it is not. Many times a set of simple syntactical or logical rules, even if it is incomplete, can be useful and helpful in avoiding unnecessary model checking.

A set of rules for proving non-interference must strike a balance between simplicity and power. A complete but complex set of rules is powerful but because of its complexity one might prefer the more direct approach of mechanical semantic verification. A simplistic set of rules might be very easy to use but it might be too incomplete to be of any help. The same can be said about proving normal termination.

A simple yet powerful method of syntactically determining non-interference is the one that Reynolds uses: two phrases do not interfere if the free identifiers of the two phrases do not interfere pairwise. In our case, we have to add that the free identifiers of the phrase must be bound by stability quantifiers. Non-stable identifiers may be interfered with, because this is exactly the meaning of the fact that they lack stability.

We define the set of variables modified by a phrase inductively on its syntax. Without loss of generality, we can assume the term is in **let**-free β-normal form (Lemma 4.4 on page 75).

Definition 6.9 *We define the set $Mod_\Gamma(M)$, where $\Gamma \vdash M : \sigma$, to be*

$$Mod_\Gamma(V := E) = \{v \in Free_\Gamma(V), \Gamma(v) = \mathbf{var}\tau\} \cup Mod(E) \cup Mod(V)$$
$$Mod_\Gamma\big(FV\big) = Mod_\Gamma(F) \cup Mod_\Gamma(V) \cup \{v \in Free_\Gamma(V), \Gamma(v) = \mathbf{var}\tau\},\ \ \mathit{if}\ \Gamma \vdash V : \mathbf{var}\tau$$
$$Mod_\Gamma(!V) = Mod_\Gamma(V)$$

$$
\begin{aligned}
&Mod_\Gamma(M \ \mathbf{op}\ M') = Mod_\Gamma(M) \cup Mod_\Gamma(M')\\
&Mod_\Gamma(\mathbf{while}\ B\ \mathbf{do}\ C) = Mod_\Gamma(B) \cup Mod_\Gamma(C)\\
&Mod_\Gamma(\mathbf{if}\ B\ \mathbf{then}\ M\ \mathbf{else}\ M') = Mod_\Gamma(B) \cup Mod_\Gamma(M) \cup Mod_\Gamma(M')\\
&Mod_\Gamma\big(FM\big) = Mod_\Gamma(M) \cup Mod_\Gamma(F),\ \ \mathit{if}\ \Gamma \vdash M : \sigma \neq \mathbf{var}\tau\\
&Mod_\Gamma(x) = \emptyset\\
&Mod_\Gamma(\mathbf{k}) = \emptyset,
\end{aligned}
$$

*where **k** is any language constant.*

Lemma 6.8 (Syntactic characterization of non-interference) *Let X be the set of identifiers bound by $\boldsymbol{\nabla}$ in Ψ. Then $\Psi \bullet P \,\#\, P'$ if*

$$Mod(P) \cap Free(P') \cap X = Mod(P') \cap Free(P) \cap X = \emptyset.$$

PROOF: Directly from the definition of the set *Mod* and trace non-interference, 6.4 on page 109. END OF PROOF.

This syntactic characterization is rather obvious. The set $Mod_\Gamma(M)$ collects all the variable-identifiers in M appearing on the left-hand-side of an assignment statement. Maybe the only not entirely obvious part of the definition is the one concerning non-local functions taking variables as arguments. Just as in the case when variable V appears on the left-hand side of an assignment, all its freely-occurring variables can be potentially assigned to and modified. For example, if $V ::= \mathbf{if}\ B\ \mathbf{then}\ x\ \mathbf{else}\ y$ then both x and y may be modified if V is assigned to or passed as argument to a function.

The rules of Definition 6.9 on the preceding page are quite crude and perhaps they do not achieve the best syntactic characterization of non-interference. For example, in the example just above, if boolean phrase $B ::= !b$, with b a boolean variable, b would appear in $Mod_\Gamma(V)$ although it cannot actually be modified. But we will not pursue a more sophisticated syntactic characterization of non-interference any further.

For normal termination, all but the most naive syntactic characterizations are too complex. Fortunately, the naive syntactic characterization is quite satisfactory if one keeps in mind that we are concerned about the normal termination of *assertions*. Any assertion which is free of **while**-loops and **diverge** will terminate normally.

Chapter 7

Procedure specifications

One of the Cinderella tasks in logic is the bookkeeping of substitution. Dirk van Dalen

7.1 Effect specifications

The specification and verification methods presented in the previous two chapters are missing an important ingredient: reasoning about non-locally defined functions and procedures by making assumptions about their *effect.* The only assumptions about non-local entities we are currently handling are of stability, which, as we have discussed on several occasions, are essential in order to support static reasoning. But in order to truly support compositional model checking we need more.

For example, consider a parameterless procedure *inc* that increments some global expression x. We would like to be able to specify something like this:

$$\nabla x : \mathbf{expint}.\nabla y : \mathbf{expint}.\forall inc : \mathbf{comm} \\ \bullet \{x = y\}\ inc\ \{x = y + 1\} \Rightarrow \{x = y\}\ inc; inc\ \{x = y + 2\} \quad (7.1)$$

Informally, if procedure *inc* increments expression x then executing it twice it will increment the expression by 2. But this clearly does not work as desired, because assuming the stability of x prevents procedure *inc* from changing its value; a stable expression cannot be modified! So the specification above is true vacuously, because the premise will always be false.

The first step is to weaken the stability quantifier so that certain non-local objects are allowed to interfere with an identifier bound by a stability quantifier; let us call this new quantifier a *relative-stability quantifier.* Syntactically, this is accomplished by providing the list of identifiers denoting interfering non-local objects as part of the quantifier:

$$S ::= \nabla x : \theta / x_1 : \theta_1, \ldots, x_k : \theta_k \bullet S$$

with $x \neq x_i$, $1 \leq i \leq k$. Note that identifiers x_i are not bound by the quantifier; this causes some technical problems, so we will give the typing rule later, in a more general framework.

Semantically, the relative-stability quantifier is interpreted by using the stability regular expressions as defined earlier, with the additional proviso that should one of the interfering identifiers be used, stability is "interrupted" at that point:

Definition 7.1 (Semantics of relative-stability quantifier)

$$\langle u, v\rangle \models \nabla x : \theta / x_1 : \theta_1, \ldots, x_k : \theta_k \bullet S : \mathbf{spec} \quad \textit{if and only if}$$
$$\big\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \gamma_\theta^{x/x_1\cdots x_k})\big\rangle \models S \quad \textit{and} \quad \big\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \gamma_\theta^{x/x_1\cdots x_k})\big\rangle \models S,$$

$$\textit{where } \gamma_\theta^{x/x_1\cdots x_k} = \Big(\gamma^x_\theta \cdot \textstyle\sum_{1\leq i \leq k} \mathcal{A}\,[\![x_i : \theta_i]\!]\Big)^* \cdot \gamma^x_\theta.$$

For example, if x is bound by $\nabla x : \mathbf{expint}/\mathit{inc} : \mathit{com}$ then a trace such as $q^{\langle x\rangle} \cdot 3^{\langle x\rangle} \cdot \mathit{run}^{\langle \mathit{inc}\rangle} \cdot \mathit{done}^{\langle \mathit{inc}\rangle} \cdot q^{\langle x\rangle} \cdot 7^{\langle x\rangle}$ is allowed by the relative-stability quantifier.

We can see that if the list of interfering identifiers is empty, the definition above reduces to the initial definition of the stability quantifier, Definition 5.9 on page 94.

Relativizing the stability quantifier, however, does not yet solve the problem. To do that we need to re-iterate a discussion we had earlier[1], about the clash between the style of semantics employed in our approach and the traditional idiom of specification based on the combined powers of the universal quantifier and implication. To accommodate model checking, both are made significantly weaker in our semantics; more precisely, no universal quantifiers are used in the semantics. As a result, the specification above is still vacuously true, because the premise of the implication is still always false, but for a different reason. The *inc* procedure is bound, in the environment, to a copy-cat regular language which has no effects on variable x. The specification says that *inc* may change x but not *how* it may change x.

We have the same situation as before, a need to specify the behaviour of a non-local object, and, as before, the problem is the same. It makes sense to use the same solution, and employ a generalized (or, rather, specialized) quantifier instead of implication to introduce x. So let us look at a second attempt to specify this, but using an "abstract-model" approach:

$$\nabla y : \mathbf{expint}.\nabla x : \mathbf{expint}/\mathit{inc} : \mathbf{comm}.\big\langle \{x = y\}\ \mathit{inc} : \mathbf{comm}\ \{x = y + 1\}\big\rangle$$
$$\bullet\ \{x = y\}\ \mathit{inc}; \mathit{inc}\ \{x = y + 2\} \quad (7.2)$$

In the above, we call $\big\langle \{x = y\}\ \mathit{inc} : \mathbf{comm}\ \{x = y + 1\}\big\rangle$ an *effect quantifier.* It introduces non-local parameterless procedure *inc* through its effect on expression x. In a certain sense, we can think of the quantifier as a model or an abstract implementation of procedure *inc.* As before, this notion is quite closely related to de Alfaro and Henzinger's *interface automata* [dAH01].

[1] See page 91.

There is still a problem to iron out. Expression-identifier y denotes a global expression. We would like to specify procedure *inc* as an increment of x when x has *any* value, not merely the value of a certain global expression. So, clearly, the role of y in the quantifier, i.e. in the specification of *inc*, is not the same as its role in the specification of the program using *inc*. In the first case, it should be internal to the procedure specification. The same argument applies to the original expression y, which also fulfills an auxiliary role. So we write, also changing one of the y's to z for clarity:

$$\nabla x : \mathbf{expint}/inc : \mathbf{comm}.\big\langle \nabla z : \mathbf{expint} \bullet \{x = z\}\ inc : \mathbf{comm}\ \{x = z + 1\}\big\rangle \\ \bullet\ \nabla y : \mathbf{expint}.\{x = y\}\ inc; inc\ \{x = y + 2\} \quad (7.3)$$

The syntax of the effect quantifier is:

$$S ::= \big\langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A_1'\}, \cdots, \{A_k\}\ x : \sigma\ \{A_k'\}\big\rangle \bullet S,$$

where Ψ is a sequence of quantifiers.

The informal semantics is that if any pre-condition A_i is true then, after the evaluation of x, post-condition A_i' is also true. Before giving the formal semantics we need to clarify a syntactical issue.

Specifications involving relative stability and effect quantifiers have the rather unpleasant feature that, depending on how one writes the specification, identifiers sometimes appear to be used outside of their scope. In the example above, for example, the identifier for procedure *inc* is used in the relative-stability quantifier binding x which occurs earlier. Notice that reversing the order of the quantifiers does not solve the problem, because x also occurs freely in the effect quantifier binding *inc*. This syntactical problem is typical of generalized quantifiers, and several ways of dealing with it have been proposed in the literature. The standard approach is to interpret all the inter-dependent quantifiers as simultaneously introduced, so the *inc* example would be written as:

$$\left(\begin{array}{c} \nabla x : \mathbf{expint}/inc : \mathbf{comm} \\ \big\langle \nabla z : \mathbf{expint} \bullet \{x = z\}\ inc : \mathbf{comm}\ \{x = z + 1\}\big\rangle \end{array}\right) \\ \bullet\ \nabla y : \mathbf{expint}.\{x = y\}\ inc; inc\ \{x = y + 2\} \quad (7.4)$$

Let $\psi x : \theta$ range over all possible quantifiers binding a variable x:

$$\psi x : \sigma ::= \forall x : \sigma \;\big|\; \nabla x : \sigma/x_1 : \theta_1, \ldots, x_n : \theta_n \;\big|\; \big\langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A_1'\}, \cdots, \{A_k\}\ x : \sigma\ \{A_k'\}\big\rangle$$
$$\psi x : \theta ::= \forall x : \theta \;\big|\; \nabla x : \theta/x_1 : \theta_1, \ldots, x_n : \theta_n$$

Definition 7.2 (Syntactic dependence) *We write $\psi x : \theta/X$ to indicate that x bound by ψ is syntactically dependent on variables $X = \{x_1, \ldots, x_n\}$:*

- *if $\psi x = \forall x : \theta$ then $X = \emptyset$;*
- *if $\psi x = \nabla x : \theta/x_1 : \theta_1, \ldots, x_n : \theta_n$ then $X = \{x_1, \ldots, x_n\}$;*

- *if* $\psi x = \langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A'_1\}, \cdots, \{A_k\}\ x : \sigma\ \{A'_k\} \rangle$ *then* $X = \bigcup_{1 \leq i \leq k} \mathit{Free}(\Psi \bullet A_i) \cup \mathit{Free}(\Psi \bullet A'_i)$.

The general typing rule for simultaneous generalized quantifiers is:

TYPING RULE

$$\frac{\Gamma, x_1 : \theta_1, \ldots, x_n : \theta_n \vdash S : \mathbf{spec}}{\Gamma \vdash \begin{pmatrix} \psi x_1 : \theta_1 / X_1 \\ \vdots \\ \psi x_n : \theta_n / X_n \end{pmatrix} \bullet S : \mathbf{spec}}\ X_i \subseteq \{x_1, \ldots, x_n\},\ 1 \leq i \leq n$$

For any $\psi x = \langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A'_1\}, \cdots, \{A_k\}\ x : \sigma\ \{A'_k\} \rangle$ the specification must also satisfy the additional premises:

$$\Gamma, x_1 : \theta_1, \ldots, x_n : \theta_n \vdash \Psi \bullet \{A_i\}\ x : \sigma\ \{A'_i\} : \mathbf{spec},\ 1 \leq i \leq k.$$

In fact, the collection of simultaneous generalized quantifiers above is a single composite quantifier, binding n identifiers simultaneously in the n formulas defining the quantifier. In generalized quantifier theory, this is called a type $(\underbrace{n, n, \ldots, n}_{n \text{ times}})$ quantifier.

Specifications with simultaneous generalized quantifiers are not *compositional* in the usual semantic sense. Due to the simultaneous introduction of the quantifiers we cannot give the semantic definitions inductively on the syntax. The reason is the potentially troublesome cross-dependencies between identifiers. Consider the following example:

$$\begin{pmatrix} \nabla x : \mathbf{varint}/\mathit{do} : \mathbf{comm}, \mathit{undo} : \mathbf{comm} \\ \langle \{!x = 0\}\ \mathit{do}\ \{\mathit{undo}; !x = 0\} \rangle \\ \langle \{!x = 0\}\ \mathit{undo}\ \{\mathit{do}; !x = 0\} \rangle \end{pmatrix} \bullet S. \tag{7.5}$$

The procedures specified above, *do* and *undo*, satisfy their specifications if one increments and the other decrements x by the same value. However, this specification of procedures through their *properties* is not in the vein of the *abstract-model* style of specifications we desire, so it is a situation we need to avoid.

Alechina and van Lambalgen [AvL96], in their proof theoretic treatment of generalized quantifiers, argue that it is reasonable to banish such circular dependencies; but, in contrast, Hintikka strongly emphasizes [Hin96] that the lack of compositionality brought about by simultaneous dependent quantifiers gives unique expressive powers to the semantics. We have argued before that, for model-checking purposes, too much expressiveness can be a bad thing, so we restrict simultaneous generalized quantifiers so that no semantic circular dependencies exist.

A possible non-circular specification of procedures for Example 7.5 on the facing page is:

$$\left(\begin{array}{c} \nabla x : \mathbf{varint}/\mathit{do} : \mathbf{comm}, \mathit{undo} : \mathbf{comm} \\ \langle \nabla y : \mathbf{expint} \bullet \{!x = y\}\ \mathit{do}\ \{!x = y + 1\}\rangle \\ \langle \nabla y : \mathbf{expint} \bullet \{!x = y\}\ \mathit{undo}\ \{!x = y - 1\}\rangle \end{array}\right) \bullet S.$$

If we consider a more abstract specification better, then the same can be better specified by:

$$\left(\begin{array}{c} \nabla x : \mathbf{varint}/\mathit{do} : \mathbf{comm}, \mathit{undo} : \mathbf{comm} \\ \langle \nabla y : \mathbf{expint} \bullet \{!x = y\}\ \mathit{do}\ \{!x = y + 1\}\rangle \\ \langle \nabla y : \mathbf{expint} \bullet \{!x = y\}\ \mathit{undo}\ \{\mathit{do}; !x = y\}\rangle \end{array}\right) \bullet S. \tag{7.6}$$

The elimination of circular dependencies in specifications is a substantial restriction, but not a crippling one. A certain style of specifications, *algebraic* or *axiomatic*, are not possible using this specification language. For example, the specification in Example 7.5 on the preceding page is not possible. This is a limitation, because this style of specifications tend to be the most abstract and elegant.

However, the same programs can be specified instead using *abstract models*.[2] Procedures are not specified in terms of their properties. They are *defined* at a level of abstraction that is more than that of an implementation but less than that of an algebraic (or axiomatic) specification. Because we are dealing with definitions rather than properties, it is critical that we prevent circular dependencies. In fact, by preventing circular dependencies we are restricting the order of the generalized quantifier from $(\underbrace{n, \ldots, n}_{n \text{ times}})$ to $(\underbrace{1, \ldots, 1}_{n \text{ times}})$.

According to the Hierarchy Theorem for generalized quantifiers [HLV96] the quantifiers of the higher order (with circular dependencies) are strictly more expressive than those of the lower order (flattened). However, to the knowledge of the author, there is no evidence that the practicality of abstract-model-style specifications is substantially impaired, compared to that of algebraic or axiomatic specifications. Recent software-specifications books, such as [Ten02], give numerous examples of specifications using abstract-model-style specifications. Dorfam and Hayer [DT96] have written an introduction to and discussion of the comparative merits of these specification styles, with many pointers to literature.

We define the dependence relation for a collection of simultaneous generalized quantifiers as follows:

Definition 7.3 (Semantic dependence) *Given a collection of quantifiers Ψ we define the semantic dependence relation $\mathcal{D}_\Psi$ on the identifiers occurring in Ψ as the transitive closure of:*

- $x\ \mathcal{D}_\Psi\ x'$ *if* $\nabla x : \theta / \cdots x' : \theta' \cdots$ *is in* Ψ;

[2]The terms *algebraic specification* and *abstract model* are used informally in this context.

- $x\ \mathcal{D}_\Psi\ x'$ *if* $\langle \Psi' \bullet \{A_1\}\ x' : \theta'\ \{A'_1\}, \ldots, \{A_k\}\ x' : \theta'\ \{A'_k\}\rangle$ *is in* Ψ *and* $x \in \bigcup_{1 \le i \le k} \mathit{Free}(\Psi' \bullet A_i) \cup \mathit{Free}(\Psi' \bullet A'_i)$.

To the general typing rule for simultaneous generalized quantifiers Ψ we add the non-circularity condition:

Definition 7.4 (Non-circularity) *A collection of simultaneously dependent generalized quantifiers is said to be non-circular if its semantic dependence relation* $\mathcal{D}_\Psi$ *is a strict partial order.*

A non-circularly dependent collection of quantifiers can be topologically sorted with respect to the ordering given by semantic dependence ("flattened") into a sequence of quantifiers. Instead of

$$\left(\begin{array}{c} \psi x_1 : \theta_1/X_1 \\ \vdots \\ \psi x_n : \theta_n/X_n \end{array}\right) \bullet S : \mathbf{spec}$$

we can write

$$(\psi x_{i_1} : \theta_{i_1}/X_{i_1}, \ldots, \psi x_{i_n} : \theta_{i_n}/X_{i_n}) \bullet S,$$

where $\{i_1, \ldots, i_n\}$ is a permutation of $\{1, \ldots, n\}$ and if $j < k$ then $(x_{i_k}, x_{i_j}) \notin \mathcal{D}_\Psi$, i.e. we must introduce all the semantic dependents of an identifier before we introduce the identifier itself. If the collection of quantifiers is circular then it is impossible to flatten it. If there is no confusion we might abuse the notation and drop the brackets. The formulation of the *inc* example in equation 7.3 on page 137 is then a properly flattened version of equation 7.4.

For the purpose of giving a semantic interpretation of the quantifiers we assume that all collections of quantifiers are flattened and we will only use the brackets to identify the collections if necessary. With this assumption, we can define the meaning of specifications involving relative-stability and effect quantifiers in the usual style, inductively on syntax. Definition 7.1 on page 136 is therefore sensible.

Let $|\Psi|$ be the set of identifiers bound by the quantifiers in Ψ. The regular-language interpretation of the effect of such a quantifier is:

Definition 7.5 (Effect) *For a model* $\mathfrak{M}_\Gamma = \langle u, v\rangle$*, the effect associated with quantifier* $\psi = \langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A'_1\}, \ldots, \{A_k\}\ x : \sigma\ \{A'_k\}\rangle$ *is the regular language*

$$\mathcal{F}\langle\psi\rangle(u, v) = \sum_{1 \le i \le k} \mathcal{K}^x_\sigma[\kappa_{pre}][\kappa_{post}] \cap \gamma(v \diamond \Psi) \upharpoonright \bigcup_{x_0 \in |\Psi|} \mathcal{A}^{\overline{x}_0}, \tag{7.7}$$

where $\kappa_{pre} : \sum_{\mathsf{q} \in \mathsf{Q}[\![\sigma]\!]} \mathsf{q} \cdot \mathsf{q}^{\langle x\rangle} \to \sum_{\mathsf{q} \in \mathsf{Q}[\![\sigma]\!]} \mathsf{q} \cdot ([\![A_i]\!]\,(u \diamond \Psi)) \cdot \mathsf{q}^{\langle x\rangle}$*, such that* $\kappa_{pre}(\mathsf{q} \cdot \mathsf{q}^{\langle x\rangle}) = \mathsf{q} \cdot (\!|A_i|\!)_{tt}(u \diamond \Psi) \cdot \mathsf{q}^{\langle x\rangle}$ *and* $\kappa_{post} : \sum_{\mathsf{a} \in \mathsf{A}[\![\sigma]\!]} \mathsf{a}^{\langle x\rangle} \cdot \mathsf{a} \to \sum_{\mathsf{a} \in \mathsf{A}[\![\sigma]\!]} \mathsf{a}^{\langle x\rangle} \cdot ([\![A'_i]\!]\,(u \diamond \Psi)) \cdot \mathsf{a}$*, such that* $\kappa_{post}\left(\mathsf{a}^{\langle x\rangle} \cdot \mathsf{a}\right) = \mathsf{a}^{\langle x\rangle} \cdot (\!|A'_i|\!)_{tt}(u \diamond \Psi) \cdot \mathsf{a}$.

The meanings of true pre-conditions are "forced" into the meaning of x just before its execution and the true post-conditions just after, realizing what can be thought of as an "abstract implementation" of x. The interactions contributed by the identifiers bound in the quantifier itself are hidden from the enclosing specification using restriction.

Looking back at the running example of the increment procedure, if in the environment *inc* is bound by the effect quantifier $\langle \nabla z : \mathbf{expint} \bullet \{x = z\}\ inc\ \{x = z + 1\}\rangle$ to the regular expression:

$$\mathcal{F}\,\langle \nabla z : \mathbf{expint} \bullet \{x = z\}\ inc\ \{x = z+1\}\rangle\,(u,v) \\ = \sum_{n \in \mathcal{Z}} run \cdot q^{\langle x \rangle} \cdot n^{\langle x \rangle} \cdot run^{\langle inc \rangle} \cdot done^{\langle inc \rangle} \cdot q^{\langle x \rangle} \cdot (n+1)^{\langle x \rangle} \cdot done.$$

then the example specification in equation 7.3 on page 137 is validated.

Definition 7.6 (Complete specification) $\langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A'_1\}, \ldots, \{A_k\}\ x : \sigma\ \{A'_k\}\rangle$ *is called a* complete specification *of x in a model $\mathfrak{M} = \langle u, v\rangle$ if for any state $\omega \in \eta(v)$*

$$\omega \cdot \mathcal{F}\langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A'_1\}, \ldots, \{A_k\}\ x : \sigma\ \{A'_k\}\rangle(u,v) \cap \gamma(v) \neq \emptyset.$$

The effect quantifier in our running example is a complete specification. The specification is complete in the sense that in any state at least one pre-condition A_i and the corresponding post-condition A'_i are true, i.e. for all $\omega \in \eta(v)$, there is i such that $\omega \cdot (\!| A_i |\!)_{tt} u \cdot (\!| A'_i |\!)_{tt} u \cap \gamma(v) \neq \emptyset$. Notice that the interactions contributed by the copy-cat strategy in Definition 7.5 on the preceding page cannot cause any interference, because their actions are removed by restriction.

In contrast, an incompletely specified procedure is *flip*, introduced by the effect quantifier in the following specification:

$$\nabla x : \mathbf{expbool}/\mathit{flip} : \mathbf{comm}.\langle \{x = \mathbf{true}\}\ \mathit{flip} : \mathbf{comm}\ \{x = \mathbf{false}\}\rangle \\ \bullet \{x = \mathbf{false}\}\ \mathit{flip}\ \{x = \mathbf{false}\}.$$

The effect quantifier binds procedure *flip* to the regular language:

$$\mathcal{F}\langle \{x = \mathbf{true}\}\ \mathit{flip} : \mathbf{comm}\ \{x = \mathbf{false}\}\rangle(u,v) \\ = run \cdot q^{\langle x \rangle} \cdot tt^{\langle x \rangle} \cdot run^{\langle \mathit{flip} \rangle} \cdot done^{\langle \mathit{flip} \rangle} \cdot q^{\langle x \rangle} \cdot \mathit{ff}^{\langle x \rangle} \cdot done.$$

The behaviour of *flip* in the case that $x = \mathbf{false}$ is not specified. However, if *flip* is interpreted as $\mathcal{F}\langle \{x = \mathbf{true}\}\ \mathit{flip}\ \{x = \mathbf{false}\}\rangle(u,v)$, the Hoare triple $\{x = \mathbf{false}\}\ \mathit{flip}\ \{x = \mathbf{false}\}$ is validated, through divergence, caused by the mismatch between the falsity of stable variable x in the pre-condition as opposed to its truth in the regular language interpreting the specification of *flip*.

This is obviously not the behaviour we want, and the Hoare triple should be false. If the effect of a non-local object is incompletely specified we must supply a default behavior for the situations not covered by the pre-conditions of the effect quantifier. A reasonable default behaviour is one in which any effect constitutes a possible outcome.

We first define the "missing states" of an effects quantifier as the set of all states which do not satisfy any of the pre-conditions of the quantifier.

Definition 7.7 (Missing states) *If* $\psi = \langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A_1'\}, \dots, \{A_k\}\ x : \sigma\ \{A_k'\}\rangle$, *then its* set of missing states *is* $\mathcal{M}\langle\psi\rangle(u,v) = \{\omega \in \eta(v) \mid \omega \cdot \mathcal{F}\langle\psi\rangle(u,v) \cap \gamma(v) = \emptyset\}$.

Definition 7.8 (Completed specification) *If* $\psi = \langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A_1'\}, \dots, \{A_k\}\ x : \sigma\ \{A_k'\}\rangle$ *is not a complete specification, then the* completed specification *is:*

$$\lceil \mathcal{F}\langle\psi\rangle(u,v)\rceil = \mathcal{F}\langle\psi\rangle(u,v) + \mathcal{K}_\sigma^x[\kappa_{pre}][\kappa_{post}] \cap \gamma(v),$$

where $\kappa_{pre} : \sum_{\mathsf{q}\in\mathsf{Q}[\![\sigma]\!]} \mathsf{q}\cdot\mathsf{q}^{\langle x\rangle} \to \mathcal{A}[\![\Gamma]\!]^*$, *such that* $\kappa_{pre}(\mathsf{q}\cdot\mathsf{q}^{\langle x\rangle}) = \mathsf{q}\cdot\mathcal{M}\langle\psi\rangle(u,v)\cdot\mathsf{q}^{\langle x\rangle}$ *and* $\kappa_{post} : \sum_{\mathsf{a}\in\mathsf{A}[\![\sigma]\!]} \mathsf{a}^{\langle x\rangle}\cdot\mathsf{a} \to \mathcal{A}[\![\Gamma]\!]^*$ *such that* $\kappa_{post}\left(\mathsf{a}^{\langle x\rangle}\cdot\mathsf{a}\right) = \mathsf{a}^{\langle x\rangle}\cdot\eta(v)\cdot\mathsf{a}$.

If the specification is complete then it can be easily seen that $\lceil \mathcal{F}\langle\psi\rangle(u,v)\rceil = \mathcal{F}\langle\psi\rangle(u,v)$.

The semantic definition of the effect quantifier is given using the completed form.

Definition 7.9 (Semantics of effect quantifiers)

$$\langle u,v\rangle \models_\Gamma \left\langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A_1'\}, \dots, \{A_k\}\ x : \sigma\ \{A_k'\}\right\rangle \bullet S$$

if and only if

$$\left\langle \left(u \mid x \mapsto \left\lceil \mathcal{F}\left\langle \Psi \bullet \{A_1\}\ x : \sigma\ \{A_1'\}, \dots, \{A_k\}\ x : \sigma\ \{A_k'\}\right\rangle(u,v)\right\rceil\right), (v \mid x \mapsto \epsilon)\right\rangle \models_{\Gamma, x:\sigma} S.$$

The introduction of the new quantifiers in the specification language requires a revision of the inference rules of the previous chapter. Fortunately, the revision is not substantial. We only need to redefine non-interference and the related concept of active actions. Using the new definitions, the proofs of soundness for the inference rules go through exactly as before.

Definition 7.10 (Active actions) *(Replaces Definition 5.12 on page 96)*

$$\begin{aligned}\mathcal{A}_{M,u,v} = &\left\{ \mathit{write}(\alpha)^{\langle x\rangle}, \mathit{ok}^{\langle x\rangle} \mid v(x) = \gamma_{\mathbf{var}\tau}^{x/x_1\cdots x_k} \text{ or } v(x) = \gamma_{\sigma\to\mathbf{var}\tau}^{x/x_1\cdots x_k},\ \alpha \in \mathcal{A}[\![\tau]\!],\ x \in \mathit{Free}(M)\right\} \\ &\cup \left\{ \alpha^{\langle x\rangle} \mid v(x') = \gamma_{\theta'}^{x'/x_1\cdots x\cdots x_k},\quad x' : \theta', x : \theta \in \Gamma,\ \alpha \in \mathcal{A}[\![x:\theta]\!],\ x' \in \mathit{Free}(M)\right\},\end{aligned}$$

where $\Gamma \vdash M : \theta$.

A passive trace must exclude not only all *write* actions to the stable identifiers of the phrase, but all actions of any objects denoted by identifiers that are allowed to interfere with the stable identifiers of the phrase. The concept of trace non-interference must be similarly extended.

Definition 7.11 (Trace non-interference) *(Replaces Definition 6.4 on page 109) Two sets of traces* $\mathcal{L}_0, \mathcal{L}_1$ *are said not to interfere over a frame* v, *denoted by* $\mathcal{L}_0 \#_v \mathcal{L}_1$, *if*

- *if* $v(x) = \gamma_{\sigma\to\mathbf{var}\tau}^{x/x_1\cdots x_k}$, *or* $v(x) = \gamma_{\mathbf{var}\tau}^{x/x_1\cdots x_k}, \alpha \in \mathcal{A}[\![\tau]\!]$ *then* $\left\lfloor \mathcal{L}_i \upharpoonright \mathit{write}(\alpha)^{\langle x\rangle}\right\rfloor = \emptyset$ *or* $\left\lfloor \mathcal{L}_{1-i} \upharpoonright \mathit{read}^{\langle x\rangle}\right\rfloor = \emptyset$

- *if* $v(x) = \gamma_\theta^{x/x_1\cdots x_k}$, $\alpha_0 \in \mathcal{A}\,[\![x:\theta]\!]$, $\alpha_1 \in \mathcal{A}\,[\![x_j:\theta_j]\!]$, $1 \leq j \leq k$ *then* $\left\lfloor \mathcal{L}_i \restriction \alpha_0^{\langle x \rangle} \right\rfloor = \emptyset$ *or* $\left\lfloor \mathcal{L}_{1-i} \restriction \alpha_1^{\langle x_j \rangle} \right\rfloor = \emptyset$.

The reason that the soundness of the inference rules is preserved even in the presence of the new quantifiers is, informally, that the proofs of soundness rely, basically, only on Lemma 6.3 on page 109 (Non-interfering string substitution) to insert, remove or substitute traces in or from other traces. The lemma remains true, but the proof must be updated to include new cases, corresponding to the second bullet in Definition 7.11; however the new cases are straightforward.

The actual regular languages assigned to identifiers in the environment are irrelevant as far as this lemma is concerned; only the presence or absence of interfering actions is relevant. Once we expand the definitions of non-interference, to include not only *write* actions but any interfering actions specified by the new (relative) stability quantifiers, the same proofs go through, unchanged.

Using the effect quantifier we can generalize the non-interference abstraction rule, on page 131, to a form that is almost as expressive as the original formulation in [O'H90]:

INFERENCE RULE

$$\text{ABS}\ \frac{\Psi \bullet \{A\}\,M\,\{A\} \quad \Psi' \bullet \{A'\}\,M\,\{A''\} \quad \Psi \bullet \big\langle \Psi' \bullet \{A'\}\ m{:}\sigma\ \{A''\}\big\rangle \bullet \{A\} \Rightarrow \{A_1\}\ F(m)\ \{A_2\}}{\Psi \bullet \Psi' \bullet \{A \textbf{ and } A_1\}\ F(M)\ \{A \textbf{ and } A_2\}}$$

Ψ and Ψ' share no identifiers
$\Psi \bullet \Psi' \bullet F \# A, \Psi \bullet \Psi' \bullet M \# A_2, \Psi \bullet \Psi' \bullet M \# F, \Psi \bullet \Psi' \bullet A \# A_1$ and $\Psi \bullet \Psi' \bullet A \# A_2$.

The soundness of this rule is proved similarly to the Abstraction rule (see Figure 6.2 on page 116).

7.2 Inference rules for parameterless procedures

We now provide additional rules dealing with collections of simultaneous quantifiers. For reasons having to do with syntactical coherence the quantifiers in a collection need to be introduced or eliminated simultaneously. On a semantic level, the need for simultaneous elimination can be seen going back to our example using procedure *inc*:

$$\begin{aligned}\nabla x : \textbf{expint}/inc : \textbf{comm}.\big\langle \nabla z : \textbf{expint} \bullet \{x = z\}\ inc : \textbf{comm}\ \{x = z+1\}\big\rangle \\ \bullet\ \nabla y : \textbf{expint}.\{x = y\}\ inc; inc\ \{x = y+2\}\end{aligned}$$

The following stability specification is valid: $\boldsymbol{\nabla}_{\textbf{expint}}(0)$.

However, eliminating ∇x : **expint**/*inc* : **comm** by substitution with 0 would be a mistake, because the resulting specification can only be fulfilled by instantiating *inc* as

diverge:

$$\big\langle \nabla z : \mathbf{expint} \bullet \{0 = z\}\ inc : \mathbf{comm}\ \{0 = z + 1\}\big\rangle \bullet \nabla y : \mathbf{expint}.\{0 = y\}\ inc; inc\ \{0 = y + 2\}.$$

Trying to discharge the effect quantifiers for *inc* first is even more problematic. We cannot write a procedure that *directly* increments an expression; only assignable variables can have their values changed.

The error appeared when we discharged the ∇x : **expint**/*inc* : **comm** quantifier. The idea is that x not only *might* be dependent on *inc*, but it *must* be dependent on *inc*. But *inc* cannot change the constant zero. So we must think of a collection of global quantifiers as a unit, as a "module." In our example, *inc* is the "interface" of the module and x its "abstract state." Discharging the quantifiers, which is analogous to providing an implementation for the module, must be therefore simultaneous.

The correct simultaneous discharge of the quantifiers we would like is (omitting the type information from the quantifiers, for compactness of notation):

$$\dfrac{\dfrac{\begin{array}{c}\nabla v.\nabla y.\nabla x/inc.\big\langle \nabla z \bullet \{x = z\}\ inc\ \{x = z + 1\}\big\rangle \bullet \{x = y\}\ inc; inc\ \{x = y + 2\}\\ \nabla v.\nabla y \bullet \boldsymbol{\nabla}(!v) \qquad \nabla v.\nabla y \bullet \{!v = y\}\ v := !v + 1\ \{!v = y + 1\}\end{array}}{\nabla v.\nabla y \bullet \big(\{x = y\}\ inc; inc\ \{x = y + 2\}\big)[x/!v][inc/v := !v + 1]}}{\nabla v.\nabla y \bullet \{!v = y\}\ v := !v + 1; v := !v + 1\ \{!v = y + 2\}}$$

So we "implement" the expression x using the stable global variable v and the increment operation in the obvious way. More interesting, we can use a combination of the substitution and **let** versions of quantifier elimination to verify the implementation of procedure *inc*:

$$\dfrac{\dfrac{\begin{array}{c}\nabla v.\nabla y.\nabla x/inc.\big\langle \nabla z \bullet \{x = z\}\ inc\ \{x = z + 1\}\big\rangle \bullet \{x = y\}\ inc; inc\ \{x = y + 2\}\\ \nabla v.\nabla y \bullet \boldsymbol{\nabla}(!v) \qquad \nabla v.\nabla y \bullet \{!v = y\}\ v := !v + 1\ \{!v = y + 1\}\end{array}}{\nabla v.\nabla y \bullet \big(\{x = y\}\ inc; inc\ \{x = y + 2\}\big)[x/!v][inc/v := !v + 1]}}{\nabla v.\nabla y \bullet \{!v = y\}\ \mathbf{let}\ inc\ \mathbf{be}\ v := !v + 1\ \mathbf{in}\ inc; inc\ \{!v = y + 2\}}$$

The rules for eliminating a collection of simultaneous quantifiers are the following. The elimination rules below must be applied to all quantifiers in the collection simultaneously:

Inference Rules

$$\frac{\Psi \bullet \big(\cdots \nabla x_i : \theta_1/\overline{x}_i \cdots\big) \bullet S \qquad \Psi \bullet \boldsymbol{\nabla}_{\theta_i}(P_i)}{\Psi \bullet S[x_1, \ldots, x_k/P_1, \ldots, P_k]}$$

$$\frac{\Psi \bullet \big(\cdots \big\langle \Psi' \bullet \{A_j\}\ x'_i : \sigma'_i\ \{A'_j\}\big\rangle \cdots\big) \bullet S \qquad \Psi \bullet \Psi' \bullet \{A_j\}\ x'_i\ \{A'_j\}[x_1, \ldots, x_k/P_1, \ldots, P_k]}{\Psi \bullet S[x_1, \ldots, x_k/P_1, \ldots, P_k]}$$

where $\overline{x}_i = x_{i_1} : \theta_{i_1}, \ldots, x_{i_n} : \theta_{i_n}$, $\{x_{i_1} : \theta_{i_1}, \ldots, x_{i_n} : \theta_n\} \subseteq \{x_1 : \theta_1, \ldots, x_n : \theta_n\}$
$P_i \#_{x_i} \Psi \bullet S$, $A_i \#_{x_i} \Psi \bullet S$, $A'_i \#_{x_i} \Psi \bullet S$, $\Psi \bullet \delta(A_i)$, $\Psi \bullet \delta(A'_i)$.

It is crucial that the newly substituted phrases P_i do not interfere with the rest of the specification. In addition, we need normal-termination side-conditions $\Psi \bullet \delta(A_i)$, $\Psi \bullet \delta(A'_i)$ for the assertions because, in a partial-correctness setting, Hoare triples with diverging assertions are satisfied by any phrases. Consider, for example, the following unsound elimination by substitution:

$$\frac{\nabla e.\big\langle\{\textbf{true}\}\ c\ \{\textbf{diverge}\}\big\rangle \bullet \{e=0\}\ c\ \{e=1\} \qquad \nabla e.\{\textbf{true}\}\ \textbf{skip}\ \{\textbf{diverge}\}}{\nabla e.\{e=0\}\ \textbf{skip}\ \{e=1\}}$$

The non-interference conditions $P_i\ \#_{x_i}\ \Psi \bullet S$, $A_i\ \#_{x_i}\ \Psi \bullet S$, $A'_i\ \#_{x_i}\ \Psi \bullet S$ prevent the substituted terms from interfering with the remaining stable variables of Ψ. This is necessary because the specification S and the implementations P_i may share global variables. For example, the following specification is valid:

$$\begin{aligned}\nabla v : \textbf{varint}.\nabla x : \textbf{expint}/inc : \textbf{comm}.&\big\langle\{x=y\}\ inc : \textbf{comm}\ \{x=y+1\}\big\rangle\\ &\bullet \nabla y : \textbf{expint}.\{x=y\ \textbf{and}\ !v=0\}\ inc; inc\ \{x=y+2\ \textbf{and}\ !v=0\}\end{aligned}$$

but in this case we can no longer use variable v in the implementation of inc; the following specification is not valid:

$$\nabla v : \textbf{varint}.\nabla y : \textbf{expint} \bullet \{!v=y\ \textbf{and}\ !v=0\}\ v := !v+1; v := !v+1\ \{!v=y+2\ \textbf{and}\ !v=0\}$$

PROOF OF SOUNDNESS: *(Of rule on page 144.)*
We use induction on the syntax of S. Quantifiers and connectives have straightforward proofs. Consider for example the case of the stability quantifier, $\nabla x : \theta/\overline{x} \bullet S$. Let $\mathfrak{M} = \langle u, v\rangle$ be any model consistent with Ψ. Let us denote by (Ψ_0) the collection of quantifiers being eliminated, binding identifiers $x_1, \ldots, x_k$.

$$\begin{aligned}\langle u, v\rangle &\models (\Psi_0) \bullet \nabla x : \theta/\overline{x} \bullet S\\ \langle u \diamond \Psi_0, v \diamond \Psi_0\rangle &\models \nabla x : \theta/\overline{x} \bullet S\\ \langle (u \diamond \Psi_0 \mid x \mapsto \mathcal{K}^x_\theta), (v \diamond \Psi_0 \mid x \mapsto \gamma^{x/\overline{x}}_\theta)\rangle &\models S\end{aligned}$$

If $x \notin \{x_1, \ldots, x_k\}$ then it follows that

$$\begin{aligned}\langle (u \mid x \mapsto \mathcal{K}^x_\theta) \diamond \Psi_0, (v \mid x \mapsto \gamma^{x/\overline{x}}_\theta) \diamond \Psi_0\rangle &\models S\\ \langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \gamma^{x/\overline{x}}_\theta)\rangle &\models (\Psi_0) \bullet S\end{aligned}$$

By applying the induction hypothesis, this further implies that

$$\begin{aligned}\langle (u \mid x \mapsto \mathcal{K}^x_\theta), (v \mid x \mapsto \gamma^{x/\overline{x}}_\theta)\rangle &\models S[x_1, \ldots, x_k/P_1, \ldots, P_k]\\ \langle u, v\rangle &\models \nabla x : \theta/\overline{x} \bullet S[x_1, \ldots, x_k/P_1, \ldots, P_k]\\ \langle u, v\rangle &\models (\nabla x : \theta/\overline{x} \bullet S)[x_1, \ldots, x_k/P_1, \ldots, P_k],\end{aligned}$$

because $\overline{x}$ and $x_1, \ldots, x_k$ are disjoint collections of identifiers; otherwise, the quantifier $\nabla x : \theta/\overline{x}$ would have been part of the collection (Ψ_0).

If $x = x_i \in \{x_1, \ldots, x_k\}$ then

$$S[x_1, \ldots, x_k/P_1, \ldots, P_k] = S[x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k/P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_k]$$

and we apply the same reasoning to the smaller collection of quantifiers (Ψ_0') that is like (Ψ_0) except that it does not contain x_{i-1}.

The non-trivial cases are the substitutions in Hoare triple and in stability specifications. To prove the first, we use Lemma 7.1 below. Substitution in stability assertions has similar proof. END OF PROOF.

Lemma 7.1 (Simultaneous substitutions in assertions) *For any* $\Gamma, \Gamma' \vdash A : \mathbf{assert}$*, boolean value* $\alpha \in \{tt, ff\}$*, collection of quantifiers* (Ψ)*, model* $\mathfrak{M} = \langle u, v\rangle$ *consistent with* Ψ*, if*

- $\omega \cdot (\!|A|\!)_\alpha u \cap \gamma(v) = \emptyset$,
- $\mathfrak{M} \models \Psi' \bullet \{A_k\}\ x_i\ \{A_k\}[x_1, \ldots, x_k/P_1, \ldots, P_k],\ 1 \leq i \leq k$, *where* $\langle \Psi' \bullet \cdots \{A_k\}\ x_i\ \{A_k\} \cdots \rangle$ *is a quantifier in* Ψ
- $\mathfrak{M} \models \nabla_{\theta_j}(P_j)$*, where* $\nabla x_j : \theta_j/\overline{x}_j$ *is a quantifier in* Ψ
- $P_i \ \#_{x_i,\mathfrak{M}}\ A,\ A_i\ \#_{x_i,\mathfrak{M}}\ A,\ A_i'\ \#_{x_i,\mathfrak{M}}\ A,\ \Psi \bullet \delta(A_i),\ \Psi \bullet \delta(A_i')$.

then $\omega \cdot (\!|A[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_\alpha u \cap \gamma(v) = \emptyset$.

PROOF: (*of Lemma 7.1*)
The proof is similar to that for substitution lemmas 6.2 on page 105 and 6.6 on page 111, based on an analysis of trace-level substitutions.

Let $\gamma' \stackrel{\text{def}}{=} \gamma(v) \bowtie \$^*$, where \$ is a reserved symbol, so for any sequence ω' over $\mathcal{A}[\![\Gamma]\!]$, $\omega' \in \gamma(v)$ is equivalent to $\omega' \in \gamma'$. (ω' is a string of γ' containing a shuffle with the empty iteration of $\$^*$.)

We use proof by contradiction. Suppose that $\omega \cdot (\!|A[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_\alpha u \cap \gamma(v)$ is non-empty, so there is a string $\omega \cdot \omega_0$ in this set. We look at all its substrings ω_P such that $\omega_P \in (\!|P_i|\!)_{\alpha_i}$.

We parse the string, making the following substitutions.

1. ω_P is bracketed by \$ symbols; do nothing.
2. the prefix of ω_0 up to and including ω_P is a prefix of $(\!|A|\!)_\alpha u$; do nothing.
3. the prefix of ω_0 up to and including ω_P is a prefix of $(\!|A[x_i/P_i]|\!)_\alpha u$ and x_i is bound by a stability quantifier in Ψ. Because of the non-interference conditions we can apply the non-interfering string substitution lemma, 6.3 on page 109, and replace ω_P by $\mathcal{K}^{x_i}_{\theta;\alpha_i}$ and still obtain a string in γ', because we are replacing a string which occurs in a stable way in ω_0 with another stable non-empty string.

4. the prefix of ω_0 up to and including ω_P is a prefix of $(\!|A[x_i/P_i]|\!)_\alpha u$ and x_i is bound by an effect quantifier $\langle \Psi' \bullet \cdots \{A_k\}\ P_i\ \{A_k\} \cdots \rangle$. We have two cases:

   (a) there is an assertion $A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]$ such that

   $$\omega \cdot \omega_0' \cdot (\!|A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{ff}} u \cap \gamma' = \emptyset, \tag{7.8}$$

   where ω_0' is the prefix of ω_0 up to but not including ω_P. From the semantic definition of the Hoare triple it follows that

   $$\omega \cdot \omega_0' \cdot (\!|P_i; A_k'[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{ff}} u \cap \gamma' = \emptyset,$$

   and from the normal termination condition for $A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]$,

   $$\omega \cdot \omega_0' \cdot (\!|P_i; A_k'[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{tt}} u \cap \gamma' \neq \emptyset. \tag{7.9}$$

   From the normal termination of $A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]$, equation 7.8 implies:

   $$\omega \cdot \omega_0' \cdot (\!|A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{tt}} u \cap \gamma' \neq \emptyset,$$

   which together with equation 7.9 and the non-interference substitution lemma give

   $$\omega \cdot \omega_0' \cdot (\!|A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{tt}} u \cdot (\!|P_i|\!)_{\alpha_i} u \cdot (\!|A_k'[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{tt}} u \cap \gamma' \neq \emptyset.$$

   We mark the occurrence of $(\!|P_i|\!)_{\alpha_i} u$ with the reserved symbols \$:

   $$\omega \cdot \omega_0' \cdot (\!|A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{tt}} u \cdot \$ \cdot (\!|P_i|\!)_{\alpha_i} u \cdot \$ \cdot (\!|A_k'[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{tt}} u \cap \gamma' \neq \emptyset.$$

   (b) there is no assertion $A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]$ such that

   $$\omega \cdot \omega_0' \cdot (\!|A_k[x_1, \ldots, x_k/P_1, \ldots, P_k]|\!)_{\mathit{ff}} u \cap \gamma' = \emptyset.$$

   We mark the occurrence of $(\!|P_i|\!)_{\alpha_i} u$ using the reserved symbols \$, and we also bracket it with states $\omega_A, \omega_B \in \eta(v)$.

   $$\omega \cdot \omega_0' \cdot \omega_A \cdot \$ \cdot (\!|P_i|\!)_{\alpha_i} u \cdot \$ \cdot \omega_B \cap \gamma' \neq \emptyset. \tag{7.10}$$

   States ω_A and ω_B do not interfere with $(\!|P_i|\!)_{\alpha_i}$ because the non-circularity condition ensures that no actions interfering with P_i are used in the completion of the effect quantifier binding x_i (the presence of reserved symbol \$ is irrelevant).

We repeat this substitution until the only occurrences of $(\!|P_i|\!)_{\alpha_i}u$ left are those that are either also occurring in $(\!|A|\!)_{tt}u$ or are bracketed by \$ symbols. The non-circularity of the dependence relation between identifiers (Definition 7.4 on page 140) insures that this substitution algorithm terminates, since we have a finite set of identifiers and every substitution of a term P_i only introduces substrings from those terms P'_i corresponding to identifiers x'_i dependent on x_i.

What we have as a result of this substitution is a trace in $(\!|A|\!)_\alpha u$ where all occurrences of $(\!|x_i|\!)_{\alpha_i}u$ have been replaced by strings in $\$ \cdot (\!|P_i|\!)_{\alpha_i}u \cdot \$$. We know that P_i does not interfere with A and we also know that x_i does not interfere with A. (If x_i interferes with any of the stable identifiers of A it must have been the case that those identifiers were also part of the collection (Ψ) and should have been substituted.) So we replace all $\$ \cdot (\!|P_i|\!)_{\alpha_i}u \cdot \$$ occurrences with $\mathcal{K}^{x_i}_{\theta_i;\alpha_i}$.

What we have obtained as a result of this substitution is a string in $\omega \cdot (\!|A|\!)_\alpha u \cap \gamma'$, which means that $\omega \cdot (\!|A|\!)_\alpha u \cap \gamma(v) \neq \emptyset$, which stands in contradiction with the hypothesis.

End Of Proof.

Before we proceed let us look at another example of substitution, 7.6 on page 139. Let $S \stackrel{\text{def}}{=} \{!x = 7\}\ \mathit{undo}; \mathit{do}\ \{!x = 7\}$. The inference is:

$$\dfrac{\begin{pmatrix} \nabla x/\mathit{do}, \mathit{undo} \\ \langle \nabla y \bullet \{!x = y\}\ \mathit{do}\ \{!x = y+1\}\rangle \\ \langle \nabla y \bullet \{!x = y\}\ \mathit{undo}\ \{\mathit{do}; !x = y\}\rangle \end{pmatrix} \bullet \{!x = 7\}\ \mathit{undo}; \mathit{do}\ \{!x = 7\} \qquad \nabla x.\boldsymbol{\nabla}(x) \\ \nabla x.\nabla y \bullet \{!x{=}y\}\ !x{=}!x{+}1\ \{!x{=}y{+}1\} \quad \nabla x.\nabla y \bullet \{!x{=}y\}\ !x{=}!x{-}1\ \{!x{=}!x{+}1; !x{=}y{+}1\}}{\nabla x.\{!x = 7\}\ !x = !x - 1; !x = !x + 1\ \{!x = 7\}}$$

The rules for substitution can be also used as rules for procedure definitions. From the Substitution Lemma, 4.3 on page 74, we know that

$$\textbf{let } x \textbf{ be } P \textbf{ in } P' \equiv P'[x/P],$$

so the conclusion of the elimination rule for simultaneous quantifiers can be written using **let** instead of substitution, becoming a rule for *procedure definition*:

Inference Rules (variations)

$$\begin{aligned} &\boldsymbol{\nabla}_\theta(P)[x_1, \ldots, x_k/P_1, \ldots, P_k] \equiv \\ &\qquad \boldsymbol{\nabla}_\theta(\textbf{let } x_i \textbf{ be } P_i \textbf{ in } P[x_1, \ldots x_{i-1}, x_{i+1}, \ldots, x_k/P_1, \ldots P_{i-1}, P_{i+1}, \ldots, P_k]) \\ &\{A\}\ P\ \{A'\}[x_1, \ldots, x_k/P_1, \ldots, P_k] \equiv \\ &\qquad \{A\}\ \textbf{let } x_i \textbf{ be } P_i \textbf{ in } P[x_1, \ldots x_{i-1}, x_{i+1}, \ldots, x_k/P_1, \ldots P_{i-1}, P_{i+1}, \ldots, P_k]\ \{A'\}, \end{aligned}$$

where $x_i \notin \mathit{Free}(A) \cup \mathit{Free}(A')$.

Any desirable combination of substitution and procedure definition can be used in the conclusion of the rule, as we have seen in the two examples on page 144.

7.3 Procedures with parameters

We now turn our attention to procedures with parameters, by having another look at our running example, the procedure *inc*:

$$\nabla x : \mathbf{expint}/inc : \mathbf{comm}.\big\langle \nabla z : \mathbf{expint} \bullet \{x = z\}\ inc : \mathbf{comm}\ \{x = z + 1\}\big\rangle \\ \bullet\ \nabla y : \mathbf{expint}.\{x = y\}\ inc; inc\ \{x = y + 2\}$$

The behaviour of non-local procedure *inc* is defined in terms of its effect on the non-local expression denoted by x. The behaviour of the parameterized version of *inc* should be defined, additionally, in terms of its effect on its arguments. The way we can specify a parameterized *inc* procedure is:

$$\nabla v : \mathbf{varint}.\big\langle \nabla x : \mathbf{varint}.\nabla z : \mathbf{expint} \bullet \{x = z\}\ inc(x) : \mathbf{varint} \to \mathbf{comm}\ \{x = z + 1\}\big\rangle \\ \bullet\ \nabla y : \mathbf{expint}.\{!v = y\}\ inc(v); inc(v)\ \{!v = y + 2\}, \quad (7.11)$$

The same syntactic issues of inter-dependence, simultaneity and circularity, and flattening arise. The way we deal with them is the same, by extending the definition of dependency.

Definition 7.12 (Dependencies of parameterized procedure quantifiers) *For the parameterized procedure quantifier*

$$\psi f : \theta = \left\langle \Psi \bullet \{A_1\}\ f x_1 \cdots x_m : \theta\ \{A'_1\}, \dots, \{A_k\}\ f x_1 \cdots x_m : \theta\ \{A'_k\}\right\rangle,$$

with $\theta = \sigma_1 \to \cdots \to \sigma_m \to \sigma$ and collection of quantifiers Ψ containing quantifiers $\psi_1 x_1 : \sigma_1, \cdots, \psi_m x_m : \sigma_m$, we define the semantic dependence relation as:

$$f\ \mathcal{D}\ x' \text{ if } x' \in \bigcup_{1 \le i \le k} \mathit{Free}(\Psi \bullet A_i) \cup \mathit{Free}(\Psi \bullet A'_i).$$

The definition of effect is a generalization of the parameterless procedure (Definition 7.5 on page 140):

Definition 7.13 (Effect of parameterized procedures) *For a model $\mathfrak{M}_\Gamma = \langle u, v\rangle$, the* effect associated with quantifier $\psi = \langle \Psi \bullet \{A_1\}\ f x_1 \cdots x_m : \theta\ \{A'_1\}, \dots, \{A_k\}\ f x_1 \cdots x_m : \theta\ \{A'_k\}\rangle$, *binding identifier $f : \theta$, is the regular language*

$$\mathcal{F}\langle\psi\rangle(u, v) = \sum_{1 \le i \le k} [\![f x_1 \cdots x_m]\!]\,(u \diamond \Psi)[\kappa_{pre}][\kappa_{post}] \cap \gamma(v \diamond \Psi) \restriction \bigcup_{x_0 \in |\Psi|} \mathcal{A}^{\overline{x}_0}, \qquad (7.12)$$

where $\kappa_{pre} : \sum_{\mathsf{q} \in \mathsf{Q}[\![\sigma]\!]} \mathsf{q} \cdot \mathsf{q}^{\langle f\rangle} \to \sum_{\mathsf{q} \in \mathsf{Q}[\![\sigma]\!]} \mathsf{q} \cdot ([\![A_i]\!]\,(u \diamond \Psi)) \cdot \mathsf{q}^{\langle f\rangle}$, such that $\kappa_{pre}(\mathsf{q}\cdot\mathsf{q}^{\langle f\rangle}) = \mathsf{q}\cdot(\!|A_i|\!)_{tt}(u\diamond\Psi)\cdot\mathsf{q}^{\langle f\rangle}$ and $\kappa_{post} : \sum_{\mathsf{a} \in \mathsf{A}[\![\sigma]\!]} \mathsf{a}^{\langle f\rangle}\cdot\mathsf{a} \to \sum_{\mathsf{a} \in \mathsf{A}[\![\sigma]\!]} \mathsf{a}^{\langle f\rangle}\cdot ([\![A'_i]\!]\,(u\diamond\Psi))\cdot\mathsf{a}$, such that $\kappa_{post}\left(\mathsf{a}^{\langle f\rangle} \cdot \mathsf{a}\right) = \mathsf{a}^{\langle f\rangle} \cdot (\!|A'_i|\!)_{tt}(u \diamond \Psi) \cdot \mathsf{a}$.

This is similar to the way we defined the effect of parameterless procedures.

Using this definition, in Equation 7.11 on the preceding page, the effect of *inc* is interpreted as:

$$
\begin{aligned}
&\mathcal{F}\langle \nabla x : \mathbf{varint}.\nabla z : \mathbf{expint} \bullet \{x = z\}\ inc(x) : \mathbf{varint} \to \mathbf{comm}\ \{x = z + 1\}\rangle(u, v) \\
&= \sum_n run \cdot read^{\langle x\rangle} \cdot n^{\langle x\rangle} \cdot q^{\langle z\rangle} \cdot n^{\langle z\rangle} \cdot run^{\langle inc\rangle} \cdot \Big(\sum_m read^{\langle 1inc\rangle} \cdot read^{\langle x\rangle} \cdot m^{\langle x\rangle} \cdot m^{\langle 1inc\rangle} \\
&\qquad + \sum_p write(p)^{\langle 1inc\rangle} \cdot write(p)^{\langle x\rangle} \cdot ok^{\langle p\rangle} \cdot ok^{\langle 1inc\rangle}\Big)^* \cdot done^{\langle inc\rangle} \\
&\qquad \cdot \sum_l read^{\langle x\rangle} \cdot l^{\langle x\rangle} \cdot q^{\langle z\rangle} \cdot (l+1)^{\langle z\rangle} \cdot done \cap \widetilde{\gamma^x_{\mathbf{varint}}} \cap \widetilde{\gamma^z_{\mathbf{expint}}} \restriction \mathcal{A}[\![inc : \mathbf{varint} \to \mathbf{comm}]\!] \\
&= \sum_n run \cdot run^{\langle inc\rangle} \cdot (read^{\langle 1inc\rangle} \cdot n^{\langle 1inc\rangle})^* \cdot read^{\langle 1inc\rangle} \cdot n^{\langle 1inc\rangle} \cdot write(n+1)^{\langle 1inc\rangle} \cdot ok^{\langle 1inc\rangle} \\
&\qquad \cdot (read^{\langle 1inc\rangle} \cdot (n+1)^{\langle 1inc\rangle} + write(n+1)^{\langle 1inc\rangle} \cdot ok^{\langle 1inc\rangle})^* \cdot done^{\langle inc\rangle} \cdot done.
\end{aligned}
$$

We can see that for any value n read by the procedure from its variable argument, value $n+1$ is written back to it. In addition, procedure *inc* may read or write the value $n+1$ an arbitrary number of times.

The same issue of completeness of specification arise, as in the case of parameterless procedures, and we use the same method to *complete* the specification (Definition 7.8 on page 142). The semantics of the parameterized procedure quantifier is:

Definition 7.14 (Semantics of parameterized procedure quantifier)

$$\langle u, v\rangle \models_\Gamma \big\langle \Psi \bullet \{A_1\}\ fx_1 \cdots x_m : \theta\ \{A_1'\}, \ldots, \{A_k\}\ fx_1 \cdots x_m : \theta\ \{A_k'\}\big\rangle \bullet S$$

if and only if

$$\Big\langle \Big(u \mid x \mapsto \Big\lceil \mathcal{F}\big\langle \Psi \bullet \{A_1\}\ fx_1 \cdots x_m\ \{A_1'\}, \ldots, \{A_k\}\ fx_1 \cdots x_m\ \{A_k'\}\big\rangle(u, v)\Big)\Big\rceil, (v \mid x \mapsto \epsilon)\Big\rangle \models_{\Gamma, x:\theta} S.$$

Another example of parameterized procedure specification using a generalized quantifier, slightly more sophisticated, is (removing type information for conciseness of notation):

$$\left\langle \left(\begin{array}{c} \nabla x / inc \\ \langle \nabla x0 \bullet \{x = x0\}\ inc\ \{x = x0 + 1\}\rangle \end{array} \right) \cdot \nabla e.\nabla x0 \bullet \{x = x0\}\ p\,(inc)\,(e)\ \{x = x0 + e\} \right\rangle \bullet S. \quad (7.13)$$

The types are $x, x0, e : \mathbf{expint}$, $inc : \mathbf{comm}$ and $p : \mathbf{comm} \to \mathbf{expint} \to \mathbf{comm}$.

The informal interpretation is as follows: if the effect of the argument *inc* is such that some expression x is incremented by 1 then the effect of applying procedure p to *inc* and stable expression e is to increment x from $x0$ to $x0+e$. What we are effectively specifying is that procedure p will use argument inc e times.

Formally, working through definitions 7.9 on page 142 and 7.14 on the preceding page for the semantics of effect quantifiers, the meaning assigned to inc is:

$$\mathcal{F}_1 = \sum_n run \cdot q^{\langle x \rangle} \cdot n^{\langle x \rangle} \cdot run^{\langle inc \rangle} \cdot done^{\langle inc \rangle} \cdot q^{\langle x \rangle} \cdot (n+1)^{\langle x \rangle} \cdot done,$$

and the meaning assigned to p is

$$\mathcal{F}_2 = \sum_n run \cdot run^{\langle p \rangle} \cdot \big((q^{\langle 2p \rangle} \cdot n^{\langle 2p \rangle})^* \cdot run^{\langle 1p \rangle} \cdot done^{\langle 1p \rangle}\big)^n \cdot (q^{\langle 2p \rangle} \cdot n^{\langle 2p \rangle})^* \cdot done^{\langle p \rangle} \cdot done.$$

The analysis is therefore supported by this formula, since the value n obtained by P for its second argument does indeed occur as the iteration exponent.

The elimination rule for the parameterized procedure quantifier is:

Inference Rule

$$\frac{\Psi \bullet \big(\cdots \big\langle \Psi' \bullet \{A_j\}\ f x_1 \cdots x_m : \theta\ \{A'_j\}\big\rangle,\ \cdots\big) \bullet \{A\}\ P_0\ \{A'\} \qquad \Psi \bullet \Psi' \bullet \{A_j\}\ P\ \{A'_j\}}{\Psi \bullet \{A\}\ \textbf{let}\ f\ \textbf{be}\ \lambda x_1 : \sigma_1 \cdots \lambda x_m : \sigma_m . P\ \textbf{in}\ P_0\ \{A'\}}$$

with f not free in A, A' and side-conditions $P \#_f \Psi \bullet \{A\}\ P_0\ \{A'\}$, $A_i \#_f \Psi \bullet \{A\}\ P_0\ \{A'\}$, $A'_i \#_{x_i} \Psi \bullet \{A\}\ P_0\ \{A'\}$, $\Psi \bullet \delta(A_i)$, $\Psi \bullet \delta(A'_i)$.

The motivation for the non-interference and termination side-conditions is the same as in the case of the parameter-less procedure. The proof of soundness is similar to that of Lemma 7.1 on page 146.

An example of applying the rule, in our running example of the procedure inc is:

$$\frac{\nabla v.\langle \nabla x.\nabla z.\{x = z\}\ inc(x)\ \{x = z+1\}\rangle.\nabla y.\{!v = y\}\ inc(v); inc(v)\ \{!v = y+2\} \qquad \nabla v.\nabla x.\nabla z.\{!x = z\}\ x := !x + 1\ \{!x = z+1\}}{\nabla v.\nabla y \bullet \{!v = y\}\ \textbf{let}\ inc\ \textbf{be}\ \lambda x.x := !x + 1\ \textbf{in}\ inc(v); inc(v)\ \{!v = y+2\}}$$

The non-interference condition is, obviously, that the implementation of inc cannot use variable v.

7.4 Temporal style specifications

Example 7.13 on the preceding page, specifying a procedure that uses its first argument a number of times equal to the value of its second argument, seems to validate Reynolds's observation that when we move from well-understood, mathematically-oriented programming such as integer arithmetic, to more arbitrary programming tasks, "the heart of the problem moves from meeting the specification to formulating it" [Rey98]. Indeed, a simple property is specified in a rather awkward way which is frustratingly indirect. We need to rely on the possible *effect* of an argument to a procedure to specify what is an essentially *temporal* property, having to do with the way in which a procedure sequences its arguments. Such properties are most naturally expressed using temporal logics or regular languages, two formalisms with similar powers of expressiveness (see for example [CGP99, Chapter 9]).

In this section we will look at a more direct specification formalism. The approach used here was first proposed by Abramsky in [Abr01, Section 4.1]. Here we will develop it and integrate it with the existing specification and verification formalism.

The idea is that temporal-style properties of a program M, such as "variable x is written to, then procedure p is executed" for a program

$$\Gamma, x : \mathbf{varint}, p : \mathbf{comm} \vdash M : \mathbf{comm}$$

can be expressed directly as an inclusion constraint between regular languages:

$$[\![M]\!]\, u_{\Gamma,x:\mathbf{varint},p:\mathbf{comm}} \subseteq \Big(\sum_n \mathit{write}(n)^{\langle x\rangle} \cdot \mathit{ok}^{\langle x\rangle} \cdot \mathit{run}^{\langle p\rangle} \cdot \mathit{done}^{\langle p\rangle}\Big)^{\sim},$$

where $R^{\sim} \stackrel{\mathrm{def}}{=} \widetilde{R}$, and the broadening context is $\mathcal{A}\,[\![\Gamma]\!]$.

A more sophisticated property for a program M of the same type is "variable x is written to before procedure p is executed", with the regular-language specification

$$[\![M]\!]\, u_{\Gamma,x:\mathbf{varint},p:\mathbf{comm}} \subseteq \Big(\big(\sum_n \mathit{write}(n)^{\langle x\rangle} \cdot \mathit{ok}^{\langle x\rangle}\big)^{+} \cdot \mathit{run}^{\langle p\rangle} \cdot \mathit{done}^{\langle p\rangle} \cdot \big(\sum_n \mathit{write}(n)^{\langle x\rangle} \cdot \mathit{ok}^{\langle x\rangle} + \mathit{run}^{\langle p\rangle} \cdot \mathit{done}^{\langle p\rangle}\big)^{*}\Big)^{\sim}.$$

Using this style of specification, procedure p in example 7.13 on page 150 can be specified directly using regular language $\mathcal{F}_2$ as its specification.

We extend the specification language with the following generalized quantifier:

TYPING RULE

$$\frac{\Gamma, f : \sigma_1 \to \cdots \to \sigma_k \to \sigma \vdash S : \mathbf{spec}}{\Gamma \vdash \langle f(x_1 : \sigma_1) \cdots (x_k : \sigma_k) : \sigma \subseteq R\rangle \bullet S : \mathbf{spec}}$$

where R is a regular expression over the alphabet

$$\begin{aligned}\mathcal{A} =&\{x \mid x : \mathbf{comm} \in \Gamma \text{ or } x = x_i \text{ and } \sigma_i = \mathbf{comm}\}\\ &\cup\{x_v \mid x : \mathbf{exp}\tau \in \Gamma \text{ or } x = x_i \text{ and } \sigma_i = \mathbf{exp}\tau, v \in \mathcal{A}\,[\![\tau]\!]\}\\ &\cup\{x_{\mathsf{r}(v)} \mid x : \mathbf{var}\tau \in \Gamma \text{ or } x = x_i \text{ and } \sigma_i = \mathbf{var}\tau, v \in \mathcal{A}\,[\![\tau]\!]\}\\ &\cup\{x_{\mathsf{w}(v)} \mid x : \mathbf{var}\tau \in \Gamma \text{ or } x = x_i \text{ and } \sigma_i = \mathbf{var}\tau, v \in \mathcal{A}\,[\![\tau]\!]\}\end{aligned}$$

having the form:

- $R \triangleright \star$ if $\sigma = \mathbf{comm}$;
- $\sum_{v \in \mathcal{A}[\![\tau]\!]} R_v \triangleright v$ if $\sigma = \mathbf{exp}\tau$;
- $\sum_{v \in \mathcal{A}[\![\tau]\!]} R_{\mathsf{r}(v)} \triangleright \mathsf{r}(v) + \sum_{v \in \mathcal{A}[\![\tau]\!]} R_{\mathsf{w}(v)} \triangleright \mathsf{w}(v)$ if $\sigma = \mathbf{var}\tau$.

The identifiers used in R can be either the global identifiers of Γ or the parameters x_i. The $\triangleright$ notation is used in order to identify the type of the regular expression defining the quantifier, as well as associating the temporal properties with each value produced by the non-local object denoted by the quantifier.

Informally, the semantics are:

- x: command x is executed,
- x_v: expression x produces value v,
- $x_{\mathsf{r}(v)}$: variable x produces value v,
- $x_{\mathsf{w}(v)}$: variable x is assigned value v,
- $R \triangleright \star$: execution of command-like f completes,
- $R_v \triangleright v$: evaluation of expression-like f produces v,
- $R_{\mathsf{r}(v)} \triangleright \mathsf{r}(v)$: reading from variable-like f produces v,
- $R_{\mathsf{w}(v)} \triangleright \mathsf{w}(v)$: variable-like f is assigned v.

For example, procedure p in example 7.13 on page 150 can be described by the quantifier:

$$\left\langle p(x : \mathbf{comm})(e : \mathbf{expint}) : \mathbf{comm} \subseteq \sum_n \underbrace{e_n^* \cdot x \cdots e_n^* \cdot x}_{n \text{ times}} \cdot e_n^* \triangleright \star \right\rangle \bullet S. \tag{7.14}$$

The semantics of the temporal specification quantifier is given directly by the regular expression in the quantifier.

Definition 7.15 (Semantics of temporal specification quantifiers)

$$\langle u, v \rangle \models_\Gamma \langle f(x_1 : \sigma_1) \cdots (x_k : \sigma_k) : \sigma \subseteq R \rangle \bullet S : \mathbf{spec}$$

if and only if

$$\langle (u \mid f \mapsto R_\sigma[\alpha^{\langle x_i \rangle}/\alpha^{\langle i \rangle}]), (v \mid f \mapsto \epsilon) \rangle \models_{\Gamma, f:\theta} S,$$

where the regular expressions R_σ are defined as

- $R_{\mathbf{comm}} = \mathit{run} \cdot \mathit{run}^{\langle f \rangle} \cdot R'_\star \cdot \mathit{done}^{\langle f \rangle} \cdot \mathit{done}$
- $R_{\mathbf{exp}\tau} = \sum_{v \in \mathcal{A}[\![\tau]\!]} q \cdot q^{\langle f \rangle} \cdot R'_v \cdot v^{\langle f \rangle} \cdot v$

- $R_{\mathbf{var}\tau} = \sum_{v\in\mathcal{A}[\![\tau]\!]} \mathit{read}\cdot\mathit{read}^{\langle f\rangle}\cdot R'_{\mathsf{r}(v)}\cdot v^{\langle f\rangle}\cdot v$
$+\sum_{v\in\mathcal{A}[\![\tau]\!]} \mathit{write}(v)\cdot\mathit{write}(v)^{\langle f\rangle}\cdot R'_{\mathsf{w}(v)}\cdot ok^{\langle f\rangle}\cdot ok,$

with R'_α above defined as

$$R'_\alpha = R_\alpha[x/\mathit{run}^{\langle x\rangle}\cdot\mathit{done}^{\langle x\rangle}][x_v/q^{\langle x\rangle}\cdot v^{\langle x\rangle}][x_{\mathsf{r}(v)}/\mathit{read}^{\langle x\rangle}\cdot v^{\langle x\rangle}][x_{\mathsf{w}(v)}/\mathit{write}(v)^{\langle x\rangle}\cdot ok^{\langle x\rangle}].$$

Using the semantic definition above we can easily verify that the specifications in example 7.13 on page 150 and in example 7.14 on the page before are indeed equivalent.

We do not give a logical rule for discharge of the temporal specification quantifier, because we cannot formulate temporal predicates in the logic. We just give a semantical rule. This discharge rule is reflected by the following proposition:

Proposition 7.1 (Temporal specification quantifier discharge)

$$\begin{aligned}
\mathit{If}\ &\langle u,v\rangle \models_\Gamma \langle f(x_1:\sigma_1)\cdots(x_k:\sigma_k):\sigma\subseteq R\rangle\bullet\{A\}\ P_0\ \{A'\},\\
&\quad f\notin \mathit{Free}(A)\cup\mathit{Free}(A'),\\
\mathit{and}\ &[\![P]\!]\,u_1\cap\gamma(v_1)\subseteq\widetilde{R_\sigma},\ \mathcal{A}=\mathcal{A}[\![\Gamma]\!]\,,\ \mathit{with}\\
&\quad x_i\notin\mathit{Free}(A)\cup\mathit{Free}(P_0)\cup\mathit{Free}(A'),\\
&\quad u_1=(u\mid x_1\mapsto\mathcal{K}^{x_1}_{\sigma_1}\mid\cdots\mid x_1\mapsto\mathcal{K}^{x_k}_{\sigma_k})\\
&\quad v_1=(v\mid x_1\mapsto\epsilon\mid\cdots\mid x_k\mapsto\epsilon)
\end{aligned}$$

and for all $\omega\in[\![P]\!]\,u_1\cap\gamma(v_1)\restriction\lfloor R_\sigma\rfloor$, $\omega\ \#_v\ [\![P_0;A']\!]\,u\cap\gamma(v)$ *then*

$$\langle u,v\rangle\models\{A\}\ \mathbf{let}\ f\ \mathbf{be}\ \lambda x_1:\sigma_1\cdots\lambda x_k:\sigma_k.P\ \mathbf{in}\ P_0\ \{A'\}$$

The non-interference condition requires that no action of P which is not already behaviourally captured by regular expression R may interfere with P_0 or A'.
PROOF:
Without loss of generality, let us assume P and P_0 are in **let**-free, β-normal form.

Let $\omega\in\eta(v)$ be a state such that $\omega\cdot(\!|A|\!)_{\mathit{ff}}\,u\cap\gamma(v)=\emptyset$. Then, it follows that

$$\omega\cdot(\!|P_0;A'|\!)_{\mathit{ff}}\,(u\mid f\mapsto R_\sigma)\cap\gamma(v\mid f\mapsto\epsilon)=\emptyset,$$

because f is not free in A, which further implies that

$$\omega\cdot(\!|P_0;A'|\!)_{\mathit{ff}}\,(u\mid f\mapsto\mathcal{K}^f_\theta)[\kappa]\cap\gamma(v\mid f\mapsto\epsilon)=\emptyset, \tag{7.15}$$

where $\kappa(\omega)=R'_{\mathsf{a}}[\kappa']$ for all $\omega\in\left(\mathsf{q}^{\langle f\rangle}\cdot\omega_0\cdot\mathsf{a}^{\langle f\rangle}[\kappa']\right)$ and $\kappa'\left(\mathsf{q}^{\langle i\rangle}\cdot\mathsf{a}^{\langle i\rangle}_j\right)=(\!|M_j|\!)_{\mathsf{a}_j}u$, with $\theta=\overline{\sigma}\to\sigma$.

The substitution is made for all instances of $fM_1\cdots M_k$ occurring in $P;A'$. The above follows from the semantic definition of function application. The assumption that P_0 is in **let**-free, β-normal form ensures that identifier f can only occur in applications. But an immediate consequence of the premise is $(\!|P|\!)_{\mathsf{a}}u_1\cap\gamma(v_1)\subseteq\widetilde{R'_{\mathsf{a}}}$.

This and the noninterference condition implies that every string in $(\!|P|\!)_{\mathsf{a}}u_1 \cap \gamma(v_1)$ is a string in R'_{a} shuffled with actions that do not interfere with P_0 under v. From the non-interference string substitution lemma (Lemma 6.3 on page 109) we know that traces consisting of non-interfering actions can be introduced and eliminated arbitrarily, therefore this together with equation 7.15 implies that

$$\omega \cdot (\!|P_0; A'|\!)_{\mathit{ff}}(u \mid f \mapsto \mathcal{K}^f_\theta)[\kappa''] \cap \gamma(v \mid f \mapsto \epsilon) = \emptyset, \tag{7.16}$$

where $\kappa''(\omega) = (\!|P|\!)_{\mathsf{a}}u[\kappa']$ for all $\omega \in \big(\mathsf{q}^{\langle f \rangle} \cdot \omega_0 \cdot \mathsf{a}^{\langle f \rangle}[\kappa']\big)$.

This is semantically equivalent to

$$\omega \cdot (\!|(\mathbf{let}\ f\ \mathbf{be}\ \lambda x_1 : \sigma_1 \cdots \lambda x_k : \sigma_k . P\ \mathbf{in}\ P_0; A')|\!)_{\mathit{ff}}u \cap \gamma(v) = \emptyset.$$

Since f is not free in A', this is further equivalent to

$$\omega \cdot (\!|(\mathbf{let}\ f\ \mathbf{be}\ \lambda x_1 : \sigma_1 \cdots \lambda x_k : \sigma_k . P\ \mathbf{in}\ P_0); A'|\!)_{\mathit{ff}}u \cap \gamma(v) = \emptyset,$$

which is what we are required to prove. END OF PROOF.

7.5 Stability and non-interference revisited

It is quite obvious that the addition of the new quantifiers to the specification language (relativized stability, effect, temporal) does not change the decidability property for the model-checking problem (Theorem 5.1 on page 97). Also, the inference rules for relativized stability and effect quantifier are sound, therefore the soundness theorem (Theorem 6.1 on page 101) remains true, subject to the revised definitions of passivity and non-interference (Definitions 7.10 and 7.11 on page 142).

One issue that needs to be revisited in the presence of the generalized quantifiers is that of syntactic characterization of non-interference. Definition 6.9 on page 133, of the set of modified variables, needs to be updated to deal with the new quantifiers. In the semantic context of the original definition we could assume that the environment of a term contained only copy-cat regular languages, so variables could be modified by explicit assignment only. But, with the addition of the new quantifier, the environment of a term depends on the quantifiers binding the variables of the term.

The sources of interference are now more numerous:

- all identifiers in X interfere with x if it is bound by the relativized stability quantifier $\nabla x : \theta / X$;
- x interferes with all free identifiers written to in A and A' if it is bound by the effect quantifier $\big\langle \psi \bullet \{A\}\ x\ \{A'\} \big\rangle$;
- f interferes with identifier x' if it is bound by an effect quantifier $\langle f(x_1) \cdots (x_k) \subseteq R \rangle$ and R contains the symbol $x'_{\mathsf{w}(\alpha)}$.

In addition, if an identifier x occurs free in a phrase P then P will interfere with all variables that x interferes with.

Definition 7.16 (Interference set) *We define the set of identifiers interfered with by x, denoted by $Mod_\Psi(x)$ as the smallest set with the properties that:*

- *if $\nabla x' : \theta / \cdots x : \sigma \cdots$ is in Ψ then $x' \in Mod_\Psi(x)$;*
- *if $\langle \Psi' \bullet \cdots \{A\}\ x'\ \{A'\} \rangle$ is in Ψ then*
  - $\big(Mod_\Gamma(A) \cup Mod_\Gamma(A')\big) \setminus Bound(\Psi') \subseteq Mod_\Psi(x)$;
  - $\big(\bigcup_{x' \in Free(A) \cup Free(A')} Mod_{\Psi.\Psi'}(x')\big) \setminus Bound(\Psi') \subseteq Mod_\Psi(x)$;
- *if $\langle x(x_1) \cdots (x_k) \subseteq R \rangle$ is in Ψ then*
  - $\{x' \mid x'_{\mathsf{w}(\alpha)} \in R\} \subseteq Mod_\Psi(x)$;
  - $\big(\bigcup_{x' \in R} Mod_\Psi(x')\big) \setminus Bound(\Psi') \subseteq Mod_\Psi(x)$;

where $Bound(\Psi)$ is the set of identifiers bound by the quantifiers in the collection Ψ.

The syntactic characterization of non-interference lemma (6.8 on page 134) becomes:

Lemma 7.2 (Syntactic characterization of non-interference) *If X is the set of all identifiers bound by $\boldsymbol{\nabla}$ in Ψ and*

$$
\begin{aligned}
&\big(\bigcup_{x \in Free(P)} Mod_\Psi(x) \cup Mod(P)\big) \cap Free(P') \cap X \\
=&\big(\bigcup_{x \in Free(P')} Mod_\Psi(x) \cup Mod(P')\big) \cap Free(P) \cap X \\
=&\emptyset.
\end{aligned}
$$

then $\Psi \bullet P \ \#\ P'$.

Proof: Directly from the definitions; the meanings of P, P' in $\langle u \diamond \Psi, v \diamond \Psi \rangle$ contain no actions that interfere with the other phrase. End Of Proof.

The syntactic characterization of normal termination is not changed by the relativized stability quantifier or by the effect quantifier. The former maps a term to a copy-cat expression in the environment, non-empty, and the latter maps a term to a non-empty regular expression because regular expressions are completed (Definition 7.8 on page 142). Only a temporal-style quantifier may map an identifier to the divergent computation, if the regular expression R in the quantifier denotes the empty language.

Chapter 8

Conclusion

The main contribution of this dissertation is the creation of a software-specification framework compatible with both model checking and inferential reasoning. As such, it provides a suitable framework for *compositional model checking.* The programming language used, IA, is quite realistic. It contains both imperative (assignment, branching, iteration) and functional (abstraction, application) features. In addition, the language is one of *active expressions*, a feature pervasive in "real-life" programming languages, yet traditionally considered to be incompatible with reasoning about program correctness.

The foundation on which we build our specification language is Hoare's logic, and we support (versions of) its axioms. Our model-checking-friendly specification language is also based on an *abstract-model* style of specification. The idea of such a style of specification is not new, but the formalization used here, based on generalized (or, rather, specialized) quantifiers is a first effort meant to incorporate such specifications in a programming logic. To reconcile side effects in expressions with mathematical reasoning we use the novel, games-inspired, technique of imposing *global* constraints on the behaviour of a non-local object; these constraints are imposed using quantifiers, which we call "stability" quantifiers. We also take a first step towards incorporating temporal-style specifications, very common in model-checking, in the specification language. At this point we only incorporate such specifications semantically, but in a way which is still compatible with compositional verification.

The logical rules of this specification language provide some new insight into the phenomenon of *interference*, which is given a simple and natural explanation in terms of trace interactions. An approximate syntactical characterization of interference also follows naturally from the semantics of the programming and specification languages. We use a *symmetrical* interpretation of non-interference $\Psi \bullet M \# M'$, i.e. phrase M does not interfere with M' and *vice versa.* Symmetrical non-interference is necessary in substitution, when the traces of one phrase is *embedded* an arbitrary number of times in the traces of the other, but it is not necessary in inference rules for programming constructs, which require only a *concatenation* of traces. For example, in the rule for IF, page 128, the symmetrical non-interference condition $\Psi \bullet B \# M_i$ is too strong. Only B must not interfere with M_i; it is irrelevant whether M_i interferes with B, because

M_i only occurs after B in computation. Tighter specifications can be obtained by using asymmetrical non-interference conditions where appropriate.

The work presented here, however, merely scratches the surface of many important semantic and logical issues encountered. The main objective of the work is to provide the semantic and logical framework for compositional software model checking, and it achieves that. But from several points of view the solution we provide must be studied further.

The first issue that needs better elucidation is that of the semantic proof techniques we use. The work-horse of many of the proofs is the non-interfering string substitution lemma (6.3 on page 109), and the proofs require a quite meticulous, often tortuous, low-level analysis of trace-level string substitutions. This is why many of the semantic-level proofs are not as formalized as we would like them to be. The lack of elegant, or at least perspicuous, proof techniques is, arguably, the most serious problem that besets game semantics. The development of more algebraic proof techniques should be a high priority for game semanticists.

The second issue that deserves a separate and more abstract treatment is the proof theory of generalized quantifiers. Several works, most notably of Hintikka and of van Lambalgen and Alechina, have provided very important syntactical and logical clues in tackling this difficult problem. But I was unable to adopt and adapt their formalisms directly to the task at hand.

The third interesting issue which must be further developed is that of *global constraints*. In our specification framework we look at stability and relativized stability as useful such global constraints. But, in addition to the two kinds of stability we study, are there any other interesting and viable kinds of global constraints? For example, one can think of specifying a global counter using a global constraint. Semantically, this is a rather simple proposition. However, the logical properties of such generalized global constraints are less clear.

The fourth issue, of more immediate practical consequence, is total correctness. A phrase specified by a Hoare triple is totally correct if, in addition to not producing *false* in the post-condition, it is also guaranteed not to diverge. The semantics of the total-correctness Hoare triple is straightforward:

Definition 8.1 (Hoare triple, total correctness semantics)

$$
\begin{aligned}
&\langle u, v\rangle \models_\Gamma [A]M[A'] \quad \textit{if and only if} \\
&\qquad \textit{for all } \omega \in \eta(v), (\omega \cdot [\![A]\!]\, u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt \quad \textit{implies} \quad (\omega \cdot [\![M; A']\!]\, u) \cap \gamma(v) \subseteq \mathcal{A}^* \cdot tt, \\
&\qquad\qquad \textit{and } (\omega \cdot [\![M; A']\!]\, u) \cap \gamma(v) \neq \emptyset.
\end{aligned}
$$

But the real problem lies in the logical properties of this definition, which is not *algebraic*: substitution of a non-diverging term in a non-diverging term can lead to a diverging term:

$$
\begin{aligned}
&S \stackrel{\text{def}}{=} \nabla x.\forall y \bullet [\textbf{if } x = 3 \textbf{ then } y; \textbf{true else diverge}] \\
&C \stackrel{\text{def}}{=} \textbf{if } x \neq 3 \textbf{ then diverge else skip}
\end{aligned}
$$

In the above, S is a totally correct specification, as it always produces the value **true**. Phrase C is a non-diverging, non-interfering phrase. But substitution of C for y in S gives a diverging phrase, because C needs x to be 3 in order to prevent divergence, while S requires $x \neq 3$. The result of the substitution is therefore not totally correct. Handling such state-dependent substitutions logically might be possible, but likely too complicated to be of any use. There is an alternative approach to making divergence specifications algebraic, by strengthening the intuitive interpretation of normal termination from "sometimes terminating" to "always terminating," so the sometimes terminating phrases used in the example above are not considered normally terminating in the first place. The technical details, however, need to be carefully considered. Note that the same difficulties with total correctness arise in the presence of non-determinism, even in languages without side-effects in expressions; phrases such as **if random**$=0$ **then diverge** raise similar problems.

The fifth issue that warrants further development is the *logical* integration of temporal-style specifications. In order to achieve this, the specification language must be extended with temporal operators, the properties of which must also be studied in the semantic and logical context of the existing specification language. Even though, for model-checking purposes, the lack of logical integration is not very problematic (semantic-level quantifier elimination rule, Proposition 7.1 on page 154, supports compositional reasoning), such an integration would quite possibly open the door to a more unified logical approach.

Our approach is based on an informal argument that the "classical" style of specification, based on stronger notions of universal quantifier and implication, is not model-checking friendly. However, we do not have a negative result showing this to be impossible. Further investigations to clarify this issue should be worthwhile.[1]

Obviously, another important issue that should be addressed is expanding the programming and specification languages with new features to improve usability and expressivity. Adapting this logical framework to call-by-value and using the regular-language semantics of that language should be fairly straightforward. Some of the logical overhead, for example ground-type stability, becomes unnecessary and the semantics of first-order stability should be simpler. Introducing non-determinism is, again, fairly straightforward and should only require the revising of only some of the stability axioms, as the new **random** constant would not be stable. A much more substantial revision of the semantic framework would be required by the introduction of parallelism or concurrency. The copy-cat interpretation of free function identifiers is obviously no longer adequate, as the arguments can be evaluated in parallel. Also, many of the proofs rely on the sequential nature of the operators and traces. However, the definition of non-interference we use is strong enough to handle parallelism and concurrency, because it is quite strict. Strengthening the type system of the language, to control stability and interference, similar to Syntactic Control of Interference, should also be interesting and useful.

Finally, perhaps the most important issue is that of designing a programming verification tool based on our approach, supporting compositional model checking. The decidability of the model-checking problem and the underlying algorithmic semantics

[1] The author has been recently pursuing this topic [Ghi02].

suggest this should be possible, but it is not a guarantee. Practical model-checking experience suggests that a model-checking tool may be useful even though the model-checking algorithms are not always terminating and, conversely, that model-checking tools based on theoretically decidable properties can still be impractical. Only implementation and practical experimentation can answer the question of usefulness.

Bibliography

[Abr96] S. Abramsky. Semantics of interaction. In *Trees in Algebra and Programming – CAAP'96, Proc. 21st Int. Coll., Linköping*, volume 1059, page 1. Springer-Verlag, 1996.

[Abr01] S. Abramsky. Algorithmic game semantics. Lecture notes, Marktoberdorf International Summer School 2001. (available from `http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/`), 2001.

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, May 1994.

[AHM98] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998.

[AJ92] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. In *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, New Delhi, 1992. Springer-Verlag. Also Imperial College Report DoC 92/24.

[AM] S. Abramsky and G. McCusker. Game semantics. Lecture notes, 1997 Marktoberdorf summer school (available from `http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/mdorf97.ps.gz`).

[AM96] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions (extended abstract). In *Proceedings of 1996 Workshop on Linear Logic*, volume 3 of *Electronic notes in Theoretical Computer Science*. Elsevier, 1996. Also as Chapter 20 of [OT97].

[AM98] S. Abramsky and G. McCusker. Call-by-value games. In *CSL: 11th Workshop on Computer Science Logic*, volume 1414 of *LNCS*, pages 1–17, 1998.

[AM99] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science*, 227:3–42, 1999.

[AMJ94] S. Abramsky, P. Malacaria, and R. Jagadeesan. *Full Abstraction for PCF*, volume 789 of *Lecture Notes in Computer Science*, pages 1–59. Springer-Verlag, April 1994.

[AvL96] N. Alechina and M. van Lambalgen. Generalized quantification as substructural logic. *The Journal of Symbolic Logic*, 61(3):1006–1044, September 1996.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[Bla92] A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:183–220, 1992. Special Volume dedicated to the memory of John Myhill.

[BM+95] S. Brookes, M. Main, A. Melton, and M. Mislove, editors. *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*, Tulane University, New Orleans, Louisiana, March 29–April 1 1995. Elsevier Science.

[BMR93] A. Borgida, J. Mylopoulos, and R. Reiter. And nothing else changes: the frame problem in procedure specifications. In *International Conference on Software Engineering 1993*. ACM Press, May 1993.

[Boe82] H. Boehm. A logic for expressions with side effects. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 268–280. ACM, January 1982.

[Boe85] H. Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions On Programming Languages And Systems*, 7(4):637–655, October 1985.

[BR01] T. Ball and S. K. Rajamani. The SLAM toolkit. In *13th Conference on Computer Aided Verification (CAV'01)*, July 2001. Available at `http://research.microsoft.com/slam/`.

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In *Proceedings, 8th Annual IEEE Symposium on Logic in Computer Science*, pages 98–109, Montreal, Canada, 1993. IEEE Computer Society Press, Los Alamitos, California. Published also as Chapter 21 of [OT97].

[Bru72] N. Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic manipulation, with application to the church-rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Bru79] N. Bruijn. Lambda calculus notation with namefree formulas involving symbols that represent reference transforming mappings. *Indag. Math.*, 40(3):348–356, 1979.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall International, London, 1990.

[CD+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, and H. Zheng. Bandera. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM Press, June 2000.

[CGJ98] C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 345–357, Montreal, Canada, June 1998.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Con76] J. H. Conway. *On Numbers and Games*. Academic Press, London, 1976.

[CW+96] E. M. Clarke, J. M. Wing, R. Alur, et al. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[dAH01] L. de Alfaro and T. A. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, September 10–14 2001. ACM Press.

[dBdBZ80] J. W. de Bakker, A. de Bruin, and J. I. Zucker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, London, 1980.

[DT96] M. Dorfman and R. H. Thayer. *Software Engineerings*. Computer Society Press, 1996. Abridged online version at `http://www.dacs.dtic.mil/techs/fmreview/toc.html`.

[FJ+96] M. P. Fiore, A. Jung, E. Moggi, et al. Domains and denotational semantics: history, accomplishments and open problems. *Bulletin of the European Association for Theoretical Computer Science*, (59):227–256, 1996. Also available at `http://www.dcs.qmw.ac.uk/~ohearn/papers.html`.

[Ghi01a] D. R. Ghica. Regular language semantics for a call-by-value programming language. In *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, pages 85–98, Aarhus, Denmark, May 2001. Elsevier.

[Ghi01b] D. R. Ghica. A regular-language model for Hoare-style correctness statements. In *Proceedings of the Verification and Computational Logic 2001 Workshop*, Florence, Italy, August 2001.

[Ghi02] D. R. Ghica. The hyperfine semantics of specification logic. Technical report, Oxford University Computing Laboratory, 2002. To appear.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[GL80] D. Gries and G. Levin. Assignment and procedure call proof rules. *ACM Trans. on Programming Languages and Systems*, 2(4):564–579, 1980.

[GM] D. R. Ghica and G. McCukser. The regular-language semantics of first-order Idealized ALGOL. *Theoretical Computer Science* (accepted for publication).

[GM00] D. R. Ghica and G. McCusker. Reasoning about Idealized ALGOL using regular languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000*, volume 1853 of *LNCS*, pages 103–116. Springer-Verlag, 2000.

[Har79] D. Harel. *First-order dynamic logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1979. Rev. version of the author's thesis, M.I.T., 1978.

[Hin96] J. Hintikka. *The Principles of Mathematics Revisited.* Cambridge University Press, 1996.

[HLV96] L. Hella, K. Luosto, and J. Väänänen. The hierarchy theorem for generalized quantifiers. *The Journal of Symbolic Logic*, 61(3):802–817, September 1996.

[HM98] C. Hankin and P. Malacaria. A new approach to control flow analysis. *Lecture Notes in Computer Science*, 1383, 1998.

[HM99] R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 422–430, Washington - Brussels - Tokyo, July 1999. IEEE.

[HO00] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163(8), December 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580 and 583, 1969.

[Hoa71] C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer-Verlag, Berlin, 1971.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hol97] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Available at `http://netlib.bell-labs.com/netlib/spin/`.

[Jac00] D. Jackson. Enforcing design constraints with object logic. In J. Palsberg, editor, *SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2000.

[Jan85] M. Jantzen. Extending regular expressions with iterated shuffle. *Theoretical Computer Science*, 38(2-3):223–247, June 1985.

[Jür] J. Jürjens. Games in the semantics of programming languages. *Synthese (Elsevier)*. To be published.

[Kur97] R. P. Kurshan. Formal verification in a commercial setting. In *Proceedings of the Design Automation Conference*, pages 258–262, Anaheim, California, June 1997.

[Lai97] J. Laird. Full abstraction for functional languages with control. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 58–67, Warsaw, Poland, 29 June–2 July 1997. IEEE Computer Society Press.

[Lai01] J. Laird. A games semantics for idealized CSP. In *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, Electronic notes in Theoretical Computer Science, pages 157–176, Aarhus, Denmark, May 2001. Elsevier.

[Lin66] P. Lindström. First order predicate logic with generalized quantifiers. *Theoria*, 32:186–195, 1966.

[Lor60] P. Lorenzen. Logik und agon. In *Atti del Congresso Internazionale di Filosofia*, pages 187–194, 1960.

[McC] G. McCusker. A graph model for imperative computation. Invited lecture at Category Theory in Computer Science 2002, available at `http://www.cogs.susx.ac.uk/users/guym/papers/imperative-graph.pdf`.

[McC97] G. McCusker. Games and definability for FPC. *Bulletin of Symbolic Logic*, 3(3):347–362, September 1997.

[McC98] G. McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. Distinguished Dissertations. Springer-Verlag Limited, 1998.

[McC02] G. McCusker. A fully abstract relational model of Syntactic Control of Interference. In *16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK*, volume 2471 of *LNCS*, pages 247–262. Springer-Verlag, 2002.

[Mos57] A. Mostowski. On a generalization of quantifiers. *Fundamenta Mathematicæ*, XLIV:12–36, 1957.

[MS88] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 191–203, San Diego, California, 1988. ACM, New York. Reprinted as Chapter 7 of [OT97].

[Nas50] J. F. Nash. Equilibrium points in n-person games. In *Proceedings of the National Academy of Sciences of the United States of America*, pages 48–49, 1950.

[NB+63] P. Naur, J. W. Backus, et al. Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6(1):1–17, 1963. Also *The Computer Journal* 5:349–67, and *Numerische Mathematik* 4:420–53.

[Nic94] H. Nickau. Hereditarily sequential functionals. *Lecture Notes in Computer Science*, 813, 1994.

[O'H90] P. W. O'Hearn. *The Semantics of Non-Interference: A Natural Approach.* Ph.D. thesis, Queen's University, Kingston, Canada, 1990.

[Old84] E. Olderog. Correctness of programs with PASCAL-like procedures without global variables. *Theoretical Computer Science*, 30:49–90, 1984.

[Ole82] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages.* Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.

[Ong02] C.-H. L. Ong. Observational equivalence of third-order Idealized Algol is decidable. In *Proceedings of IEEE Symposium on Logic in Computer Science, 2002*, pages 245–256, July 2002.

[OP+95] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In Brookes et al. [BM+95].

[OP+99] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228:175–210, 1999. Preliminary version reprinted as Chapter 18 of [OT97].

[OR95a] P. W. O'Hearn and U. S. Reddy. Objects, interference and the Yoneda embedding. In Brookes et al. [BM+95].

[OR95b] P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF. *Information and Computation*, 120(1):107–116, July 1995.

[OR00] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the Association for Computing Machinery*, 47(1):167–223, January 2000.

[ORY01] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'01)*, volume 2142 of *LNCS*, pages 1–19, 2001.

[OT93a] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–184, Charleston, South Carolina, 1993. ACM, New York. A version also published as Chapter 16 of [OT97].

[OT93b] P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, 2. *Information and Computation*, 107(1):25–57, 1993. Published also as Chapter 19 of [OT97].

[OT97] P. W. O'Hearn and R. D. Tennent, editors. ALGOL-*like Languages*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1997. Two volumes.

[Par68] D. Park. Some semantics for data structures. In D. Michie, editor, *Machine Intelligence 3*, pages 351–71. American Elsevier, New York, 1968.

[Pit96] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *11th Annual Symposium on Logic in Computer Science*, pages 152–163. IEEE Computer Society Press, Washington, 1996. A version also published as Chapter 17 of [OT97].

[Pla66] R. A. Platek. *Foundations of Recursion Theory.* PhD thesis, Stanford, 1966.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Red96] U. S. Reddy. Global state considered unnecessary: Introduction to object-based semantics. LISP *and Symbolic Computation*, 9(1):7–76, 1996. Published also as Chapter 19 of [OT97].

[Red98] U. S. Reddy. Objects and classes in Algol-like languages. Presented at the Fifth International Workshop on Foundations of Object-Oriented Languages, January 17–18, 1998, San Diego, CA. To appear in Information and Computation. Avaliable at `http://www.cs.bham.ac.uk/~udr/`., 1998.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, January 1978. ACM, New York.

[Rey81a] J. C. Reynolds. The essence of ALGOL. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages,* Proceedings of the International Symposium on Algorithmic Languages, pages 345–372, Amsterdam, October 1981. North-Holland, Amsterdam. Reprinted as Chapter 3 of [OT97].

[Rey81b] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall International, London, 1981.

[Rey81c] J. C. Reynolds. IDEALIZED ALGOL and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161, Nice, France, December 1981. Cambridge University Press, Cambridge, 1982.

[Rey98] J. C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[Sch97] D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32(1):115–116, January 1997.

[Sie85] K. Sieber. A partial correctness logic for procedures (in an ALGOL-like language). In R. Parikh, editor, *Logics of Programs 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 320–342, Brooklyn, N.Y., 1985. Springer-Verlag, Berlin.

[Sie94] K. Sieber. Full abstraction for the second order subset of an ALGOL-like language. In *Mathematical Foundations of Computer Science*, volume 841 of *Lecture Notes in Computer Science*, pages 608–617, Kŏsice, Slovakia, August 1994. Springer-Verlag, Berlin. A version also published as Chapter 15 of [OT97].

[SS71] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*, pages 19–46. Polytechnic Institute of Brooklyn Press, New York, 1971. Also Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, Oxford.

[Sto74] L. J. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report MIT/LCS/TR-133, Massachusetts Institute of Technology, Laboratory for Computer Science, July 1974.

[Str64] C. Strachey. Towards a formal semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming, Proceedings of the IFIP Working Conference*, pages 198–220, Baden bei Wien, Austria, September 1964. North-Holland, Amsterdam (1966).

[Ten87] R. D. Tennent. A note on undefined expression values in programming logics. *Inf. Proc. Letters*, 24:331–333, 1987.

[Ten90] R. D. Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, 1990.

[Ten91] R. D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, 1991.

[Ten02] R. D. Tennent. *Specifying Software.* Cambridge University Press, 2002.

[TG00] R. D. Tennent and D. R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000.

[THM83] B. A. Trakhtenbrot, J. Y. Halpern, and A. R. Meyer. From denotational to operational and axiomatic semantics for ALGOL-like languages: an overview. In E. M. Clarke, Jr. and D. Kozen, editors, *Logics of Programs 1983*, volume 164 of *Lecture Notes in Computer Science*, pages 474–500, Pittsburgh, PA, 1983. Springer-Verlag, Berlin, 1984.

[TT91] R. D. Tennent and J. K. Tobin. Continuations in possible-world semantics. *Theoretical Computer Science*, 85(2):283–303, 1991.

[Vää00] J. Väänänen. Generalized quantifiers, an introduction. *Lecture Notes in Computer Science*, 1754:1–17, 2000.

[vD83] D. van Dalen. *Logic and Structure.* Springer, Berlin, second edition, 1983.

[vNM44] J. van Neumann and O. Morgenstern. *The Theory of Games and Economic Behaviour.* John Weily and Sons, 1944.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359, 4th International Symposium, Imperial College, London, September 1989. ACM, New York.

[YS97] D. M. Yellin and R. E. Strom. Protocol specifications and component adapters. *ACM Transactions on Programming Languages*, 19(2):292–333, March 1997.

[Zer13] E. Zermelo. Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Proceedings of the Fifth International Congress of Mathematicians*, pages 501–504, 1913.

Appendix A

Notations

$\Gamma \vdash P : \theta$	typing judgement
$\Omega \vdash s, P \ \Downarrow_\theta \ s', P'$	reduction relation (big-step)
$\mathit{States}(\Omega)$	the set of states for a world
$(f \mid x \mapsto y)$	a function equal to f except that it maps x to y
$(f \mid g)$	as above, generalized to all elements in $\mathrm{dom}(g)$
$\Omega \vdash s, P \Uparrow_\theta$	non-termination in operational semantics
$\Gamma \vdash P \cong_\theta P'$	extensional equivalence
$\Gamma \vdash P \cong_\theta P'$	observational equivalence (congruence)
$P[x_1, \ldots, x_n / P_1, \ldots, P_n]$	simultaneous substitution
$(X_\perp, \leq)$	lift of set X
$\perp_X$	the minimal element of a lift
$\mathit{Rel}(\Omega)$	the set of relations on a world
$\mathcal{R}$	relation
$\mathcal{I}_\Omega$	identity relation for a world
$\mathcal{R}_\theta$	parametric logical relation for terms of type θ
$\mathcal{R} \otimes \mathcal{R}'$	smash product of two relations

$\int_1 \otimes s_2$	smash product of two states
$\Gamma \vdash C[-] : \theta$	program context of type θ
$\mathit{Free}(P)$	set of free variables of P
$P \# P_1$	non-interference relation
$\{A\}\ C\ \{A'\}$	Hoare triple
τ	data types
σ	ground phrase types
θ	phrase types
Γ	type assignment
Ω	world
P	term of IA
M	ground type term of IA
F	first-order term of IA
V	variable-typed term of IA
E	integer expression term of IA
B	boolean expression term of IA
C	command term of IA
A	assertion phrase
S	specification phrase
x	identifier of IA
n	integer constant
v	variable identifier
e	integer expression identifier
b	boolean expression identifier
c	command identifier

f	function identifier
m	ground type identifier
s	state
Ar	arena
$\Game_A$	game
Σ	strategy
id	the copy-cat strategy
m	move
n	move occurrence
M	set of moves
I	set of initial moves
s	sequence
$\mathsf{s}' \sqsubseteq \mathsf{s}$	prefix of a sequence
$\star$	enabler of initial move
q	question
a	answer
L	legal position
$A \multimap B$	linear implication
$A \& B$	product (of games)
$\mathsf{I}_\emptyset$	the unit game
$\Sigma; \Sigma$	composition of strategies
$\langle \Sigma, \Sigma' \rangle$	pairing of strategies
$\mathsf{p}_i : A_0 \& A_1 \multimap A_i$	projection (copy-cat) strategy
$!A$	linear exponential
$\Sigma^\dagger : !A \multimap !B$	the promotion strategy

$\mathsf{der_A} : !A \multimap A$	the dereliction strategy
$A \times B$	categorical product
$A \Rightarrow B$	categorical exponentiation
$\mathbb{I}$	categorical initial object
$\mathsf{Y_A} : A \Rightarrow A \rightarrow A$	categorical recursion family of morphisms
Σ^{comp}	the set of complete plays of a strategy
α	arbitrary symbol
$-^{\langle\alpha\rangle}$	tagging (lexical or of languages)
$-\!\uparrow$	increment (lexical or of languages)
$-\!\downarrow$	decrement (lexical or of languages)
ω	string
$\omega \sqsubseteq \omega'$	prefix of a string
$\mathcal{A}$	alphabet
$\mathcal{R}$	set of regular expressions
$\emptyset$	empty language
ϵ	empty string
R	regular expression
$R \cdot R'$	concatenation
R^*	Kleene closure
$R + R'$	union
$R \cap R'$	intersection
$R \restriction \mathcal{A}$	restriction to an alphabet
$\lfloor R \rfloor$	effective alphabet
$R[R'/\omega]$	substitution
$R[\kappa]$	substitution, alternative notation

R^{det}	the set of deterministic sequences of R
$\widetilde{R}$	broadening
$\mathcal{K}_{\theta}^{\alpha}$	copy-cat regular expression
γ_{θ}^{x}	stability regular expression
γ_{ϵ}	empty dynamic constraint regular expression
u_{Γ}	the default environment, mapping all identifiers to copy-cat strategies
v_{Γ}	the default frame, not constraining any identifiers
$u_{\emptyset}$	the empty environment
$\mathfrak{M} = \langle u, v \rangle$	model for specifications
v	frame
$\mathcal{P}_{M,u,v}$	the set of passive traces for a phrase M
$\Psi \bullet -$	collection of quantifiers
$\mathfrak{M} \diamond \Psi$	model consistent with Ψ
$\#_{x}$	non-interference of two phrases at variable x
$\#_{\mathfrak{M},x}$	semantic non-interference of two phrases at variable x
$\Psi \bullet - \# -$	non-interference of two phrases
$\psi x : \theta / X$	generalized quantifier binding x, dependent on X
$-\mathcal{D}_{\Psi}-$	semantic dependence relationship between identifiers
$\mathcal{F}\langle\psi\rangle(u, v)$	regular language for an effect quantifier
$\lceil \mathcal{F}\langle\psi\rangle(u, v) \rceil$	completed specification
$R \triangleright \mathsf{a}$	temporal-style specification.

Vita

Education

- 1999–present, *Doctor of Philosophy.* Queen's University, School of Computing.
- 1995–1997, *Master of Science.* Queen's University, Department of Computing and Information Science.
- 1993–1995, *Bachelor of Science (Honors).* Memorial University of Newfoundland, Department of Computer Science.

Major academic and research awards

- 2002, NSERC Postdoctoral Fellowship
- 2001, Ontario Graduate Scholarship in Science and Technology
- 2000, ICALP Best Paper Award (for [GM00])
- 1999–2001, NSERC Post-graduate Scholarship (B)
- 1995–1997, NSERC Post-graduate Scholarship (A)