MIT/LCS/TR-564

THE CATEGORY OF FUNCTORS FROM STATE SHAPES TO BOTTOMLESS CPOS IS ADEQUATE FOR BLOCK STRUCTURE

Arthur Franklin Lent

February 1993

This blank page was inserted to preserve pagination.

The Category of Functors from State Shapes to Bottomless CPOs is Adequate for Block Structure

Arthur Franklin Leat
Massachusetts Institute of Technology
Laboratory for Computer Science

January 5, 1993

Abstract

We present a programming language EoA, which embodies what Reynolds has described as the "essence of ALGOL." In particular, EoA allows higher-order procedures and the declaration of block structured local variables. We develop a Plotkin-style Structured Operational Semantics for EoA. We present Tennent had developed a denotational model which aims to capture the semantics of stack allocated local variables. Tennent's denotational model for EoA, which, although not fully-abstract, appears to be the best model of block structure developed to this date. Tennent's model is based a category of functors from a collection of store shapes (possible worlds) to the category of bottomless cpos. The model is based on ideas pioneered in the early 1980's by Reynolds and Oles. The main result of this thesis is to prove that Tennent's model is **adequate** for EoA. We then abstract away some of the details of Tennent's model and state sufficient conditions for a functor category to be adequate for EoA. This thesis concludes with a comparison of the best known models of block structure, and discusses where Tennent's model fails to reach full abstraction.

Keywords: ALGOL, operational semantics, denotational semantics, adequate, fully abstract

Acknowledgments

I would first like to thank my advisor, Albert R. Meyer for suggesting the project and for his time spent discussing the research and suggestions for improving this thesis. I would also like to thank P. O'Hearn, K. Sieber, R. Tennent, and D. Wald for their comments on drafts of this thesis and many helpful technical discussions. I would also like to thank D. Jones and D. Wald for helpful comments regarding the formatting of this thesis. I would especially like to thank my wife, Nicole Sherry Skinner, for providing moral support when I needed it. Finally, I gratefully acknowledge the financial support of an ONR graduate fellowship.

Remarks

This is a slightly revised version of [8]. In particular, Section 4.3 had been added, some notation has been changed, and numerous typos have been corrected.

Contents

Chapter 1

Introduction

The goal of this thesis is to present a language which embodies many of the essential features of an a ALGOL-like language which we call EoA. The language EoA is a minor variant of Reynolds's "essence of ALGOL[20]". In Chapter 2 we define the syntax of EoA and develop a Plotkin-style Structured Operational Semantics for EoA [19]. In Chapter 3 we present Tennent's denotational semantics for EoA [12–14, 24, 25], which, although not fully-abstract, appears to be the best semantics of EoA developed to this date. Tennent's model is based on ideas pioneered in the early 1980's by Reynolds and Oles [15–17, 21]. The model uses the category of functors from a collection of store shapes (possible worlds) to the category of bottomless cpo's. Chapter 4 contains the main technical result of this thesis—a proof that Tennent's model is **adequate** for EoA. We then abstract away some of the details of Tennent's model and state some sufficient conditions for a functor category model to be adequate for EoA. Chapter 5 includes a discussion of how the Tennent model handles the Meyer-Sieber Examples [9], and discusses two counterexamples to full abstraction.

The remainder of this Chapter provides a general description of the criteria for "ALGOL-like languages" and presents a quick overview of Reynolds's "essence of ALGOL."

1.1 ALGOL-like Languages

We follow the work of Reynolds [20] and Halpern, Meyer and Trakhtenbrot [4, 26] in taking the following four principles as characterizing the class of ALGOL-like languages:

1. There is a consistent distinction between *commands* (or *programs*) which alter the store but do not return values, and *expressions* which return values but have no side-effects on the store.

2. The only explicit calling mechanism is *by-name*. (For parameters of basic data types, other mechanisms such as *by-value* or *by-reference* are available by simulation (syntactic sugaring).)

3. The language is fully typed. Higher-order procedures of all *finite* types (in ALGOL jargon, *modes*) are allowed. There is a clear distinction between *locations* and *storable values*.

4. The stack discipline is an explicit aspect of the semantics. Note that this discipline should be understood as a language design principle encouraging modularity in program construction rather than as an implementation technique for efficient storage management. It is better called the *local storage discipline* to avoid misunderstanding, and we do so henceforth.

—Trakhtenbrot, Halpern, and Meyer[26]

1.2 Reynolds's "Essence of ALGOL"

A key observation embodied in Reynolds's essence of ALGOL is that ALGOL is essentially a simply-typed functional language. With a careful understanding of types and call-by-name as the only built in parameter-passing mechanism, we can see that β-equivalence is a sound reasoning principle for ALGOL! Consequently, the kernel syntax of Reynolds's essence of ALGOL is a simply typed λ-calculus over the appropriate base types and constants required to capture the imperative features of ALGOL. Although the only parameter passing mechanism explicitly present in the essence of ALGOL is call-by-name, the mechanisms of call-by-value and call-by-reference for storable data types can easily be simulated.

1.2.1 Imperative Types

To understand Reynolds's essence of ALGOL, we must understand his view of imperative types. Since the local storage discipline is essentially incompatible with storing state-dependent objects, there is a clear distinction between data and phrase types. Essentially, data types are those primitive entities which we can "store" (*i.e.* those which are kept track of in the state), whereas phrase types are the types which code can take on. Some examples of plausible data types are real, integer, and boolean. Some examples of plausible phrase types are real thunks, integer thunks, and commands. For an ALGOL-like language, the data types are "state independent," whereas *all* of the phrase types implicitly involve the state in some way.

For each data type, there corresponds a phrase type for talking about such values. Reynolds takes as primitive data types: real, integer, and boolean, so the base phrase types corresponding to these basic data types are: exp[real],

exp[integer], and exp[boolean]. For example, an object of type exp[real] is a "real exp(ressor)"—something which manages to read a real from the state. From the point of view of this thesis and in spite of our suggestive notation, exp[·] is not to be considered a general type constructor. Code can not take on the type integer. Code takes on the type exp[integer]. From the point of view of programming in "essence of ALGOL," the phrase type integer *does not exist.* All "expressions" in an imperative language have an implicit ability to depend on the state. It would be possible to augment our collection of phrase types by a type which corresponded to "pure" (state-independent) integers. Most code, however, is actually written to expect the more general (state-dependent) expressions. Moreover, since expressions are side-effect free, there seems to be little motivation to make a distinction between the integer "5" and the thunk "$\underline{5}$" which in any state evaluates to the integer 5.

Another base phrase type is comm. We have already noted that a fundamental criterion for being an "ALGOL-like language," is that expressions return values, but do not have side-effects on the state. All side-effects are restricted to occur at the type comm. Terms of type comm correspond to **statements** in the more traditional jargon.

The most fundamental notion of an imperative language is the **variable.**[1] Many authors model variables by *locations.* Some authors make the location-like behavior of variables an explicit part of the syntax of the languages (sometimes introducing a type "location" in place of the type "variable"). We follow the view that it is best not to commit to such representation issues at the level of the language design (Note: we will be modeling variables by locations). A variable has within it the ability to either write an element of a data type in the state, or read an element of a data type from the state. Consequently, we have the phrase types: var[real], var[integer], and var[boolean]. In fact we can view a variable as consisting of two separate components: a reading component, and a writing component. The reading component of a variable of type var[τ] (which corresponds to what is often called the r-value), would have type exp[τ]. The writing component of a variable (which corresponds to what is often called the l-value), could be viewed as type exp[τ] $\rightarrow$ comm. The writing component is a state-to-state mapping which is parameterized by a state-dependent element of the data type τ. Reynolds calls the writing component of a variable an acceptor, and gives acceptors their own phrase type. So, essentially a var[τ] is a exp[τ] paired with an acc[τ].

Reynolds also addresses the issue of subtyping of data types in his definition of the essence of ALGOL. Since we feel that the issues of subtyping data types

[1] We break with somewhat traditional terminology in programming language semantics, and use "variable" in the traditional programming sense—something which allows the state to be accessed or changed. We use "identifier" in the sense in which logicians have traditionally used variable, *i.e.* to denote a place holder in a term. So in this thesis, a variable is something we read from or write to, whereas an identifier appears free in a term, or is bound by a λ or some other binding construct.

and implicit coercions are largely orthogonal to the issue of local variables, from here on we will describe Reynolds's essence of ALGOL as though it does not have the data type real. We also feel that having the data type boolean does not raise any substantial issues in the context of local variables; we will also drop it from our discussion. The fact that we only have a single storable data type will allow us to simplify syntax in several ways. For example, we can now use exp for exp[integer], acc for acc[integer], and var for var[integer]. It will also allow us to simplify the syntax of local-variable-declaration blocks, as all of our local variables will be for the data type integer.

1.2.2 Kernel Syntax

Now, given this intuitive understanding of types, we can understand the kernel syntax of Reynolds's essence of ALGOL as a simply typed λ-calculus with base types exp, acc, var, and comm, with appropriate typed constants and a few extra term constructors. When talking informally, we will write our binary operators in infix notation, *e.g.* writing $c_0\,;c_1$ instead of $((\mathsf{seq}\ c_0)\ c_1)$. For our proofs however, we restrict ourself to the "correct" syntax.

The constants of type exp are simply the numerals, namely $0, 1, 2, \ldots$ At type comm, there is only one constant—the do nothing command skip. There are no constants of type acc or var. There are also a variety of constants of higher type. For example, conditionals are uniformly available at all types. We will write the conditional constant for type θ as IFexp_θ; it has type $\mathsf{exp} \to \theta \to \theta \to \theta$. The intended semantics of IFexp_θ will be that if the exp evaluates to 0 then the first consequent of the conditional will be used, if the exp evaluates to a non-zero integer then the second consequent of the conditional will be used.

There are also assignment operators, $:=$, for the types acc and var. The intended semantics of the command $a := e$ is the command which assigns the value of expressor e in the current store to the variable whose updating component is a. The intended semantics of $v := e$ is the command which assigns the value of expressor e in the current store to the variable v.

We also have curried versions of the standard arithmetic operations, which operate on types just built up from exp.[2] For example we have $\mathsf{succ} : \mathsf{exp} \to \mathsf{exp}$, and $\mathsf{plus} : \mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp})$.

The standard λ-calculus term constructors of application and λ-abstraction $(\lambda\ \iota\ :\ \theta\ .\ M)$ are a part of the essence of ALGOL. There is also a term constructor rec which behaves like a least fixpoint operator. Finally, there is also an operator newvar of type $\mathsf{var} \to \mathsf{comm} \to \mathsf{comm}$ which is used for local variable declarations.

[2] Due to the availability of implicit conversions they can also operate on var arguments.

1.2.3 Syntactic Sugar

The full version of Reynolds's essence of ALGOL is the kernel language augmented by six sugaring constructs: higher-order conditionals, multiple abstraction, multiple application, let, letrec, and New v in B. For completeness, we provide the sugared and desugared constructs, even though this is now fairly well established in the literature. Our treatment is largely taken from that of Tennent [24].

Our first bit of sugar is higher-order conditionals. Since our the intended parameter passing mechanism is call-by-name, we can define $\mathsf{IFexp}_{\theta\to\theta'}$ as follows:

$$\mathsf{IFexp}_{\theta\to\theta'} =_{\mathrm{df}} \lambda e:\mathsf{exp}.\lambda p_1:\theta\to\theta'.\lambda p_2:\theta\to\theta'.\lambda q:\theta.(\mathsf{IFexp}_{\theta'}\ e\ (p_1 q)\ (p_2 q))$$

When there is ambiguity about parenthesis, we assume application associates to the left.

The next sugaring construct allows multi-parameter procedures (and thus also applications of multi-parameter procedures) via currying:

$$\frac{\pi[\iota_1:\theta_1,\ldots\iota_n:\theta_n]\vdash P:\theta}{\lambda(\iota_1:\theta_1,\ldots,\iota_n:\theta_n).P:\theta_1\times\cdots\times\theta_n\to\theta}$$

$$\frac{\pi\vdash P:(\theta_1\times\cdots\times\theta_n)\to\theta \quad \pi\vdash Q_1:\theta_1\cdots\pi\vdash Q_n:\theta_n}{\pi\vdash P(Q_1,\ldots,Q_n):\theta}$$

So our desugaring is:

$$\lambda(\iota_1:\theta_1,\ldots,\iota_n:\theta_n).P =_{\mathrm{df}} \lambda\iota_1:\theta_1.\ldots.\lambda\iota_n:\theta_n.P$$

$$P(Q_1,\ldots,Q_n) =_{\mathrm{df}} P(Q_1)\cdots(Q_n)$$

Reynolds's essence of ALGOL uses a convenient notation for making (non-recursive) local definitions in programs which was suggested by Landin [7]:

$$\frac{\pi[\iota_1:\theta_1,\ldots,\iota_n:\theta_n]\vdash Q:\theta \quad \pi\vdash P_i:\theta_i(\text{for } i=1,2,\ldots,n)}{\pi\vdash \mathsf{let}\ \iota_1\ \mathsf{be}\ P_1\ \&\cdots\&\ \iota_n\ \mathsf{be}\ P_n\ \mathsf{in}\ Q:\theta}$$

The let *does not* bind $\iota_1,\ldots,\iota_n$ in any of $P_1,\ldots,P_n$. A let creates a non-recursive set of declarations, in which Q gets evaluated (the letrec construct, which will be described shortly is used for setting up a possibly mutually recursive set of declarations in which Q can be evaluated). Also, since our premises are of the form $\pi\vdash P_i:\theta_i$, rather than $\pi[\iota_1:\theta_1,\ldots,\iota_n:\theta_n]$, we did not need to include the types of $\iota_1,\ldots,\iota_n$ explicitly in the let construct; these types can be inferred by looking at the P_i's.

Our desugaring is:

$$\mathsf{let}\ \iota_1\ \mathsf{be}\ P_1\ \&\cdots\&\ \iota_n\ \mathsf{be}\ P_n\ \mathsf{in}\ Q =_{\mathrm{df}} (\lambda\iota_1:\theta_1.,\cdots\lambda\iota_n:\theta_n.Q)(P_1)\cdots(P_n)$$

Reynolds's essence of ALGOL also uses a convenient notation for making possibly recursive local definitions in programs (also from Landin):

$$\frac{\pi[\iota_1:\theta_1,\ldots,\iota_n:\theta_n] \vdash Q:\theta \quad \pi[\iota_1:\theta_1,\ldots,\iota_n:\theta_n] \vdash P_i:\theta_i (\text{for } i=1,2,\ldots,n)}{\pi \vdash \mathsf{letrec}\ \iota_1:\theta_1\ \mathsf{be}\ P_1\ \&\ \cdots\ \&\ \iota_n:\theta_n\ \mathsf{be}\ P_n\ \mathsf{in}\ Q:\theta}$$

The letrec construct *does* bind $\iota_1,\ldots,\iota_n$ in all of $P_1,\ldots,P_n$ (and also, of course, Q). Also, it is no longer possible to infer θ_i merely from π and P_i; consequently the θ_i's must appear explicitly in the letrec construct.

Desugaring a letrec is a little harder. We first show how to desugar the case when $n = 1$. Then we reduce the case of $n > 1$ to the case of $n = 1$. For the non-multiple case ($n = 1$), we have:

$$\mathsf{letrec}\ \iota_1:\theta_1\ \mathsf{be}\ P_1\ \mathsf{in}\ Q =_{\mathrm{df}} (\lambda\iota_1:\theta_1.Q)(\mathsf{rec}\ \lambda\iota_1:\theta_1.P_1)$$

We reduce the case of multiple cases to the single case by using Bekić's theorem [2, 27]. In our notation, Bekić's theorem reads as follows:

Theorem 1 (Bekić)

$$\begin{array}{l}\mathsf{letrec}\ \iota_1:\theta_1\ \mathsf{be}\ P_1\ \&\ \iota_2:\theta_2\ \mathsf{be}\ P_1\ \mathsf{in}\ Q \\ \quad = \quad \mathsf{letrec}\ \iota_1:\theta_1\ \mathsf{be}\ (\mathsf{letrec}\ \iota_2:\theta_2\ \mathsf{be}\ P_2\ \mathsf{in}\ P_1) \\ \qquad\quad \mathsf{in}\ (\mathsf{letrec}\ \iota_2:\theta_2\ \mathsf{be}\ P_2\ \mathsf{in}\ Q)\end{array}$$

Bekić's theorem obviously generalizes to handle $n \geq 3$ as follows:

$$\begin{array}{l}\mathsf{letrec}\ \iota_1:\theta_1\ \mathsf{be}\ P_1\ \&\ \iota_2:\theta_2\ \mathsf{be}\ P_2\ \&\ \cdots\ \&\ \iota_n:\theta_n\ \mathsf{be}\ P_n\ \mathsf{in}\ Q \\ \quad = \quad \mathsf{letrec}\ \iota_1:\theta_1 \mathsf{be}\ (\mathsf{letrec}\ \iota_2:\theta_2\ \mathsf{be}\ P_2\ \&\ \cdots\ \&\ \iota_n:\theta_n\ \mathsf{be}\ P_n\ \mathsf{in}\ P_1) \\ \qquad\quad \mathsf{in}\ (\mathsf{letrec}\ \iota_2:\theta_2\ \mathsf{be}\ P_2\ \&\ \cdots\ \&\ \iota_n:\theta_n\ \mathsf{be}\ P_n\ \mathsf{in}\ Q)\end{array}$$

The final piece of sugar in Reynolds's essence of ALGOL which is worth mentioning here is variable declaration blocks. The syntax of such a block is:

$$\frac{\pi[\iota:\mathsf{var}[\tau]] \vdash P:\mathsf{comm}}{\pi \vdash \mathsf{New}[\tau]\ \iota\ \mathsf{in}\ P:\mathsf{comm}}$$

Formally, we could let the binding be done by a λ-expression, and then introduce a typed constant newvar_τ into the language in order to capture the rest of the intention of the declaration. This then gives us the following desugaring:

$$\mathsf{New}[\tau]\iota\,\mathsf{in}\,P =_{\mathrm{df}} \mathsf{newvar}_\tau(\lambda\iota:\mathsf{var}[\tau].P).$$

Note: since we have dropped all data types other than integer, we do not even need to explicitly mention τ in the declaration, thus giving $\mathsf{New}\,\iota\,\mathsf{in}\,P$, and the desugaring $\mathsf{newvar}\ (\lambda\iota:\mathsf{var}.P)$ instead.

Chapter 2

Definition of EoA

The language EoA is built upon a simply typed λ-calculus over base types exp and comm. In addition to the standard λ-calculus term constructors of λ-abstraction and application, EoA contains a term constructor New for expressing local-variable declaration blocks. EoA contains typed constants for basic arithmetic operations on exp. Recursion is uniformly available at all types. Conditionals are provided for the base types, and are uniformly definable for higher types. The "do nothing" command skip is available, as is sequencing of commands. For simplicity there is only one basic data type, which we assume to be the non-negative integers; it should be a routine exercise to modify the results here to extend EoA with additional basic data types (such as booleans or characters).

2.1 Differences Between EoA and Reynolds's Essence of ALGOL

There are three fundamental differences EoA between and Reynolds's essence of ALGOL. First, the only storable data type of EoA is integer, whereas Reynolds also considers real and boolean. We do not consider the data type real, because our primary interest is in understanding the nature of block structured local variables, and we believe that the issue of subtyping and implicit coercions is largely orthogonal to the issue of block structured local variables. For simplicity, we also do not have boolean as a basic data type. We expect that it would be a routine exercise to extend the work here to handle additional basic data types.[1]

The second fundamental difference is the lack of the type var(iable) in EoA, although we do retain acc(eptor) and exp(ressor). The technical justifications

[1] Although we expect it to be simple to extend the language with other basic data types, we have not thought carefully about extending our results for a version of EoA extended with a rich structure of subtypes and implicit coercions.

for eliminating var appear in Section 2.2.1. EoA treats acc as a type synonym for exp $\rightarrow$ comm. A fundamental consequence of the lack of var and the change to acc is that we no longer need coercions (implicit or explicit) from var to acc or exp. Reynolds's essence of ALGOL provides an implicit coercion from the type var to exp. When we think of acc as merely a type synonym for exp $\rightarrow$ comm, we see that the assignment operator := merely functions as an explicit coercion from var to (exp $\rightarrow$ comm). Of course, there will be some minor syntactic modifications to new-variable blocks. For example, in the sugared language the New structure will now need to take two identifiers in addition to the block-body as argument (where one identifier will be tied to the writing component of the newly allocated variable, and the other tied to its reading component).

The final substantial difference between EoA and Reynolds's essence of ALGOL is that in EoA, identifiers come explicitly with types—as opposed to using a simply-typed lambda calculus based on untyped identifiers. Specifically, Reynold's essence of ALGOL uses typing judgments of the form $\pi \vdash M : \theta$, and lambda abstraction takes the form $\lambda x : \theta . M$. In EoA, typing judgments are unnecessary, and lambda abstraction looks like $\lambda x . M$, where x, by definition, has some specified type. We make this simplification for expository reasons. A suitable version of the adequacy result also holds for the version with untyped identifiers, but some substantial changes do need to be made to the definitions for it to work.

There are two other minor differences between EoA and Reynolds's essence of ALGOL, both existing to simplify our exposition. For reasons which arise when defining the operational semantics of EoA, we find it more convenient *not* to use Reynolds's desugaring of new variable declarations. Thus we leave New as a term constructor in EoA. Moreover, since we keep New structures in our kernel language, we do not bother to introduce the constant newvar. There is one additional change to the New structure. We find it useful to include a fourth component in a New structure—an argument of type exp which will provide the initial value of the newly allocated location. So, if E is a term of type exp, P a term of type comm, ι an identifier of type acc, and κ an identifier of type exp, then (New ι, $\kappa \leftarrow E$ in P) is a term of type comm.

Finally, we introduce a collection of recursion constants Y_θ, one for each type θ. The constant Y_θ, has type $(\theta \rightarrow \theta) \rightarrow \theta$. These constants enable us to discard the term constructor rec. Specifically if M is a term of type $\theta \rightarrow \theta$, then in EoA we use $(\mathsf{Y}_\theta\ M)$, instead of (rec M).

The rest of this Chapter consists of a formal definition of the syntax and operational semantics of EoA and a precise statement of our definition of adequacy.

2.2 Kernel Syntax

2.2.1 Types

The base types of EoA are exp (for expressor) and comm (for command). The type exp contains integer **thunks**—objects which when given a state evaluate to an integer. The type comm contains imperative commands—objects which when given a state evaluate to a new state (*e.g.* storing 5 in location x). The full set of types is defined inductively from the base types such that: if θ and τ are types then $\theta \rightarrow \tau$ is a type (corresponding to the type of functions from θ to τ).

EoA does not have integer as a base type. Consequently, we can not have integers as constants in our language. But we *can* have constants of type exp which behave like integer constants. In particular, if n is an integer, we could write $\underline{n}$ to denote the constant thunk which in any state produces the integer n. There is a subtle, but fundamental, difference between constants in the language and constant thunks. In EoA we generally think of an abstract meaning associated with a piece of code. Usually this meaning is a function on states. Such state-dependent objects are termed thunks. A constant thunk is a thunk that always gives the same result regardless of the state. The notion of a "constant thunk" is unrelated to the notion of being a linguistic constant. For example, (plus 1 1) is a constant thunk, but is clearly not a syntactic constant in the language EoA. Similarly it is plausible to introduce into EoA linguistic constants which do depend on the state. We have simply chosen not to do so.

The most commonly used function type is exp $\rightarrow$ comm. Frequently we will use acc to abbreviate exp $\rightarrow$ comm. Some authors would also like to have available a type which would correspond to that of integer $\rightarrow$ comm. When this type is available, it, rather than exp $\rightarrow$ comm, is typically used for acc. The type var is also occasionally introduced as a base type to correspond to the type acc $\times$ exp. Now that we have a better understanding of the nature of local variables, these types are not as necessary. For example, we can simulate the term M : integer $\rightarrow$ comm by the term $(\delta\ M)$: exp $\rightarrow$ comm, where δ is:

$$\delta \equiv \lambda\iota.\ \lambda\kappa.\ (\mathsf{New}\ \iota',\ \kappa' \leftarrow \kappa\ \mathsf{in}\ (M\ \kappa'))$$

To see how this works, consider

$$C \equiv \lambda\kappa.(\mathsf{seq}\ (l_1^{\mathsf{acc}}\ (\mathsf{succ}\ \kappa))\ (l_2^{\mathsf{acc}}\ (\mathsf{succ}\ \kappa)))$$

in $(C\ l_1^{\mathsf{exp}})$ and $((\delta\ C)\ l_1^{\mathsf{exp}})$, assuming that l_i^{exp} and l_i^{acc} are identifiers properly bound to the reading and writing components of location i. Consider executing these two commands in a state where location 1 has the value 10 and location 2 has the value 100. The term $(C\ l_1^{\mathsf{exp}})$ results in 11 in location 1 and 12 in location 2. The term $((\delta\ C)\ l_1^{\mathsf{exp}})$ results in 11 in both locations 1 and 2.

The type var is also unnecessary. We can eliminate var by referring to a location via two distinct identifiers, one whose job is to represent the r-value part (the reading part) and the other whose job is to represent the l-value part (the updating part). Since we do not use var, we no longer need the explicit coercions cont : var → exp, and := which has type var → acc. Depending on one's choice of semantics, the traditional assignment operator, :=, might be used as more than just an explicit coercion—it may also be used to pass an argument to an acceptor "by value." The choice is between evaluating the argument in the current state and then passing the resulting constant thunk to the acceptor (by-value), or passing the unevaluated, potentially non-constant thunk to the acceptor (the way EoA does it). The by-value version can be expressed in EoA using the term δ defined above.

For example, consider a New block in a language with the type var, and where New blocks have an argument slot for the initial value of the new location. Specifically, we consider a block of code to allocate two variables (the first initialized to 0 and the second initialized to 1) with a body which assigns the value stored in the location pointed to by the second variable to value stored in the location pointed to by the first variable:

$$\mathsf{New}\ \mathit{fst} \leftarrow 0\ \mathsf{in}\ (\mathsf{New}\ \mathit{snd} \leftarrow 1\ \mathsf{in}\ (:=\mathit{fst}\ \mathit{snd})).$$

The translation of the block into EoA where ":=" is only doing an explicit conversion looks like

$$\mathsf{New}\ \mathit{fstL},\ \mathit{fstR} \leftarrow 0\ \mathsf{in}\ (\mathsf{New}\ \mathit{sndL},\ \mathit{sndR} \leftarrow 1\ \mathsf{in}\ (\mathit{fstL}\ \mathit{sndR}))$$

When ":=" is both doing a coercion and forcing a call-by-value application the translation looks like

$$\mathsf{New}\ \mathit{fstL},\ \mathit{fstR} \leftarrow 0\ \mathsf{in}\ (\mathsf{New}\ \mathit{sndL},\ \mathit{sndR} \leftarrow 1\ \mathsf{in}\ (\delta\ \mathit{fstL}\ \mathit{sndR})).$$

2.2.2 Identifiers and Terms

We assume that we have an infinite set of identifiers of each type. We let α^θ be a metavariable ranging over identifiers of type θ (we'll just write α when we need a metavariable ranging over identifiers, but the type is unimportant). We let ι be a metavariable ranging over identifiers of type acc $\equiv$ exp → comm, and κ range over identifiers of type exp. The set of constants from which we build up EoA is shown in Figure 2.1. The set of EoA terms, denoted by $\mathcal{L}$, is built up from these constants by the usual inductive definition—augmented with one extra case for new blocks:

- α^θ is term of type θ;
- $c^\theta \in \mathit{Const}$ is a term of type θ;

n	exp	skip	comm
$\mathsf{IFexp}_{\mathsf{exp}}$	exp → exp → exp → exp	seq	comm → comm → comm
$\mathsf{IFexp}_{\mathsf{comm}}$	exp → comm → comm → comm	Y_θ	$(\theta \to \theta) \to \theta$
succ	exp → exp	plus	exp → exp → exp

Figure 2.1: *Const*, the set of constants of EoA and their types.

- $(M\ N)$ is a term of type τ if M has type $\theta \to \tau$ and N has type θ;
- $(\lambda\alpha^\theta.\ M)$ is a term of type $\theta \to \tau$ if M has type τ;
- (New ι, $\kappa \leftarrow E$ in P) is a term of type comm if E has type exp and P has type comm.

We abbreviate the phrase "M is a term of type θ" by writing $M:\theta$.

We adopt the following conventions about associativity and the extent of binding operators which allow us to drop certain parenthesis without introducing ambiguity.

- All applications are parenthesized to the left so the expressions $(M_1\ M_2\ M_3)$ or $M_1\ M_2\ M_3$ can be written for $((M_1\ M_2)\ M_3)$.
- The body of a λ-expression or New declaration extends to the first unmatched right parenthesis.
- Although readability is usually enhanced by dropping unneeded parentheses, we feel free to leave in any parenthesis that will enhance readability.

For types, we assume that $\to$ associates to the right, so that we may also drop unnecessary parenthesis in type expressions. For example, we can write $\tau_1 \to \tau_2 \to \tau_3$ in place of $(\tau_1 \to (\tau_2 \to \tau_3))$. In addition, in informal discussion, we may write binary operators in infix notation (*e.g.* $c_1\ ;c_2$ instead of $(\mathsf{seq}\, c_1\, c_2)$, and $e_1 + e_2$ instead of $(\mathsf{plus}\ e_1\ e_2)$).

The intended semantics of a term constructed with New is not obvious. Consider the term

$$\mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P.$$

The intended semantics is to evaluate E in the current state to an integer n. Then, extend the state by a new location whose initial value is n. Finally, evaluate P in a manner in which the identifier ι is bound to the writing component of this new location, and the identifier κ is bound to the reading component of this new location. There are several important details to note about this construct. First, the order of the identifiers is important to remember—the writing component (the acceptor) comes first, and the reading component (the expressor) comes second. In addition, the choice of the names of these identifiers in no way

affects the evaluation of E; however, the choice of the names of these identifiers has a major impact upon the evaluation of P. In other words, the New binds ι and κ the term P, but not in the term E.

2.2.3 Free and Bound Identifiers, Substitution, and Closed Terms

The standard λ-calculus concepts of free and bound identifiers, closed and open terms, and substitution make sense for EoA. When we say "free identifier" we are really talking about the standard λ-calculus notion of "free variable." Unfortunately, in the setting of an ALGOL-like language, it is necessary to co-opt the meaning of "variable." Instead of using "variable" in the "logical" sense we use it in the "programming sense"—variables are how we introduce side-effects and dependence on a state. To remain consistent with the usual definitions, we will write "FV(M)" and "BV(M)" to denote the free identifiers of M and bound identifiers of M respectively.

Since we have an extra term constructor (New) we need to modify the usual definitions to fit EoA. This requires some care, since, like λ-abstraction, New is a binding construct. We adapt most of the standard conventions and definitions of Barendregt [1] to EoA.

The set FV(M) of **free identifiers** of M is defined inductively on the structure of M as follows:

$\mathrm{FV}(c) = \emptyset$, for all constants c.
$\mathrm{FV}(\alpha) = \{\alpha\}$, for all identifiers α.
$\mathrm{FV}(MN) = FV(M) \bigcup \mathrm{FV}(N)$.
$\mathrm{FV}(\lambda\alpha.M) = \mathrm{FV}(M) - \{\alpha\}$.
$\mathrm{FV}(\mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P) = \mathrm{FV}(E) \bigcup (\mathrm{FV}(P) - \{\iota, \kappa\})$.

Note that either ι or κ (or both) can be free in New ι, $\kappa \leftarrow E$ in P iff it appears free in E.

The set BV(M) of **bound identifiers** of M is defined inductively on the structure of M as follows:

$\mathrm{BV}(c) = \emptyset$, for all constants c.
$\mathrm{BV}(\alpha) = \emptyset$, for all identifiers α.
$\mathrm{BV}(MN) = BV(M) \bigcup \mathrm{BV}(N)$.
$\mathrm{BV}(\lambda\alpha.M) = \mathrm{BV}(M) \bigcup \{\alpha\}$.
$\mathrm{BV}(\mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P) = \mathrm{BV}(E) \bigcup \mathrm{BV}(P) \bigcup \{\iota, \kappa\}$.

It is perfectly reasonable for an identifier to appear both free and bound in the same term (*e.g.* α in $(x\ (\lambda\alpha.\ \alpha)\ \alpha)$).

Definition 2 *A term M is* **closed** *iff $FV(M) = \emptyset$.*

Barendregt defines a change of bound identifiers in M as the replacement of a part $(\lambda\alpha.\, N)$ of M by $(\lambda\alpha'.\, N[\alpha'/\alpha])$, where α' does not occur (at all) in N. Since α' is fresh we can syntactically replace all occurrences of α in N by α' without the usual dangers. For EoA, we must also allow for the change of identifiers bound by New. In particular, a change of bound identifiers in M can be of the above form, or of the following form— the replacement of a part $(\mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ C)$ of M by either:

- $(\mathsf{New}\ \iota,\ \kappa' \leftarrow E\ \mathsf{in}\ C[\kappa'/\kappa])$ where κ' does not occur (at all) in C, or
- $(\mathsf{New}\ \iota',\ \kappa \leftarrow E\ \mathsf{in}\ C[\iota'/\iota])$ where ι' does not occur (at all) in C.

We can now define α-congruence ($\equiv_\alpha$) for EoA terms by saying: $M \equiv_\alpha N$, if N results from M by a series of changes of bound identifiers. We adopt the convention that terms that are α-congruent are identified. We also adopt the further **identifier convention:** *If $M_1, \ldots, M_n$ occur in a certain mathematical context* (*e.g.* definition, proof), *then in these terms all bound identifiers are chosen to be different from the free identifiers.*

Since the binding constructs in EoA are somewhat different from the pure simply-typed λ-calculus, it is worth providing the full definition of the substitution operator in order to avoid confusion. Specifically we write $M[N/\alpha]$ to represent the result of substituting N for all free occurrences of α in M. We define $M[N/\alpha]$ by induction on the structure of M as follows:

$$\begin{aligned}
\alpha[N/\alpha] &= N \\
\alpha'[N/\alpha] &= \alpha' \text{ for } \alpha' \text{ any identifier other than } \alpha \\
c[N/\alpha] &= c \text{ for } c \text{ any constant} \\
(PQ)[N/\alpha] &= (P[N/\alpha])(Q[N/\alpha]) \\
(\lambda\alpha'.\, M)[N/\alpha] &= (\lambda\alpha'.\, M[N/\alpha])
\end{aligned}$$

In the above clause it is not necessary to say "provided that $\alpha' \not\equiv \alpha$ and $\alpha' \notin \mathrm{FV}(N)$," as the identifier convention insures that this is the case. Finally we have the case for New which is again simplified by adopting the identifier convention.

$$\begin{aligned}
&(\mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ C)[N/\alpha] \\
&\quad = \ (\mathsf{New}\ \iota,\ \kappa \leftarrow (E[N/\alpha])\ \mathsf{in}\ C[N/\alpha])
\end{aligned}$$

So long as $N : \theta$ then it is obvious from the definition of $M[N/\alpha^\theta]$ that the result is a term which has the same type as M.

2.3 Assigning Operational Semantics

An ability to read from and write to the state is a fundamental aspect of imperative programming. What distinguishes EoA from other languages which combine imperative programming and the higher-order functionality of the simply-typed lambda calculus is its variable allocation mechanism. This allocation occurs via New blocks. Intuitively we think of the process of executing a New block as: upon entrance, extending the state by another "location"; then evaluating the body of the block in an "environment" where the "block identifiers" are appropriately bound to this new location; and finally, at the end of the block removing this "new" location from the state. This informal description points out three elements which are crucial to defining our operational semantics:

- We need a way of representing the state.
- We need a notion of "location," a way of extending the state by a new location, and a way of removing a location from the state.
- We need a way to evaluate the body of a block where the block identifiers are properly tied to a new location.

Given that we will only be storing integers in the state, a very simple representation of states and locations arises naturally. Specifically, a state will simply be a finite sequence of integers. A location can be an index into such a sequence. To extend a state by a new location, we simply need to append the initial value for the new location onto the state. To remove the last location from the state, we simply strip the last value off the sequence.

In order to evaluate the body of a block we need a way to tie terms to states. There are two plausible approaches. On the one hand, we could introduce location constants into the language. This would enable us to determine the behavior of a New block by looking at the behavior of the body of the block when we instantiate the block identifiers by appropriate location constants (one constant for the acceptor and a different constant for the expressor). On the other hand, we could introduce the notion of binding—to variables. The purpose of a binding would be to tie the block identifiers properly to the new location. Thus, to determine the behavior of a New block under a binding B, we look at the behavior of the body of the block under a binding B' (which is exactly like B except B' ties the block identifiers properly to the new location). We choose the second approach.

We now make this precise with a few definitions. A **state** is a finite sequence of integers. The collection of all possible states, written $\mathcal{S}$, is $\mathbb{N}^*$. We will typically use the metavariable σ to range over states. The following five items are important for manipulating states:

- length: $\mathcal{S} \to \mathbb{N}$.

- proj(σ, k): the k-th component of σ.
- update(σ, k, n): returns a state just like σ except the k-th component is set to n.
- EMPTYSTATE: denotes the zero-length state.
- $s_1 \cdot s_2$ is used to denote the concatenation of two states, s_1 and s_2. More formally,

$$\langle v_1, \ldots, v_k \rangle \cdot \langle v'_1, \ldots v'_{k'} \rangle \equiv \langle v_1, \ldots v_k, v'_1, \ldots v'_{k'} \rangle.$$

Often we will write the integer v when we really mean the state $\langle v \rangle$; however, the intended interpretation should always be clear from the context.

A **binding**—to variables, henceforth called binding, is a finite function from identifiers of type exp or acc to $\mathbb{N}$. We call the collection of all bindings $\mathcal{B}$. The purpose of a binding is to tie a term M to a state σ via its free identifiers of type exp and acc. Specifically, a binding explains how to use an identifier of type exp to read from a location in the state and it also explains how to use an identifier of type acc to write to a location in the state.

Definition 3 *The* **index** *of a non-empty binding* B (written index(B)) *is the greatest integer* i *in the range of* B. *The* **index** *of the empty binding,* EMPTYBIND, *is* 0.

Definition 4 *An* **lterm** *is a pair* $[M, B] \in (\mathcal{L} \times \mathcal{B})$ (recall that $\mathcal{L}$ is the set of all (open and closed) EoA terms). *We extend the notion of free identifiers to lterms in the following way:* $\mathrm{FV}([M, B]) = \mathrm{FV}(M) - \mathrm{Dom}(B)$ (note: our identifier convention allows us to assume that bound identifiers of M are also not in $\mathrm{Dom}(B)$). *A* **closed lterm** *is an lterm* $[M, B]$ *such that* $\mathrm{FV}([M, B]) = \emptyset$. *An* **instantiation** $[L, B]$ *of a closed lterm* $[M, B]$ *is the result of a sequence substitutions of a term* $N_1 : \theta_1, \ldots, N_k : \theta_k$ *for an sequence of identifiers* $\alpha_1^{\theta_1}, \ldots \alpha_k^{\theta_k} \notin \mathrm{Dom}(B)$ *to obtain the term* L. *A* **closed instantiation** *of a closed lterm* $[M, B]$ *is an instantiation of* $[M, B]$ *which is a closed lterm.*

We will explain the behavior of EoA code by defining the behavior of closed lterms via a Structured Operational Semantics (SOS) in the style of [19]. Before we go into the definition we first note that EoA is somewhat unconventional. EoA combines two very different notions. On the one hand, it has the full type structure of the simply typed lambda calculus, the intended semantics for the functional fragment of EoA is the call-by-name lambda calculus, and the functional fragment can be handled by term rewriting. On the other hand, the evaluation of base type expressions involving imperative features requires a full fledged SOS involving configurations.

Consequently, our operational semantics is a hybrid of these two schemes. The configuration rewriting does not even apply for higher order terms—we

directly restrict ourselves to the base types comm and exp for configuration rewriting. All evaluation of higher order terms is handled via the term rewriting rules (as we will see in Figure 2.3). The method of evaluation of base type terms depends on the principal operator of the term. If it is of the functional sort (a lambda abstraction) then term rewriting will be used to determine how the configuration behaves. If it is of an imperative sort (*e.g.* an assignment, or the evaluation of an expression) then the rules for the imperative features kick in.

We define a **configuration** to be a triple $\langle Q, B, \sigma \rangle \in (\mathcal{L} \times \mathcal{B} \times \mathcal{S})$ such that $[Q, B]$ is a closed lterm of base type, and index$(B) \leq$ length(σ). We call the collection of all configurations $\mathcal{C}$. Formally we define a binary relation, $\rightarrow_{\mathcal{L}}$, on $\mathcal{C} \cup \mathcal{L}$. The intended interpretation of $\langle Q, B, \sigma \rangle \rightarrow_{\mathcal{L}} \langle Q', B', \sigma' \rangle$ is that in one step of the evaluation of Q (tied to state σ by binding B) results in Q' (tied to state σ' by binding B'). The intended interpretation of $M \rightarrow_{\mathcal{L}} M'$ is the usual one. We then make the further definition of $\twoheadrightarrow_{\mathcal{L}}$ as the reflexive transitive closure of $\rightarrow_{\mathcal{L}}$. Figure 2.2 lists the metavariable conventions which we make in giving the operational semantics. We attempt to follow them throughout the rest of this thesis. The full set of rules defining $\rightarrow_{\mathcal{L}}$ appear in Figures 2.3, 2.4 and 2.5.

θ	arbitrary types
β	base types (exp or comm)
M, N	terms of arbitrary type
E	terms of type exp
P	terms of type comm
Q	terms of base type (exp or comm)
α	(typed) identifiers of arbitrary type
α^{θ}	identifiers of type θ
ι	identifiers of type acc
κ	identifiers of type exp

Figure 2.2: Metavariable conventions.

2.4 Properties of $\rightarrow_{\mathcal{L}}$

There are some quite simple, but quite important properties of $\rightarrow_{\mathcal{L}}$ which we will use later. All but the last are verified by a simple induction on the structure of terms. In stating the properties, we assume that $\langle Q, B, \sigma \rangle$ is an arbitrary element of $\mathcal{C}$.

1. $\rightarrow_{\mathcal{L}}$ is deterministic, *viz.* it is the graph of a partial function on $\mathcal{C} \cup \mathcal{L}$.
2. If $\langle Q, B, \sigma \rangle \rightarrow_{\mathcal{L}} \langle Q', B', \sigma' \rangle$ then $B = B'$, length(σ) = length(σ'), and Q and Q' have the same type.

$$(\lambda\alpha.\, M)N \rightarrow_{\mathcal{L}} M[N/\alpha] \qquad \text{(beta)}$$

$$\frac{M \rightarrow_{\mathcal{L}} M'}{(MN) \rightarrow_{\mathcal{L}} (M'N)} \qquad \text{(eval-operator)}$$

$$\Upsilon_\theta M \rightarrow_{\mathcal{L}} M(\Upsilon_\theta M) \qquad \text{(rec-unwind)}$$

$$\frac{Q \rightarrow_{\mathcal{L}} Q'}{\langle Q, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle Q', B, \sigma\rangle} \qquad \text{(interaction)}$$

Figure 2.3: Functional rules.

3. If Q is not of type comm, then if $\langle Q, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle Q', B', \sigma'\rangle$ then $\sigma = \sigma'$ (*i.e.* only commands can have side effects).

4. Let $\alpha \notin \mathrm{FV}(Q)$ and $i \leq \mathrm{length}(\sigma)$, then

$$\langle Q, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle Q', B, \sigma'\rangle \quad \text{iff} \quad \langle Q, B[i/\alpha], \sigma\rangle \rightarrow_{\mathcal{L}} \langle Q', B[i/\alpha], \sigma'\rangle$$

   i.e. for $\alpha \notin \mathrm{FV}(Q)$, $B(\alpha)$ does not affect the behavior of the configuration $\langle Q, B, \sigma\rangle$.

5. If P is of type comm, then either

$$\begin{array}{ll} & \langle P, B, \sigma\rangle \rightarrow_{\mathcal{L}}^{*} \langle \mathsf{skip}, B, \sigma'\rangle \text{ for some } \sigma' \\ \text{or} & \langle P, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle P_1, B, \sigma_1\rangle \rightarrow_{\mathcal{L}} \langle P_2, B, \sigma_2\rangle \rightarrow_{\mathcal{L}} \cdots \end{array}$$

   In the first case we say "P (with binding B in state σ) converges." In the second case we say "P (with binding B in state σ) diverges."

The last property can be verified by defining an appropriate notion of normal form for closed lterms and noticing that the only normal forms of type comm have term part skip.

2.5 Observations

Now that we have defined our operational semantics and established some consistency properties of this semantics, we can introduce some notions of "behaviors" of programs. Of course, we first need to decide what a program is.

Definition 5 *An EoA* **program** *is an EoA command whose free variables are of type* exp *or* acc.

A binding provides the mechanism for linking a program to the state, possibly introducing sharing. We can view our operational semantics as specifying how

$$\langle \mathsf{succ}\, n, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle n+1, B, \sigma\rangle \qquad \text{(succ-do)}$$

$$\frac{\langle E, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle E', B, \sigma\rangle}{\langle \mathsf{succ}\, E, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \mathsf{succ}\, E', B, \sigma\rangle} \qquad \text{(succ-eval-arg)}$$

$$\langle (\mathsf{plus}\, n_1\, n_2), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle n_1 + n_2, B, \sigma\rangle \qquad \text{(plus-do)}$$

$$\frac{\langle E_1, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle E_1', B, \sigma\rangle}{\langle (\mathsf{plus}\, E_1\, E_2), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle (\mathsf{plus}\, E_1'\, E_2), B, \sigma\rangle} \qquad \text{(plus-eval-arg1)}$$

$$\frac{\langle E_2, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle E_2', B, \sigma\rangle}{\langle (\mathsf{plus}\, n\, E_2), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle (\mathsf{plus}\, n\, E_2'), B, \sigma\rangle} \qquad \text{(plus-eval-arg2)}$$

$$\langle (\mathsf{IFexp}_\beta\, 0\, Q_1\, Q_2), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle Q_1, B, \sigma\rangle \qquad \text{(IFexp-true)}$$

$$\langle (\mathsf{IFexp}_\beta\, n+1\, Q_1\, Q_2), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle Q_2, B, \sigma\rangle \qquad \text{(IFexp-false)}$$

$$\frac{\langle E, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle E', B, \sigma\rangle}{\langle (\mathsf{IFexp}_\beta\, E\, Q_1\, Q_2), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle (\mathsf{IFexp}_\beta\, E'\, Q_1\, Q_2), B, \sigma\rangle} \qquad \text{(IFexp-eval-guard)}$$

Figure 2.4: Plotkin Style SOS rules defining the operational semantics of EoA for ordinary non-imperative terms.

$$\langle (\iota\, n), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \mathrm{update}(\sigma, B(\iota), n)\rangle \qquad \text{(variable-write)}$$

$$\langle \kappa, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \sigma_{B(\kappa)}, B, \sigma\rangle \qquad \text{(variable-read)}$$

$$\frac{\langle E, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle E', B, \sigma\rangle}{\langle (\iota\, E), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle (\iota\, E'), B, \sigma\rangle} \qquad \text{(assign-eval-arg)}$$

$$\langle (\mathsf{seq}\, \mathsf{skip}\, P), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle P, B, \sigma\rangle \qquad \text{(seq-discharge)}$$

$$\frac{\langle P_1, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle P_1', B, \sigma'\rangle}{\langle (\mathsf{seq}\, P_1\, P_2), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle (\mathsf{seq}\, P_1'\, P_2), B, \sigma'\rangle} \qquad \text{(seq-eval-arg1)}$$

$$\langle (\mathsf{New}\, \iota,\ \kappa \leftarrow n\, \mathsf{in}\, \mathsf{skip}), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma\rangle \qquad \text{(New-discharge)}$$

$$\frac{\langle E, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle E', B, \sigma\rangle}{\langle (\mathsf{New}\, \iota,\ \kappa \leftarrow E\, \mathsf{in}\, P), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle (\mathsf{New}\, \iota,\ \kappa \leftarrow E'\, \mathsf{in}\, P), B, \sigma\rangle} \qquad \text{(New-init)}$$

$$\frac{\langle P, B', \sigma{\cdot}n\rangle \rightarrow_{\mathcal{L}} \langle P', B', \sigma'{\cdot}n'\rangle}{\langle (\mathsf{New}\, \iota,\ \kappa \leftarrow n\, \mathsf{in}\, P), B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle (\mathsf{New}\, \iota,\ \kappa \leftarrow n'\, \mathsf{in}\, P'), B, \sigma'\rangle} \qquad \text{(New-eval)}$$

where $l = 1 + \mathrm{length}(\sigma)$ *and* $B' = B[l/\iota, l/\kappa]$.

Figure 2.5: The new rules which need to be introduced in order to capture imperative features.

to associate with each closed lterm of type comm a partial function on states. Note that a closed lterm of type comm is merely a program P paired with a binding B such that $FV(P) \subseteq \mathrm{Dom}(B)$. Specifically, the partial function $f_{[P,B]}$ associated with $[P, B]$ is defined by:

$$f_{[P,B]}(\sigma) = \begin{cases} \sigma' & \text{if } \langle P, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle, \\ \textit{undefined} & \text{if } P \text{ with binding } B \text{ diverges in state } \sigma, \\ \textit{undefined} & \text{if } \mathrm{length}(\sigma) < \mathrm{index}(B) \\ & (\textit{viz.}\ \text{the state is not "large enough" to fit with } B). \end{cases}$$

It is the closed lterm $[P, B]$ which most closely coincides with the traditional notion of ALGOL-like program, with the binding B linking the program P to the state via the free exp and acc identifiers in P. There is a natural notion of equivalence between programs, $\approx^{\mathrm{P}}_{obs}$, which we formally define as follows

Definition 6 *Let P_1, P_2 be programs. Define $P_1 \approx^{\mathrm{P}}_{obs} P_2$ by the condition:*

For all B such that $[P_1, B]$ and $[P_2, B]$ are both closed lterms, $f_{[P_1,B]} = f_{[P_2,B]}$.

It requires a little bit of proof (using the previously mentioned properties of $\rightarrow_{\mathcal{L}}$) to show that $\approx^{\mathrm{P}}_{obs}$ is actually an equivalence relation. Unfortunately, the relation $\approx^{\mathrm{P}}_{obs}$ is not a congruence on programs, as free acceptors can be instantiated by "bad" variables (they could have side-effects). For example while

$$(\iota\ 1) \approx^{\mathrm{P}}_{obs} (\iota\ 9)\,;(\iota\ 1),$$

it is not hard to see how to write a term M of type acc such that in some binding B, we have $f_{[(M\ 1),B]} \neq f_{[(M\ 9)\,;(M\ 1),B]}$ and so

$$(M\ 1) \not\approx^{\mathrm{P}}_{obs} (M\ 9)\,;(M\ 1).$$

Given such a term M, the context $C[\cdot] = (\lambda\iota.[\cdot])M$ (a **context** is merely a term with a "hole") will obviously distinguish $(\iota\ 1)$ from $(\iota\ 9)\,;(\iota\ 1)$.

Definition 7 *Let $\equiv^{\mathrm{P}}_{obs}$ be the congruence on terms generated by $\approx^{\mathrm{P}}_{obs}$. Specifically, $M \equiv^{\mathrm{P}}_{obs} N$ iff for all contexts $C[\cdot]$ such that $C[M]$ and $C[N]$ are programs, $C[M] \approx^{\mathrm{P}}_{obs} C[N]$*

It again requires a little proof to show that $\equiv^{\mathrm{P}}_{obs}$ is in fact an equivalence relation, but requires no new ideas.

On the other hand one could make the argument that $\approx^{\mathrm{P}}_{obs}$ does not match with Reynolds's goal of eliminating locations from explicitly appearing in an ALGOL-like language. Essentially our notion of program amounts to observing open commands in a very special set of contexts. From a technical point of view, one might have expected a definition of **completely closed command**, *viz.* a command P such that $\mathrm{FV}(P) = \emptyset$. Our observation would then simply

be termination of a completely closed command. We now generate a definition of observational equivalence and observational congruence based on observing termination.

Definition 8 *Let P_1, P_2 be completely closed commands, define $P_1 \approx^{\mathsf{c}}_{obs} P_2$ by the condition:*

$$\begin{array}{ll} & \langle P_1, \textsc{EmptyBind}, \textsc{EmptyState}\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, \textsc{EmptyBind}, \textsc{EmptyState}\rangle \\ \textit{iff} & \langle P_2, \textsc{EmptyBind}, \textsc{EmptyState}\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, \textsc{EmptyBind}, \textsc{EmptyState}\rangle. \end{array}$$

The relation $\approx^{\mathsf{c}}_{obs}$ is obviously an equivalence relation; it is also a congruence on completely closed commands. We can generalize $\approx^{\mathsf{c}}_{obs}$ to a congruence on terms, by defining $M \equiv^{\mathsf{c}}_{obs} N$ by the condition that whenever $C[M]$ and $C[N]$ are completely closed commands, $C[M] \approx^{\mathsf{c}}_{obs} C[N]$. (It again requires a little proof to show that $\equiv^{\mathsf{c}}_{obs}$ is an equivalence relation.)

We have now defined two seemingly different notions of observational congruence, $\equiv^{\mathsf{p}}_{obs}$ and $\equiv^{\mathsf{c}}_{obs}$. We would like to claim that restricting observations to completely closed commands, rather than observing programs directly, does not change the congruence generated. In other words $\equiv^{\mathsf{p}}_{obs}$ and $\equiv^{\mathsf{c}}_{obs}$ define the same relation on terms. We prove this as the following theorem, which then justifies the use of the notation $\equiv_{obs}$.

Theorem 9 $M \equiv^{\mathsf{p}}_{obs} N \quad$ *iff* $\quad M \equiv^{\mathsf{c}}_{obs} N$.

Proof: The "only if" ($\Rightarrow$) direction is obvious. The proof of the "if" ($\Leftarrow$) direction requires the following further uniformity property of $\rightarrow_{\mathcal{L}}$ and the analogous version for acc, both of which are proven by a straightforward structural induction.

Lemma 10 *Suppose $B(\alpha_1^{\mathsf{exp}}) = B(\alpha_2^{\mathsf{exp}})$. Let $N \equiv M[\alpha_2^{\mathsf{exp}}/\alpha_1^{\mathsf{exp}}]$.*

$$\textit{If} \quad \langle M, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle M', B, \sigma'\rangle \quad \textit{then} \quad \langle N, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle M'[\alpha_2^{\mathsf{exp}}/\alpha_1^{\mathsf{exp}}], B, \sigma'\rangle.$$

Conversely, if $\langle N, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle N', B, \sigma'\rangle$ then there exists an M' such that

$$N' = M'[\alpha_2^{\mathsf{exp}}/\alpha_1^{\mathsf{exp}}] \quad \textit{and} \quad \langle M, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle M', B, \sigma'\rangle.$$

We now show sketch the proof that $M \not\equiv^{\mathsf{p}}_{obs} N$ implies $M \not\equiv^{\mathsf{c}}_{obs} N$. Let $C[\cdot]$ be a program context distinguishing M and N. Let B be a binding and σ a state demonstrating this distinction. Without loss of generality, suppose

$$\langle C[M], B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle \quad \text{but} \quad \langle C[N], B, \sigma\rangle \not\twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle$$

From the context $C[\cdot]$ we will construct a completely closing command context $C'[\cdot]$ which will allocate $\mathit{length}(\sigma)$ variables such that the ith allocation is initialized to the ith component of σ. The body will consist of an execution of $C[\cdot]$

in a manner such that the binding B is encoded by tying the free identifiers of $C[\cdot]$ to the variables allocated above. After properly executing $[\cdot]$, we check that the resulting state is σ', if it is the term halts, if it is not the term diverges.

To construct this context $C'[\cdot]$, let

$$\begin{array}{rrcl} & \{\alpha_1^{\mathsf{exp}}, \ldots, \alpha_k^{\mathsf{exp}}, \alpha_1^{\mathsf{acc}}, \ldots, \alpha_l^{\mathsf{acc}}\} & = & \mathrm{Dom}(B), \\ & n & = & \mathrm{length}(\sigma), \\ & \iota_1, \ldots, \iota_n & \text{be} & \text{fresh acceptors}, \\ \text{and} & \kappa_1, \ldots, \kappa_n & \text{be} & \text{fresh expressors}. \end{array}$$

We take the context $C'[\cdot]$ to be:

$$\begin{array}{l} \mathsf{New}\ \iota_1, \kappa_1 \leftarrow \mathrm{proj}(\sigma, 1) \\ \mathsf{in}\quad \mathsf{New}\ \iota_2, \kappa_2 \leftarrow \mathrm{proj}(\sigma, 2) \\ \qquad \mathsf{in} \\ \qquad\quad \vdots \\ \qquad\quad \mathsf{New}\ \iota_n, \kappa_n \leftarrow \mathrm{proj}(\sigma, n) \\ \qquad\quad \mathsf{in}\quad (\lambda \alpha_1^{\mathsf{exp}} \ldots \alpha_k^{\mathsf{exp}}, \alpha_1^{\mathsf{acc}}, \ldots \alpha_l^{\mathsf{acc}}.C[\cdot])\, \kappa_{B(\alpha_1^{\mathsf{exp}})} \cdots \kappa_{B(\alpha_k^{\mathsf{exp}})}\, \iota_{B(\alpha_1^{\mathsf{acc}})} \cdots \iota_{B(\alpha_l^{\mathsf{acc}})}\,; \\ \qquad\qquad (\mathsf{IFexp}_{\mathsf{comm}}\ (\kappa_1 \neq \mathrm{proj}(\sigma', 1))\ \mathit{diverge} \\ \qquad\qquad\qquad\quad (\mathsf{IFexp}_{\mathsf{comm}}\ (\kappa_2 \neq \mathrm{proj}(\sigma', 2))\ \mathit{diverge} \\ \qquad\qquad\qquad\qquad\quad \vdots \\ \qquad\qquad\qquad\qquad\quad (\mathsf{IFexp}_{\mathsf{comm}}\ (\kappa_n \neq \mathrm{proj}(\sigma', n))\ \mathit{diverge}\ \mathsf{skip}) \cdots)) \end{array}$$

The completely closed command $C'[M]$ will always converge, whereas the completely closed command $C'[N]$ will always diverge, thus $M \not\equiv^{\mathrm{c}}_{obs} N$. ■

Now that we have a satisfactory notion of observational congruence, we address what we mean when we say that a model is adequate for this notion of observational congruence. Jim and Meyer [6] provide a discussion of the basic principles behind the definitions of observations, observational congruence, adequacy and full abstraction. They also give some Lemmas which provide alternative characterizations of adequacy when a model possesses various special properties. The rest of this Section is adapted from that discussion.

A meaning function for EoA is a function $[\![\cdot]\!]$ from terms M to values in some space (for the Tennent model these values will be natural transformations between functors from a category of possible worlds to the category of (possibly) bottomless cpos). A meaning function is **compositional** iff for all terms M, N and contexts $C[\cdot]$, if $[\![M]\!] = [\![N]\!]$ then $[\![C[M]]\!] = [\![C[N]]\!]$.

Definition 11 *A meaning function is* **adequate** *for a definition of observational congruence, $\equiv_{obs}$, if for all terms M,N*

$$[\![M]\!] = [\![N]\!] \quad \textit{implies} \quad M \equiv_{obs} N.$$

Equality on adequate meanings may be strictly finer than observational congruence; in the ideal situation, known as full abstraction, the two relations coincide:

Definition 12 *Let* $[\![\cdot]\!]$ *be a meaning function for EoA. It is* fully abstract *if for all* M *and* N,

$$[\![M]\!] = [\![N]\!] \quad \textit{iff} \quad M \equiv_{obs} N$$

The main result of this thesis (proven in Chapter 4) is to show that the model of Tennent presented in Section 3.4 is adequate for EoA.

Chapter 3

Defining the Denotational Model

O'Hearn and Tennent [12] provide a good exposition of the methods of "possible worlds" models of block structure. This idea of using a category-theoretic form of possible worlds to model block structure was pioneered in the early 1980's by Reynolds and Oles [15, 17, 20]. The model of Reynolds and Oles provided great insight into the nature of local variable declarations. Later Tennent and O'Hearn [12–14, 23, 25] modified the Oles model in an effort to incorporate the notion of non-interference. Tennent incorporated this model of EoA which understood non-interference into a larger model of Reynolds's Specification Logic [22] for the Essence of ALGOL. Our focus right now is Tennent's original model of EoA which captures non-interference. In Chapter 5 we discuss some examples of equivalences of EoA code which O'Hearn and Tennent have shown [12, 13] that this model validates. These examples demonstrate the substantial power of the model; however, we also also show some counterexamples to full abstraction [12, 13].

We begin this Chapter by giving a review of the basic concepts of these possible worlds models of ALGOL and the definition of the Tennent model; a more comprehensive discussion is found in [12], Chapter 9 of [24] and in [14]. Section 3.5 contains a collection of definitions which we will need for our proof of adequacy in Chapter 4. We focus our discussion on the model of Tennent [24, 25]. This model incorporates several substantial modifications to Oles's original model. The Tennent model provides some simplifications in description and some improvements in power. In particular Tennent separated the issue of implicit coercions from the issue of block structure, which had been intertwined in the model of Oles. Tennent also found a few other technical simplifications of the category of state shapes which were used for modeling block structure. In addition, to properly model interference (which helps in understanding how

higher order objects behave) Tennent slightly modified the category of state shapes, and chose an "ad hoc" definition for $[\![\mathsf{comm}]\!]$, rather than taking the functor $S \multimap S$.

Later, we will also be interested in the O'Hearn-Tennent model [13]. This model arose from an attempt to model a principle of "non-interference abstraction." Technically it is obtained by slightly perturbing the category of state shapes used in [24, 25], and modifying the definition of $[\![\mathsf{comm}]\!]$. Most of the definitions and results of this paper make sense and hold for both the Tennent and O'Hearn-Tennent models; we will specifically point out when a definition or result is specific to one or the other. When we abstract away the specifica of Tennent's model and state our general adequacy Theorem it will be obvious that the O'Hearn-Tennent model is also adequate. It should be a routine exercise to varify that the the model of Reynolds and Oles (for a language without implicit coercions) also satisfies the sufficient conditions for adequacy.

3.1 Functor Category Models of Simply Typed Lambda Calculi

Given two categories $\mathbf{A}$ and $\mathbf{B}$, we define the **functor category** from $\mathbf{A}$ to $\mathbf{B}$ ($\mathbf{B}^{\mathbf{A}}$) to be the category whose objects are all covariant functors from $\mathbf{A}$ to $\mathbf{B}$ and whose morphisms are all natural transformations between covariant functors from $\mathbf{A}$ to $\mathbf{B}$. Composition in $\mathbf{B}^{\mathbf{A}}$ is the standard composition of natural transformations, and the identity morphism is simply the identity natural transformation. Note that some authors write $\mathbf{A} \to \mathbf{B}$ for $\mathbf{B}^{\mathbf{A}}$.

Let $\mathbf{D}$ be to the category of *possibly-bottomless* cpo's; a careful definition of $\mathbf{D}$ is given in Section 3.2. We will take advantage of the fact that for any small category $\mathbf{X}$, the functor category $\mathbf{D}^{\mathbf{X}}$ is cartesian closed. In [11] Nelson proves this for any complete cartesian closed category in the place of $\mathbf{D}$. The case of $\mathbf{D}$ was explicitly treated by Oles [15]. $\mathbf{D}^{\mathbf{X}}$ is also closed under denumerable products, a fact that makes life simpler, as there will be a single environment object, Env (rather than merely an environment object for each finite set of typed identifiers). Our ability to take advantage of denumerables products in this way is a consequence of identifiers coming explicitly with types. If the category were not closed by denumerable products we would have to count identifiers and introduce finite environments relative to a finite set of identifiers. Berry, Curien and Lévy provide a quick summary of the general methods of c.c.c. models of simply-typed lambda calculi and what they look like when we take advantage of the single environment object [3, Section 6]. Going along the general lines of interpreting a simply typed λ-calculus in a c.c.c., we interpret types by objects of the functor category via a function $[\![\cdot]\!]_{\text{type}}$ (remember, these objects are *functors* from $\mathbf{X}$ to $\mathbf{D}$). We can generate the function $[\![\cdot]\!]_{\text{type}}$ merely by specifying $[\![\beta]\!]_{\text{type}}$ for all of our base types β (for EoA, this is simply for

$\beta \in \{\mathsf{comm}, \mathsf{exp}\}$).

In contrast to an interpretation of the simply typed lambda calculus in a category of sets with structure, there is a lot more to the meaning of a type than a (structured) set of elements of that type. In this case a type is a functor from a category to a (potentially bottomless) cpo. It no longer makes sense to think of an element of the type T, per se. We can however, make sense of the notion of a *global element* (constant) of type T. It is a morphism, m, from the terminal object ($\mathbf{1}_{\mathbf{D}^{\mathbf{X}}}$) of $\mathbf{D}^{\mathbf{X}}$ to $[\![T]\!]_{\text{type}}$. So m is a *natural transformation* from $\mathbf{1}_{\mathbf{D}^{\mathbf{X}}}$ to $[\![T]\!]_{\text{type}}$. By definition, this is a function that assigns to every $\mathbf{X}$-object w, a morphism in $\mathbf{D}$, $m_w : \mathbf{1}_{\mathbf{D}^{\mathbf{X}}}(w) \to T(w)$, which satisfies certain uniformity properties. As we will see, $\mathbf{1}_{\mathbf{D}^{\mathbf{X}}}(w)$ will simply turn out to be the one-element cpo $\{*\}$, so we can in fact just think of m as a method of selecting a single element from each cpo $T(w)$. But the $\mathbf{D}$-morphisms which m picks for each $\mathbf{X}$-object w cannot be unrelated. In fact, we must have for any $f : w \xrightarrow{\mathbf{X}} w'$, that

$$m(w') = T(f); m(w),$$

where ";" denotes composition in "diagrammatic" order. Since each type is a functor into $\mathbf{D}$ (which is a category of sets with structure), it does make sense to talk about an element of a type T *at a world* w.

Half of the meaning of a type T is its behavior on objects, $w \in \text{obj}(\mathbf{X})$, essentially telling us the collection of meanings appropriate to type T at world w. As mentioned above, the meanings for different worlds are not completely unrelated. The other half of the meaning of a type tells us this relation. Specifically, given $f : w \xrightarrow{\mathbf{X}} w'$, the function $T(f)$ tells us how to transform meanings appropriate to world w into meanings appropriate to world w'. Since there may be many morphisms from w to w', there might in fact be many *different* ways to transform the space of meanings appropriate to world w into the space of meanings appropriate to world w'. As we will see in Section 3.5.2 this provides us with greater flexibility than we had with the "global elements of type T."

We also have an indexed product object

$$\text{Env} = \prod_{\alpha^\theta \in \mathit{Identifiers}} [\![\theta]\!]_{\text{type}},$$

useful for providing meanings for the free identifiers of a term M. Since Env is a functor, it is incorrect to think of it as a set of environments. Nevertheless, it does make sense to think of an element of Env at a world w, *viz.* an environment at world w. As with types, the behavior of Env on morphisms, $\text{Env}(f : w \xrightarrow{\mathbf{X}} w') : \text{Env}(w) \xrightarrow{\mathbf{D}} \text{Env}(w')$, tells how to transform (along f) an environment at w into an environment at w'.

We now look to the general method of c.c.c. interpretations to see what we should do with terms. We interpret a term $M : \theta$ as a morphism in $\mathbf{D}^{\mathbf{X}}$ from

Env to the object $[\![\theta]\!]_{\text{type}}$. In our framework, this will be a natural transformation from the functor Env to the functor $[\![\theta]\!]_{\text{type}}$, which will give us a nice commutative diagram which, for any $\mathbf{X}$-morphism $f : w \to w'$, looks like:

$$\begin{array}{ccccc} w & & \mathrm{Env}(w) & \xrightarrow{[M]w} & [\![\theta]\!]_{\text{type}}w \\ {\scriptstyle f}\downarrow & & {\scriptstyle \mathrm{Env}f}\downarrow & & \downarrow{\scriptstyle [\theta]_{\text{type}}f} \\ w' & & \mathrm{Env}(w') & \xrightarrow{[M]w'} & [\![\theta]\!]_{\text{type}}w' \end{array}$$

3.2 The Category D

We now review the basic structure of $\mathbf{D}$, the category whose objects are directed-complete, partially ordered, possibly bottomless sets (dcpos), and whose morphisms are continuous functions. Composition of morphisms is simply functional composition and the identity morphism on a dcpo A is simply the identity function on A.

The terminal objects of $\mathbf{D}$ are precisely the dcpos with exactly one element. We fix our attention on $\{*\}$ which we refer to as $\mathbf{1_D}$. The product of two dcpos $\langle A, \sqsubseteq_A \rangle$ and $\langle B, \sqsubseteq_B \rangle$ is the cartesian product of A and B ordered componentwise, *i.e.* $\langle A \times B, \sqsubseteq_{A\times B} \rangle$, where $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$ and $(a, b) \sqsubseteq_{A\times B} (a', b')$ iff $a \sqsubseteq_A a'$ and $b \sqsubseteq_B b'$. The exponent $\langle A, \sqsubseteq_A \rangle \to \langle B, \sqsubseteq_B \rangle$ is defined to be $\langle A \to_c B, \sqsubseteq_{A\to_c B} \rangle$, where $A \to_c B$ is the set of continuous (*wrt.* $\sqsubseteq_A$ and $\sqsubseteq_B$) functions from A to B, and $\sqsubseteq_{A\to_c B}$ is the standard "pointwise ordering"—namely $f \sqsubseteq_{A\to_c B} g$ iff for all $a \in A$, $f(a) \sqsubseteq_B g(a)$. From now on we write $\to$ for $\to_c$, A for the dcpo $\langle A, \sqsubseteq_A \rangle$, and when obvious from the context, we drop the A from $\sqsubseteq_A$.

Two other constructions in $\mathbf{D}$ that we will later be using are:

- Lifting: $A_\perp$ is an copy of A augmented by a new least element $\perp$
- Partial exponentiation: $A \rightharpoonup B$ is the set of all continuous partial functions from A to B, ordered pointwise. Note that in $\mathbf{D}$, $A \rightharpoonup B$ is isomorphic to $A \to B_\perp$.

3.3 Constructing the C.C.C.

We now define the analogous constructions in the category $\mathbf{D^X}$. These definitions are taken almost directly from [24, Chapter 9], although throughout the presentations we are careful to maintain the distinction between $\to$ (a morphism in some category) and $\Rightarrow$ (the internal hom-functor of some category). In this

Section we decorate $\rightarrow$ with the category in which it lies, *e.g.* $f : x \xrightarrow{\mathbf{X}} y$ is a morphism in the category $\mathbf{X}$ from x to y. We also decorate $\Rightarrow$ with the category in which it lies, *e.g.* $A \Rightarrow_{\mathbf{D}} B$ is the object of $\mathbf{D}$ which is the cpo of continuous functions from A to B. After this Section, we will typically drop the decorations on $\rightarrow$ and $\Rightarrow$, and may in fact even use $\rightarrow$ for $\Rightarrow$ when it can easily be disambiguated from the context. When working in $\mathbf{D}$, we will also leave implicit the isomorphism between $f : d_1 \xrightarrow{\mathbf{D}} d_2$ as a continuous function from d_1 to d_2, and $f \in (d_1 \Rightarrow_{\mathbf{D}} d_2)$ an element of the cpo which is the exponential object.

Much of the structure of these constructions in $\mathbf{D}^{\mathbf{X}}$ is inherited from that of $\mathbf{D}$. Take $F, G \in \mathrm{obj}(\mathbf{D}^{\mathbf{X}})$, namely functors from $\mathbf{X}$ to $\mathbf{D}$; $x, y, z \in \mathrm{obj}(\mathbf{X})$, namely arbitrary possible worlds; and let $f : x \xrightarrow{\mathbf{X}} y$, $g : y \xrightarrow{\mathbf{X}} z$ be arbitrary $\mathbf{X}$-morphisms from x to y and y to z, respectively. The constructions are then as follows:

- Terminal object, $\mathbf{1}_{\mathbf{D}^{\mathbf{X}}}$ (remember $\mathbf{1}_{\mathbf{D}} \equiv \{*\}$): The behavior on worlds is $\mathbf{1}_{\mathbf{D}^{\mathbf{X}}}(x) = \mathbf{1}_{\mathbf{D}} \equiv \{*\}$ and the behavior on morphisms is $\mathbf{1}_{\mathbf{D}^{\mathbf{X}}}(f) = \mathrm{id}_{\mathbf{1}_{\mathbf{D}}} \equiv \mathrm{id}_{\{*\}}$.
- The product object, $F \times_{\mathbf{D}^{\mathbf{X}}} G$: On objects $(F \times_{\mathbf{D}^{\mathbf{X}}} G)(x) = F(x) \times_{\mathbf{D}} G(x)$, ordered componentwise. On morphisms, we have
$$(F \times_{\mathbf{D}^{\mathbf{X}}} G)(f)\big[\langle d, d' \rangle \in F(x) \times_{\mathbf{D}} G(x)\big] = \langle F(f)d, G(f)d' \rangle.$$
- The exponential object, $F \Rightarrow_{\mathbf{D}^{\mathbf{X}}} G$:
$$(F \Rightarrow_{\mathbf{D}^{\mathbf{X}}} G)(x) = \left\{ m \in \prod_{f : x \xrightarrow{\mathbf{X}} y} [F(y) \Rightarrow_{\mathbf{D}} G(y)] \;\middle|\; \begin{array}{l} \text{for all } f : x \xrightarrow{\mathbf{X}} y \text{ and } g : y \xrightarrow{\mathbf{X}} z, \\ m(f); G(g) = F(g); m(f; g) \end{array} \right\}$$
ordered pointwise (*i.e.* $m_1 \sqsubseteq m_2$ iff $m_1(f) \sqsubseteq m_2(f)$ for every $f : x \rightarrow y$), where
$$\prod_{f : x \xrightarrow{\mathbf{X}} y} \cdots y \cdots$$
here and in later definitions is an abuse of notation which stands for
$$\prod_{f \in \mathbf{X}(x, \cdot)} \cdots \mathrm{codom}\, f \cdots,$$
where $\mathbf{X}(x, \cdot)$ is the set of all $\mathbf{X}$-morphisms with domain x. The behavior of this construction on $\mathbf{X}$-morphisms, $f : x \xrightarrow{\mathbf{X}} y$, is defined to be:
$$(F \Rightarrow_{\mathbf{D}^{\mathbf{X}}} G)(f : x \xrightarrow{\mathbf{X}} y)(m \in [F \Rightarrow_{\mathbf{D}^{\mathbf{X}}} G]x)(g : y \xrightarrow{\mathbf{X}} z) = m(f; g)$$
Tennent provides the following motivation for the exponential construction:

> To motivate the $\Rightarrow$ construction, consider that a procedure defined in possible world x might be called in any possible world y accessible from x using any $\mathbf{X}$-morphism $f : x \xrightarrow{\mathbf{X}} y$, and it is the domain structure determined by y which should be in effect when the procedure body is executed. This suggests that we cannot just define $(F \Rightarrow_{\mathbf{D}^{\mathbf{X}}} G)(x)$ to be $F(x) \Rightarrow_{\mathbf{D}} G(x)$; the meaning of a procedure defined in possible world x must be a *family* of functions, indexed by $\mathbf{X}$-morphisms $f : x \xrightarrow{\mathbf{X}} y$. But such families of functions must be appropriately *uniform*; ... [24].

The uniformity condition is the commutativity of all diagrams (in $\mathbf{D}$) of the form

$$\begin{array}{ccc} F(y) & \xrightarrow{m(f:x \xrightarrow{\mathbf{X}} y)} & G(y) \\ {\scriptstyle F(g:y \xrightarrow{\mathbf{X}} z)}\downarrow & & \downarrow{\scriptstyle G(g:y \xrightarrow{\mathbf{X}} z)} \\ F(z) & \xrightarrow{m((f;g):x \xrightarrow{\mathbf{X}} z)} & G(z) \end{array} \tag{3.1}$$

There are several other useful constructions, which are not directly a part of the c.c.c. structure of $\mathbf{D}^{\mathbf{X}}$. For example, we have the lifted object, $F_{\perp_{\mathbf{D}^{\mathbf{X}}}}$:

$$\begin{aligned} F_{\perp_{\mathbf{D}^{\mathbf{X}}}}(x) &= [F(x)]_{\perp_{\mathbf{D}}} \\ F_{\perp_{\mathbf{D}^{\mathbf{X}}}}(f)[d \in F_{\perp_{\mathbf{D}}}(x)] &= \begin{cases} \perp, & \text{if } d = \perp, \\ F(f)(d) \text{ in } F_{\perp_{\mathbf{D}}}(y) & \text{otherwise.} \end{cases} \end{aligned}$$

We also have a construction for *partial* exponentiation, which will be similar to the construction for (total) exponentiation; however, the uniformity condition on elements of the resulting cpo will not be as stringent. Specifically, we have

$$(F \rightharpoonup_{\mathbf{D}^{\mathbf{X}}} G)(x) = \left\{ m \in \prod_{f : x \xrightarrow{\mathbf{X}} y} [F(y) \rightharpoonup_{\mathbf{D}} G(y)] \;\middle|\; \begin{array}{l} \text{for all } f : x \xrightarrow{\mathbf{X}} y \text{ and } g : y \xrightarrow{\mathbf{X}} z, \\ m(f); G(g) \subseteq F(g); m(f;g) \end{array} \right\}$$

ordered pointwise (*i.e.* $m_1 \sqsubseteq m_2$ iff $m_1(f) \sqsubseteq m_2(f)$ for every $f : x \to y$). Notice the two differences in the definition of $\rightharpoonup_{\mathbf{D}^{\mathbf{X}}}$ from the definition of $\Rightarrow_{\mathbf{D}^{\mathbf{X}}}$: we do comprehension over the underlying set of *partial* continuous functions from $F(x)$ to $G(x)$, and we have changed the "$=$" to a "$\subseteq$" in giving the constraint $m(f); G(g) \subseteq F(g); m(f;g)$. The effect of this second change is to weaken the

uniformity condition suggested in Diagram 3.1 to only require commutativity when the result of the partial mapping along the top of the Diagram is defined, giving Diagram 3.2.

$$
\begin{array}{ccc}
F(y) & \xrightarrow{m(f:x\xrightarrow{\mathbf{X}}y)} & G(y) \\
{\scriptstyle F(g\,:\,y\xrightarrow{\mathbf{X}}z)}\downarrow & \subseteq & \downarrow{\scriptstyle G(g\,:\,y\xrightarrow{\mathbf{X}}z)} \\
F(z) & \xrightarrow{m((f;g):x\xrightarrow{\mathbf{X}}z)} & G(z)
\end{array}
\tag{3.2}
$$

Note that, even though in $\mathbf{D}$, $A \Rightarrow B_\perp$ is isomorphic to $A \multimap B$, this is not generally the case in $\mathbf{D}^{\mathbf{X}}$.

Similarly, we can introduce indexed products. Specifically, if I is a denumerable set and for every $i \in I$, $F[i]$ is a functor from $\mathbf{X}$ to $\mathbf{D}$, then we can define the object, $\prod_{i\in I}^{\mathbf{D}^{\mathbf{X}}} D_i$, by

$$
\begin{aligned}
\left(\prod_{i\in I}^{\mathbf{D}^{\mathbf{X}}} F[i]\right)(x \in \mathrm{obj}(\mathbf{X})) &= \prod_{i\in I}^{\mathbf{D}} (F[i]w) \\
\left(\prod_{i\in I}^{\mathbf{D}^{\mathbf{X}}} F[i]\right)(f : x\xrightarrow{\mathbf{X}}x') &= \prod_{i\in I}^{\mathbf{D}} (F[i]f)
\end{aligned}
$$

3.4 A Model of EoA

3.4.1 The Category of Possible Worlds, W

The intuition behind the category of Possible Worlds is to provide the "shape of the state" (*i.e.* the set of allowable states) during program execution, and to show how one state shape evolves into another. The category of possible worlds has state shapes as objects and "store evolutions" as morphisms. There are two principle kinds of operations performed on state shapes in Tennent's work (although many others are allowed). The first is an "expansion" to correspond to allocating a new variable. The other is a "restriction" operation by which we restrict the set of allowable states to include only those states which satisfy a certain property.

For our model of EoA we will use as the category of possible worlds, $\mathbf{W}$, a category of state shapes. Specifically, an object of $\mathbf{W}$ is any subset of $\mathbb{N}^n$ for any integer n. The set of objects of $\mathbf{W}$ is closed under subset, intersection and product (if we interpret product as concatenation of sequences). A morphism from the world X to the world Y is a pair (f, Q), where

1. f is a function from the set Y to the set X.
2. Q is an equivalence relation on the set Y.

3. f restricted to any Q-equivalence class is injective; *i.e.*, for all $y, y' \in Y$ if yQy' and $f(y) = f(y')$ then $y = y'$.

When (f, Q) is describing a pure "restriction" operation, Y will be a subset of X. Furthermore, Q will turn out to be T_Y (the everywhere-true binary relation on Y), and f will simply be the injection of Y into X. When (f, Q) describes a new variable allocation, Y will be $X \times \mathbb{N}$, Q will be defined so that

$$(x_0 . n_0) Q (x_1 . n_1) \quad \text{iff} \quad n_0 = n_1,$$

and $f(x.n) = x$. Intuitively, f extracts the old portion of the stack which is embedded in the new one, and Q relates new stacks which have the same "new components" (but possibly differ on the old part).

We define composition (in diagrammatic order) of **W**-morphisms, $(f, Q) : X \rightarrow Y$, and $(g, R) : Y \rightarrow Z$ as follows: $(f, Q); (g, R) = (h, P)$ where:

- $h(z) = f(g(z))$,
- $z_0 P z_1$ iff $z_0 R z_1$ and $g(z_0) Q g(z_1)$.

Thus the identity morphism of **W** at world X, $\mathrm{id}_X : X \rightarrow X$, will be (I_X, T_X), where I_X is the identity function on X, and T_X is the everywhere-true binary relation on X.

We introduce the following useful notation for abbreviating some common restriction morphisms. Specifically, if X' is a subset of a world X, we have the **state-set restriction** morphism, $\lceil X' : X \rightarrow X'$, defined to be $(f, T_{X'})$, where f is the inclusion function from X' to X and $T_{X'}$ is the everywhere-true relation on X' (The notation $\lceil X'$ does not completely specify a morphism of **W**, as it also depends on the object X. Thus we will make sure that the X in question will always be clear from context). Similarly we introduce a useful notation for abbreviating some common expansion morphisms. Specifically, we define morphisms $\times\mathbb{N} : X \rightarrow X \times \mathbb{N}$, for which the function part is the projection from $X \times \mathbb{N}$ to X, and the equivalence relation part relates $x_0 . n_0$ and $x_1 . n_1$ iff $n_0 = n_1$. As with our notation for $\lceil X'$, although the notation $\times\mathbb{N}$ does not identify a morphism uniquely (as we need to know which object X is), the intended morphism will always be evident from the context. In future discussions, we will let the metavariables $X, Y, Z, \ldots$ range over objects of **W**, and we will let $x_0, x_1, \ldots$ range over X, etc.

3.4.2 Interpreting Types

The base types of EoA are exp and comm. An integer thunk (exp) is essentially a function which given a store, returns an integer—or is undefined. A command is essentially a function which given a store returns another store—or is undefined. Notice that in describing both of our base types, we have functions of stores arising. In fact, in our semantics it is only in the definition of the base types

that functions on stores arise directly. Given our intuitive description of exp and comm it seems that we should first describe a meaning for stores and for integers, and then we are looking for something like: $[\![\mathsf{exp}]\!] = S \Rightarrow \mathbb{N}$, $[\![\mathsf{comm}]\!] = S \Rightarrow S$, and for higher types: $[\![\theta \to \tau]\!] = [\![\theta]\!] \Rightarrow [\![\tau]\!]$.

So what are the functors S and $\mathbb{N}$? S is the **contravariant** functor which, given a store shape X, gives a discretely-ordered dcpo with X as its underlying set. In order to avoid getting bogged down in notation, we will often simply write X to denote $S(X)$. To define S on morphisms, we take $S(f, Q : X \xrightarrow{\mathbf{W}} Y) = f : S(Y) \xrightarrow{\mathbf{D}} S(X)$, the projection from Y to X along f. The functor $\mathbb{N}$ is the constant functor which in any world returns the set of non-negative integers, considered as a discrete dcpo. $\mathbb{N}$ applied to any morphism simply returns the identity on the natural numbers.

Note that since S is a *contravariant* functor, it is not an object of $\mathbf{D}^{\mathbf{W}}$. O'Hearn and Tennent [12, Section 4] give a good explanation of why we need S to be a contravariant functor. We adapt the conclusion of their discussion to our setting. There is a variant of the exponential construction used in $\mathbf{D}^{\mathbf{W}}$ which works on contravariant functors to yield a covariant functor (which is an object in $\mathbf{D}^{\mathbf{W}}$). The definition of the **contra-exponentiation** for $S \Rightarrow \mathbb{N}_\perp$ is defined just like the usual exponential, but with a reversal of vertical arrows in uniformity diagrams to account for contravariance. That is, $e(\cdot) \in (S \Rightarrow \mathbb{N}_\perp)X$ is a family of functions, indexed by morphisms out of X, such that

$$\begin{array}{ccc} Y & \xrightarrow{e(f)} & \mathbb{N}_\perp \\ {\scriptstyle S(g)}\uparrow & & \uparrow{\scriptstyle \mathrm{id}_{\mathbb{N}_\perp}} \\ Z & \xrightarrow{e(f;g)} & \mathbb{N}_\perp \end{array}$$

commutes, where $f : X \to Y$, $g : Y \to Z$. The morphism part manages to be defined as before, namely $(S \Rightarrow \mathbb{N}_\perp) f m g = m(f; g)$, thereby yielding a covariant functor.

We could try doing the analogous flipping of the arrows of Diagram 3.2 for defining $[\![\mathsf{comm}]\!] = S \to S$, giving the following:

$$\begin{array}{ccc} Y & \xrightarrow{m(f : x \xrightarrow{\mathbf{X}} y)} & Y \\ {\scriptstyle S(g\,:\,y \xrightarrow{\mathbf{X}} z)}\uparrow & \supseteq & \uparrow{\scriptstyle S(g\,:\,y \xrightarrow{\mathbf{X}} z)} \\ Z & \xrightarrow{m((f;g) : x \xrightarrow{\mathbf{X}} z)} & Z \end{array} \tag{3.3}$$

Unfortunately, as discussed in [13], this does not impose enough uniformity constraints on the behavior of command meanings. On the other hand, using

$S \Rightarrow S_\perp$ imposes uniformity constraints that are too strong. They find a good compromise by making $[\![\mathsf{comm}]\!]$ a subfunctor of $S \multimap S$ as follows:

$$[\![\mathsf{comm}]\!](X) = \left\{ c \in (S \multimap S)(X) \;\middle|\; \begin{array}{r} \text{for all } (f,Q) : X \xrightarrow{\mathbf{W}} Y, \text{ and } (g,R) : Y \xrightarrow{\mathbf{W}} Z, \text{ and } z \in Z, \\ S(\lceil Z'); c((f,Q);(g,R)) = c((f,Q);(g,R)); \lceil Z'); S(\lceil Z'), \\ \text{where } Z' = \{z' \mid zRz'\} \end{array} \right\}.$$

The extra condition above enforces an additional commutativity requirement arising from the equivalence-class component of the morphisms. Specifically, for any $(f,Q) : X \to Y$, and $y \in Y$, let

$$Y' = \{y' \in S(Y) \mid yQy'\}$$

(the set of states Q-equivalent to y); then

$$\begin{array}{ccc} Y & \xrightarrow{c(f,Q)} & Y \\ {\scriptstyle S(\lceil Y')}\uparrow & & \uparrow{\scriptstyle S(\lceil Y')} \\ Y' & \xrightarrow{c((f,Q);\lceil Y')} & Y' \end{array}$$

must commute (and not just semi-commute). This insures that, when it is defined, $c(f,Q)$ preserves the Q-equivalence class of its argument.

For the behavior of $[\![\mathsf{comm}]\!]$ on morphisms, we still have:

$$[\![\mathsf{comm}]\!](f)(c)(g) = c(f;g)$$

There is one other important special case of the commutativity property for elements of $[\![\mathsf{comm}]\!]X$, which is a consequence of the "semi-commutativity" of $(S \multimap S)X$. This special case is expressed by the following Lemma.

Lemma 13 *Let $c \in ([\![\mathsf{comm}]\!])(X)$ or, more generally, $c \in (S \multimap S)X$. Let $(f,Q) : X \to Y$. Finally, let $(g,R) : Y \to Z$ be an isomorphism in the category of possible worlds (then g is a bijection from Z to Y, and R is T_Z—the everywhere-true binary relation on Z). The following diagram commutes:*

$$\begin{array}{ccc} Y & \xrightarrow{c(f,Q)} & Y \\ {\scriptstyle S(g,R)=g}\uparrow & & \uparrow{\scriptstyle S(g,R)=g} \\ Z & \xrightarrow{c(f,Q;g,R)} & Z \end{array}$$

In other words, $c(f,Q)(g(z)) = g(c(f,Q;g,R)z)$.

Since g is a bijection, the constraint expressed by the commutative diagram can also be expressed as $c((f,Q);(g,R))z = g^{-1}(c(f,Q)(g(z)))$.

3.4.3 Interpreting Constants and Terms

In Chapter 2, we defined the syntax of EoA by giving some constants and three primary ways of constructing new terms from old: λ-abstraction, application, and new-variable declaration. To define our semantics we will need to provide the meaning of the constants, and we will also need to provide a way of finding the meaning of terms constructed using λ-abstraction, application and new-variable declarations. The general methods of providing these interpretations using c.c.c.'s is very well established, except for the case of new-variable declarations. For concreteness, we show specifically what all of these look like for our model of EoA.

The meaning of a term of type θ is a morphism from Env to $[\![\theta]\!]$. For the meanings of constants we directly give $[\![n]\!] : \mathrm{Env} \xrightarrow{\mathbf{D^W}} [\![\mathsf{exp}]\!]$ and $[\![\mathsf{skip}]\!] : \mathrm{Env} \xrightarrow{\mathbf{D^W}} [\![\mathsf{comm}]\!]$. Assuming X and Y are arbitrary worlds, $u \in \mathrm{Env}(X)$, $f : X \xrightarrow{\mathbf{W}} Y$ and $y \in Y$, the definitions are

$$\begin{aligned} [\![n]\!] X u f y &= n \\ [\![\mathsf{skip}]\!] X u f y &= y. \end{aligned}$$

We uncurry all of the other constants in order to simplify the presentation. For a constant c of type $\theta_1 \to \cdots \to \theta_n \to \tau$, we indirectly give its meaning (which is a morphism from Env to $\theta_1 \to \cdots \to \theta_n \to \tau$) by providing a morphism from $\theta_1 \times \cdots \times \theta_n$ to τ. These interpretations are shown in Figure 3.1, where X and Y are arbitrary worlds, $f : X \xrightarrow{\mathbf{W}} Y$ is arbitrary, and y is an arbitrary element of Y.

To obtain

$$[\![c^{\theta_1 \to \cdots \to \theta_n \to \tau}]\!] : \mathrm{Env} \to [\![\theta_1 \to \cdots \to \theta_n \to \tau]\!]$$

from

$$[\![c^{\theta_1 \to \cdots \to \theta_n \to \tau}]\!]_{uncurry} : ([\![\theta_1]\!] \times [\![\theta_n]\!]) \to [\![\tau]\!]$$

we take advantage of the *curry* isomorphism, between the hom sets $hom([\![\theta_1]\!] \times \cdots \times [\![\theta_n]\!], [\![\tau]\!])$ and $hom(\mathbf{1}, [\![\theta_1]\!] \Rightarrow \cdots [\![\theta_n]\!] \Rightarrow [\![\tau]\!])$. So, we will end up with

$$[\![c^{\theta}]\!] = !\mathrm{Env}; (\mathrm{curry}([\![c^{\theta}]\!]_{uncurry})).$$

For concreteness, we show how to obtain $[\![\mathsf{plus}]\!]$ from $[\![\mathsf{plus}]\!]_{uncurry}$.

$$[\![\mathsf{plus}]\!](X)(u)(f)(e_1)(g)(e_2) = [\![\mathsf{plus}]\!]_{uncurry} Z \langle [\![\mathsf{exp}]\!](f; g) e_1, [\![\mathsf{exp}(g) e_2]\!] \rangle,$$

and so

$$\begin{aligned} & [\![\mathsf{plus}]\!](X)(u)(f)(e_1)(g)(e_2)(h)(w) \\ &= [\![\mathsf{plus}]\!]_{uncurry} Z \langle [\![\mathsf{exp}]\!](f; g) e_1, [\![\mathsf{exp}(g) e_2]\!] \rangle h w \\ &= \begin{cases} n + m & \text{if} \quad e_1(f; g; h) w = n \quad \text{and} \quad e_2(g; h) w = m, \\ \bot & \text{if} \quad e_1(f; g; h) w = \bot \quad \text{or} \quad e_2(g; h) w = \bot, \end{cases} \end{aligned}$$

$$
\begin{aligned}
[\![\mathsf{IFexp}_{\mathsf{exp}}]\!]_{uncurry}X\langle e_0,e_1,e_2\rangle fy &= \begin{cases} e_1fy, & \text{if } e_0fy=0,\\ e_2fy, & \text{if } e_0fy=n+1,\\ \bot, & \text{otherwise.}\end{cases}\\
[\![\mathsf{IFexp}_{\mathsf{comm}}]\!]_{uncurry}X\langle e,c_1,c_2\rangle fy &= \begin{cases} c_1fy, & \text{if } efy=0,\\ c_2fy, & \text{if } efy=n+1,\\ \text{undefined}, & \text{otherwise.}\end{cases}\\
[\![\mathsf{succ}]\!]_{uncurry}Xefy &= \begin{cases} n+1, & \text{if } efy=n,\\ \bot & \text{if } efy=\bot.\end{cases}\\
[\![\mathsf{plus}]\!]_{uncurry}X\langle e_1,e_2\rangle fy &= \begin{cases} n+m, & \text{if } e_1fy=n \text{ and } e_2fy=m,\\ \bot & \text{if } e_1fy=\bot \text{ or } e_2fy=\bot.\end{cases}\\
[\![\mathsf{seq}]\!]_{uncurry}X\langle c_1,c_2\rangle fy &= \begin{cases} c_2fy', & \text{if } c_1fy=y',\\ \text{undefined}, & \text{if } c_1fy \text{ is undefined.}\end{cases}\\
[\![\mathsf{Y}_\theta]\!]_{uncurry}Xm &= \text{the least fixed point of } m(\mathrm{id}_X).
\end{aligned}
$$

Figure 3.1: Meanings of EoA constants

In the case of Y_θ it is not obvious that the specified least fixed points always exist, as the objects of $\mathbf{D}^{\mathbf{W}}$ are *arbitrary* functors into $\mathbf{D}$. Since many objects of $\mathbf{D}$ are *bottomless*, it is definitely not the case for every element m of an arbitrary functor F at an arbitrary world X that $m(\mathrm{id}_X)$ has a least fixed point. Moreover, it is not necessarily the case for arbitrary worlds X and Y that the fixed points if they were to exist would fit together well enough for $[\![\mathsf{Y}_F]\!]$ to be a natural transformation (a morphism in $\mathbf{D}^{\mathbf{W}}$). In [14, pages 26-28], it is argued that for all of the types we use (and some more), all of the necessary least fixed points do exist, and they do fit together naturally.

Application and abstraction are understood using the standard c.c.c. structure of $\mathbf{D}^{\mathbf{W}}$. When $M:\theta\to\tau$ and $N:\theta$, $[\![(M\ N)]\!]$ is a natural transformation from Env to $[\![\tau]\!]$. Specifically, for an arbitrary world X and $u\in\mathrm{Env}(X)$, then

$$[\![(M\ N)]\!]Xu = [\![M]\!]xu\,\mathrm{id}_x([\![N]\!]Xu).$$

We can see how this comes from the c.c.c. structure of $\mathbf{D}^{\mathbf{W}}$ by observing that the definition of $\mathrm{eval}_{A,B}:[A\Rightarrow B]\times A\xrightarrow{\mathbf{D}^{\mathbf{W}}}B$, is:

$$\mathrm{eval}_{A,B}\,X\langle f,a\rangle = f\,\mathrm{id}_X\,a.$$

The meaning of the term $(\lambda\alpha^\theta.\,M):\theta\to\tau$ is a natural transformation from Env to $[\![\theta\to\tau]\!]$. Taking X and u as before, $f:X\to Y$ and $v\in[\![\theta]\!]Y$, we have:

$$[\![(\lambda\alpha^\theta.\,M)]\!]Xufv = [\![M]\!]Y((\mathrm{Env}(f)u)[v/\alpha^\theta]).$$

We do a little bit of preliminary work before giving the valuation for **New** blocks. First, we define the new acceptor and expressor which are allocated in world X. Formally we define the functions:

$$\begin{aligned} a_{\text{new}} &: \text{obj}(W) \to \bigcup_{X \in \text{obj}(W)} [\![\mathsf{acc}]\!](X \times \mathbb{N}) \\ e_{\text{new}} &: \text{obj}(W) \to \bigcup_{X \in \text{obj}(W)} [\![\mathsf{exp}]\!](X \times \mathbb{N}) \end{aligned}$$

Given arbitrary worlds X, Y, Z, and arbitrary morphisms $(f, Q) : X \to Y$, $(g, R) : Y \to Z$, and given states $y \in Y$, $z \in Z$, we let

$$\begin{aligned} a_{\text{new}}(X)(f,Q)(e)(g,R)z &= z' \\ e_{\text{new}}(X)(f,Q)y &= v \quad \text{where} \quad g(y) = x.v. \end{aligned}$$

In the definition of a_{new}, z' is the unique element of Z such that

1. $f(g(z')) = x.(e(g,R)z)$, provided $x.n = f(g(z))$. If $e(g,R)z = \perp$ then a_{new} is undefined.
2. zRz' and $g(z)Qg(z')$.

There need not be a state z' satisfying the above two conditions, in which case the result is undefined; however, if such a state does exist, it must be unique by the injectivity property of $(f, Q); (g, R)$.

We further abuse the notation "$\times\mathbb{N}$" and define it as a function from **W**-morphisms to **W**-morphisms. Recall that we also use the notation $\times\mathbb{N}$ as a function from **W**-objects to **W**-objects. We also have morphisms $\times\mathbb{N} : X \to X \times \mathbb{N}$ for which the function part is the projection from $X \times \mathbb{N}$ to X, and the equivalence relation part relates $x_0.n_0$ and $x_1.n_1$ iff $n_0 = n_1$. These three aspects fit together quite closely; in fact, if we just viewed $\times\mathbb{N}$ via its behavior on **W**-objects and **W**-morphisms, then $\times\mathbb{N}$ would be a functor from **W** to **W**. Given $(f, Q) : X \to Y$, we define $(f, Q) \times \mathbb{N} = (f', Q')$, where $f'(x.n) = x$, and $\langle x_0.n_0\rangle Q' \langle x_1.n_1\rangle$ iff $x_0 Q x_1$. A valuable property of how they fit together is that for any **W**-morphism $(f, Q) : X \to Y$, the following diagram commutes:

$$\begin{array}{ccc} X & \xrightarrow{\times\mathbb{N}} & X \times \mathbb{N} \\ {\scriptstyle (f,Q)}\downarrow & & \downarrow{\scriptstyle (f,Q)\times\mathbb{N}} \\ Y & \xrightarrow{\times\mathbb{N}} & Y \times \mathbb{N} \end{array}$$

The valuation for New blocks is then:

$$[\![\mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P]\!]Xufy = S(\times\mathbb{N})\Big([\![P]\!](X\times\mathbb{N})u'(f\times\mathbb{N})\big(y.([\![E]\!]Xufy)\big)\Big)$$

where $u' = \big(\mathrm{Env}(\times\mathbb{N})u\big)[a_{new}(X)/\iota, e_{\mathrm{new}}(X)/\kappa]$.

This definition is essentially the same as that of [25], except there is a little bit of extra work here. The extra work arises from two particular aspects of EoA which are different from Reynolds's essence of ALGOL: the initial value for the location given by E and taking acc as a type synonym for exp $\rightarrow$ comm rather than taking it in such a way as to have $[\![\mathsf{acc}]\!] = \mathbb{N} \rightarrow [\![\mathsf{comm}]\!]$.

3.5 Definitions Needed for Adequacy

Given our style of operational semantics for EoA, we will also need to assign meanings to configurations. This will require several more steps. The state in the configuration provides us with two important pieces of semantic information. First, the possible world in which to interpret the term part of the configuration will depend on the length of the state. Second, the state component of the configuration gives us the semantic state to which we apply the meaning of the term part of the configuration. We then show how to turn a binding into its semantic analogue—an environment. In the last Section, we show how to combine these extra semantic pieces of information with the meaning of the term in order to figure out the denotation of the configuration.

3.5.1 A Sequence of Worlds

We first define a sequence of possible worlds X_i such that X_i is the smallest world useful for finding the meanings of *all* closed lterms $[M, B]$: comm where index$(B) \leq i$. We let $X_n = \mathbb{N}^n = \{\text{states of length } n\}$. Alternatively, if we had viewed X_i as the smallest world appropriate for finding the meanings of all configurations whose state part has exactly length i, we would have reached the same definition.

We will let e^i_k denote the expressor for world X_i which reads from the k-th location. So

$$\mathrm{e}^i_k = \lambda(f, Q) : X_i \rightarrow Y.\lambda y \in Y.\,\mathrm{proj}(f(y), k).$$

To see that this works right in a simple case, we observe: $\mathrm{e}^i_k(\mathrm{id}_{X_i})(x) = \mathrm{proj}(x, k)$.

Similarly, we define a^i_k to be the acceptor for world X_i which writes to the k-th location:

$$\mathrm{a}^i_k = \lambda(f, Q) : X_i \rightarrow Y.\ \lambda e \in [\![\mathsf{exp}]\!]Y.\ \lambda(g, R) : Y \rightarrow Z.\ \lambda z \in Z.\ z'$$

where z' is the unique element of Z such that $\mathrm{update}(f(gz), k, e(g, R)z) = f(gz')$ if it exists. It is possible that no such z' exists. If one exists, however, it is guaranteed to be unique by the injectivity aspect of $\mathbf{W}$ morphisms. As a sanity check, we see that:

$$\mathrm{a}_k^i(\mathrm{id}_{X_i})(e)\,\mathrm{id}_{X_i}(x) = \mathrm{update}(x, k, e(\mathrm{id}_{X_i})x).$$

The following Lemma shows how a_k^k lines up with a_{new} and how e_k^k lines up with e_{new}. It is easily justified from the definitions.

Lemma 14 *Let $k \geq 1$ then $\mathrm{a}_k^k = a_{\mathrm{new}}(X_{k-1})$, and $\mathrm{e}_k^k = e_{\mathrm{new}}(X_{k-1})$*

The final important consistency property for a_k^i and e_k^i (which also comes by examining the definitions) is given by the following Lemma.

Lemma 15 *Let $1 \leq k < i$. Let $(f, Q) = \overbrace{\times\mathbb{N} \times \cdots \times \mathbb{N}}^{i-k}$ then $\mathrm{a}_k^i = [\![\mathsf{acc}]\!](f, Q)\,\mathrm{a}_k^k$ and similarly $\mathrm{e}_k^i = [\![\mathsf{acc}]\!](f, Q)\,\mathrm{e}_k^k$.*

3.5.2 Meanings for Bindings: Environments

There is one technical difficulty in defining the denotations of bindings. The meaning of a binding, B, cannot be a global element of Env (*viz.* a natural transformation from $\mathbf{1}$ to Env). To see this, we consider the binding B which maps κ (an identifier of type exp) to 1. Thus $[\![B]\!]$ should be such that: $[\![B]\!]X_2\kappa = \mathrm{e}_1^2$. Now consider the endomorphism $f, Q : X_2 \xrightarrow{\mathbf{W}} X_2$, where $f(\langle x, y\rangle) = \langle y, x\rangle$, and $Q = \Delta_{X_2}$—the equality relation on X_2. If $[\![B]\!]$ were natural, then we would have to have

$$[\![\mathsf{exp}]\!](f, Q)([\![B]\!]X_2\kappa) = \mathrm{e}_1^2.$$

But, as the left hand side is equal to e_2^2, we have a contradiction.

On the other hand, the meaning of a binding cannot be world-independent. We observe that the only time in which we use the meaning of a binding is to find the meaning of a configuration. Furthermore, the meanings for configurations are defined in worlds of a very special form (an X_i for some i). It is therefore not surprising that we will only be needing the meaning of an environment in one of the X_i's. Moreover, for any of these worlds it is quite clear what the "right" meaning of the above B is at κ.. The right meaning is e_1^i. We therefore take the meaning of B to be a function from $\mathbb{N}$ to $\bigcup_i \mathrm{Env}(X_i)$—actually the domain of $[\![B]\!]$ is $\{n \mid n \geq \mathrm{index}(B)\}$ rather than $\mathbb{N}$. $[\![B]\!]$ will have the property that $[\![B]\!]i \in \mathrm{Env}(X_i)$. Specifically, we define $[\![B]\!]i$, for all $i \geq \mathrm{index}(B)$ as follows:

$$[\![B]\!]i(\alpha^\theta) = \begin{cases} \mathrm{e}_j^i & \text{if } B(\alpha^\theta) = j \text{ and } \theta = \mathsf{exp}, \\ \mathrm{a}_j^i & \text{if } B(\alpha^\theta) = j \text{ and } \theta = \mathsf{acc}, \\ \bot_{[\![\theta]\!]X_i} & \text{otherwise.} \end{cases}$$

We conclude this Section by mentioning the following important Lemma about $[\![B]\!]$ which follows immediately from Lemma 15.

Lemma 16 *Suppose* $i \geq \text{index}(B)$, *then*

$$[\![B]\!](i+m) = \text{Env}(\overbrace{\times\mathbb{N} \times \cdots \times \mathbb{N}}^{m})([\![B]\!]i).$$

3.5.3 Interpreting Configurations

We have a natural transformation from worlds to cpo's to interpret terms. We have a way to choose the world appropriate to any given state. We are able to think of a state as a semantic entity. We have a function which takes a binding and state length (world) to give an environment. How do we put these together to interpret a configuration $\langle M, B, \sigma\rangle$ with $\text{length}(\sigma) = i$ by:

$$\begin{array}{ll} [\![M]\!]X_i([\![B]\!]i) & \text{for } M \text{ of higher type,} \\ [\![M]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma & \text{for } M \text{ of base type.} \end{array}$$

The following Lemma provides an important consistency result between our interpretation of configurations and the way the model interprets New blocks.

Lemma 17 *Let* $\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E \text{ in } P, B, \sigma\rangle$ *be a program configuration* $l = 1 + \text{length}(\sigma)$, *and* $B' = B[l/\iota, l/\kappa]$. *Furthermore, suppose* $[\![\langle E, B, \sigma\rangle]\!] = n \neq \bot$. *Then the following always holds:*

$$[\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E \text{ in } P, B, \sigma\rangle]\!] = S(\times\mathbb{N})([\![\langle P, B', \sigma{\bullet}n\rangle]\!])$$

Proof:

$$\begin{array}{lll} [\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow n \text{ in } P, B, \sigma\rangle]\!] & & \\ \quad = \quad [\![\mathsf{New}\ \iota,\ \kappa \leftarrow n \text{ in } P]\!](X_{l-1})([\![B]\!](l-1))(\text{id}_{X_{l-1}})(\sigma) & & (\text{def of } [\![\langle\cdot,\cdot,\cdot\rangle]\!]) \\ \quad = \quad S(\times\mathbb{N})\Big([\![P]\!](X\times\mathbb{N})(u')(\text{id}_{X_{l-1}}\times\mathbb{N})(\sigma{\bullet}n)\Big) & & (\text{def of } [\![\mathsf{New}\cdots]\!]) \\ \quad = \quad S(\times\mathbb{N})\Big([\![P]\!](X\times\mathbb{N})([\![B']\!]l)(\text{id}_{X_l})(\sigma{\bullet}n)\Big) & & (\text{see below}) \\ \quad = \quad S(\times\mathbb{N})\Big([\![\langle P, B', \sigma{\bullet}n\rangle]\!]\Big) & & (\text{def of } [\![\langle\cdot,\cdot,\cdot\rangle]\!]) \end{array}$$

We have $\text{id}_{X_{l-1}} \times\mathbb{N} = \text{id}_{X_l}$ simply by virtue of the definitions of $X_l = X_l \times \mathbb{N}$ and the behavior of $\times\mathbb{N}$ on **W**-morphisms. For $[\![B']\!]l = u'$, we first note that u' is defined to be

$$\Big(\text{Env}(\times\mathbb{N})([\![B]\!](l-1))\Big)[a_{new}(X_{l-1})/\iota, e_{new}(X_{l-1})/\kappa].$$

By definition of B', $[\![B']\!]l = ([\![B]\!]l)[\text{a}_l^l/\iota, \text{e}_l^l/\kappa]$. So, by Lemma 16 we have

$$[\![B']\!]l = \Big(\text{Env}(\times\mathbb{N})([\![B]\!](l-1))\Big)[\text{a}_l^l/\iota, \text{e}_l^l/\kappa]$$

Lemma 14 gives $\text{a}_l^l = a_{\text{new}}(X_{l-1})$ and $\text{e}_l^l = e_{\text{new}}(X_{l-1})$, from which we can conclude $[\![B']\!]l = u'$ as required. ■

Chapter 4

Adequacy

In this thesis, we have formally defined EoA, an ALGOL-like language which is a variant of Reynolds's essence of ALGOL. We have given two different definitions of the semantics for EoA. In Section 2.3 we gave an operational semantics for EoA via a set of Plotkin-style SOS rules. In Section 3.4 we gave a denotational semantics for EoA using the category of functors from state shapes to bottomless cpos. Each semantics carries with it a natural notion of equality between terms in EoA. It should be relatively clear that our operational semantics for EoA is "correct" (in the sense that it matches how we expect EoA code to behave). It is not nearly so clear that the denotational semantics for EoA is "correct." Thus to show the "correctness" of our denotational model we need to tie its behavior to the behavior of our operational model in a mathematically rigorous way. Ideally, we would like for this connection to be that equality in our denotational semantics coincides with equality in our operational semantics (observational congruence). When we have this correspondence exactly, we call the denotational model **fully abstract** with respect to the operational model. Unfortunately the semantics of variable allocation (for either block structured allocation or dynamic allocation) has resisted several attempts for fully abstract models. Meyer and Sieber [9] provide a good summary of this difficulty. Although, as we discuss in Chapter 5, some progress has been made towards full abstraction, it looks like we are still quite far away. For example, the denotational model considered here is not fully abstract for EoA [13].

A denotational model which is **adequate**, but not fully abstract, can still be very useful for deducing observational congruences and other properties of code. Any equalities between terms which we are able to prove using such an adequate denotational model are also observational congruences between terms. Thus, if we establish that our denotational model is adequate, then at least we can say that any equalities which we prove denotationally are "correct." Unfortunately, since the denotational model is not fully abstract, it will not be possible to prove *all* observational congruences denotationally.

Section 2.5 contains the precise definition of observational congruence ($\equiv_{obs}$) for EoA. The goal of this Chapter is to prove the following theorem

Theorem 18 (Adequacy) *The Tennent model defined in Section 3.4 is* adequate *for EoA. That is, for all EoA terms M, N,*

$$[\![M]\!] = [\![N]\!] \quad \textit{implies} \quad M \equiv_{obs} N$$

Since the Tennent model is compositional, we need only prove: *for all completely closed EoA commands* P_1, P_2, $[\![P_1]\!] = [\![P_2]\!]$ *implies* $P_1 \equiv_{obs} P_2$.

We will prove this theorem in two steps. First we will show that the evaluation relation preserves denotation (soundness), that is, if $\gamma \in \mathcal{C} \cup \mathcal{L}$ and $\gamma \rightarrow_{\mathcal{L}} \gamma'$ then $[\![\gamma]\!] = [\![\gamma']\!]$. Then we will prove that (at least for programs) denotation determines the final result of evaluation. That is, if $[\![\langle P, B, \sigma\rangle]\!] = \sigma'$ then $\langle P, B, \sigma\rangle \rightarrow\!\!\!\!\rightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle$. This result combined with the fact that $\rightarrow_{\mathcal{L}}$ is the graph of a function gives the result that *meaning determines observation*, the key property of adequacy. It is then a simple task to prove the precise statement of Theorem 18.

4.1 Soundness

Technically we only need the result of soundness for program configurations (configurations whose term part is a program), however the proof requires a somewhat more general induction hypothesis which is captured by the following Theorem.

Theorem 19 (Soundness) *Let* $\gamma \in \mathcal{C} \cup \mathcal{L}$. *Suppose* $\gamma \rightarrow_{\mathcal{L}} \gamma'$ *then* $[\![\gamma]\!] = [\![\gamma']\!]$.

Proof: The proof divides into two separate cases, depending on whether $\gamma \in \mathcal{C}$ or $\gamma \in \mathcal{L}$. The case of $\gamma \in \mathcal{L}$ is a simple induction on the derivation of $\gamma \rightarrow_{\mathcal{L}} \gamma'$, (see Figure 2.3 on page 17). The subcases of the rules (beta) and (eval-operator) fall right out by the standard ccc properties of the model. The subcase of (rec-unwind) is slightly more complicated than usual, but still easy.

The case of $\gamma \equiv \langle Q, B, \sigma\rangle \in \mathcal{C}$ is also a simple induction on the derivation of $\gamma \rightarrow_{\mathcal{L}} \gamma'$ (see Figures 2.4 and 2.5 on page 18, and also the rule (interaction)). The subcase of (interaction) comes directly from the preceding case for $\gamma \in \mathcal{L}$. We will show the case of the rule (New-eval). The other cases are uninteresting.

Recall the rule (New-eval):

$$\frac{\langle P, B', \sigma.n\rangle \rightarrow_{\mathcal{L}} \langle P', B', \sigma'.n'\rangle}{\langle \mathsf{New}\ \iota,\ \kappa \leftarrow n\ \mathsf{in}\ P, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \mathsf{New}\ \iota,\ \kappa \leftarrow n'\ \mathsf{in}\ P', B, \sigma'\rangle}$$

where $l = 1 + \mathrm{length}(\sigma)$ *and* $B' = B[l/\iota, l/\kappa]$. So suppose

$$\langle \mathsf{New}\ \iota,\ \kappa \leftarrow n\ \mathsf{in}\ P, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \mathsf{New}\ \iota,\ \kappa \leftarrow n'\ \mathsf{in}\ P', B, \sigma'\rangle,$$

because $\langle P, B', \sigma.n\rangle \rightarrow_{\mathcal{L}} \langle P', B', \sigma'.n'\rangle$. Then,

$$\begin{array}{lll} [\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow n \ \mathsf{in}\ P, B, \sigma\rangle]\!] & & \\ = & S(\times\mathbb{N})([\![\langle P, B', \sigma.n\rangle]\!]) & \text{(Lemma 17)} \\ = & S(\times\mathbb{N})([\![\langle P', B', \sigma'.n'\rangle]\!]) & \text{(induction)} \\ = & [\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow n' \ \mathsf{in}\ P', B, \sigma'\rangle]\!] & \text{(Lemma 17)} \end{array}$$

exactly as required. ■

4.2 The Computability Relation

The following Theorem expresses the other half of the adequacy proof. To prove this Theorem we use the method of Logical Relations, following very much along Plotkin's lines for PCF [18]. The proof we give uses a syntactic unary logical relation on lterms. The modern trend in adequacy proofs (see [10]) has been to use a inclusive binary logical relation between elements of the denotational model and syntax. We have also proven the Theorem using this modern method. There are various tradeoffs between the two methods. The unary relation requires substantial syntactic work to handle the constants Y_θ. On the other hand, parameterizing the model by worlds and extending terms to lterms complicates the relationship between the model and syntax. We are thus unable to come up with a single binary relation between the denotational model and syntax. Instead we require a distinct relation for every possible binding.

Theorem 20 *For all Program configurations* $\langle P, B, \sigma\rangle$,

$$\textit{if}\quad [\![\langle P, B, \sigma\rangle]\!] = \sigma' \quad \textit{then}\quad \langle P, B, \sigma\rangle \rightarrow\!\!\!\!\rightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle.$$

We prove this by induction on the structure of terms. The induction hypothesis is stated in the form of a Logical Relation. More precisely, by induction on types we define a series of predicates on lterms which we will call comp_θ. Our goal will then be to prove by induction on the structure of terms that all lterms of type θ have the property comp_θ.

The task remaining is to find an appropriate definition of comp_θ (for all types θ) and then to prove that all lterms $[M, B]:\theta$ has the property comp_θ. If $[M, B]:\theta$ has the property comp_θ then we call $[M, B]$ **computable**. We define the relation comp_θ on lterms $[M, B]$ by induction on types as follows:

1. We have two subcases for closed lterms $[Q, B]$ of base type:

   (a) $[Q, B]:\mathsf{comm}$. Let $i = \mathrm{index}(B)$. $[Q, B]$ has property $\mathrm{comp}_{\mathsf{comm}}$ iff $\forall \sigma \in X_i$:

   $$[\![\langle Q, B, \sigma\rangle]\!] = \sigma' \quad \text{implies} \quad \langle Q, B, \sigma\rangle \rightarrow\!\!\!\!\rightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle$$

(b) $[Q, B]$:exp. Let $i = \text{index}(B)$. $[Q, B]$ has property $\text{comp}_{\mathsf{exp}}$ iff $\forall \sigma \in X_i$:

$$[\![\langle Q, B, \sigma\rangle]\!]\sigma = n \neq \perp \quad \text{implies} \quad \langle Q, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle n, B, \sigma\rangle$$

2. $[M, B] : \theta \to \tau$ is closed. $[M, B]$ has property $\text{comp}_{\theta\to\tau}$ iff
   For any closed lterm $[N, B]:\theta$, the closed lterm $[(M\ N), B]$ has property comp_{τ}.

3. $[M, B] : \theta$ is open. Let $\text{Dom}(\pi) - \text{Dom}\, B = \{\alpha_1^{\theta_1}, \ldots, \alpha_k^{\theta_k}\}$. $[M, B]$ has property comp_{θ} iff
   For all sequences of closed lterms $[N_1, B] : \theta_1, \ldots [N_k, B] : \theta_k$ having properties $\text{comp}_{\theta_1}, \ldots, \text{comp}_{\theta_k}$ respectively, the closed lterm $\left[M[N_1/\alpha_1^{\theta_1}] \cdots [N_k/\alpha_k^{\theta_k}], B\right]$ has property comp_{θ}.

   In other words, $[M, B]$ has property comp_{θ} if all closed instantiations of $[M, B]$ by computable closed lterms is computable.

Of course, in order for this to be useful, we make sure in selecting the definition of comp, that if the closed lterm $[P, B]$ has property $\text{comp}_{\mathsf{comm}}$ then, for all states σ, σ' (with length $\geq \text{index}(B)$)

$$[\![\langle P, B, \sigma\rangle]\!] = \sigma' \quad \text{implies} \quad \langle P, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle.$$

This requires one extra step from the definition of $\text{comp}_{\mathsf{comm}}$ in the event that $\text{length}(\sigma) > \text{index}(B)$. This extra step can be done either purely operationally or via a combination of denotational and operational reasoning. We choose the latter. Specifically, let κ be a fresh exp identifier, $i = \text{length}(\sigma)$ and $B' = B[i/\kappa]$. Since $\kappa \notin FV(P)$,

$$[\![\langle P, B, \sigma\rangle]\!] = [\![\langle P, B', \sigma\rangle]\!] = \sigma'.$$

If we could also show that the closed lterm $[P, B']$ has the property $\text{comp}_{\mathsf{comm}}$, then $\langle P, B', \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \sigma'$. Finally, by Property 4 of $\to_{\mathcal{L}}$ which was

Let $\alpha \notin \text{FV}(Q)$ and $i \leq \text{length}(\sigma)$, then

$$\langle Q, B, \sigma\rangle \to_{\mathcal{L}} \langle Q', B, \sigma'\rangle \quad \textit{iff} \quad \langle Q, B[i/\alpha], \sigma\rangle \to_{\mathcal{L}} \langle Q', B[i/\alpha], \sigma'\rangle.$$

We can relate the behavior of $\langle P, B', \sigma\rangle$ back to the behavior of $\langle P, B, \sigma\rangle$, getting

$$\langle P, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \sigma'$$

as required.

4.2.1 Proving All Lterms Computable

We prove by induction on the structure of terms, that all lterms are computable. We now carefully present the most interesting cases in the proof (although we defer the case for Y_θ to the next section). The base cases in a proof by structural induction are identifiers and constants. The inductive cases are applications, λ-abstractions, and New-variable declarations.

Identifiers come in three flavors. They are: an identifier appearing free in a lterm, an exp identifier appearing in the domain of the binding, and an acc identifier appearing in the domain of the binding. Consider an arbitrary such lterm $[\alpha^\theta, B]$. We have:

$[\alpha^\theta \notin \mathrm{Dom}(B)]$ Any instantiation of the lterm $[\alpha^\theta, B]$ by a closed computable lterm $[N, B]:\theta$ is $[N, B]$ which has property comp_θ (by definition).

$[\theta = \mathsf{exp}$ **and** $B(\alpha^{\mathsf{exp}}) = i]$ Let $j = \mathrm{index}(B)$ and $\sigma \in X_j$. In this case we have $[\![\langle \alpha^{\mathsf{exp}}, B, \sigma\rangle]\!] = \mathrm{proj}(\sigma, i)$ because

$$\begin{array}{rcll} [\![\langle \alpha^{\mathsf{exp}}, B, \sigma\rangle]\!] & = & [\![\alpha^{\mathsf{exp}}]\!] X_j([\![B]\!]j)\,\mathrm{id}_{X_j}\,\sigma & (\text{def of } [\![\langle\cdot,\cdot,\cdot\rangle]\!]) \\ & = & (\mathrm{e}_i^j\,\mathrm{id}_{X_j})\sigma & (\text{def of } [\![B]\!]) \\ & = & \sigma_i & (\text{def of } \mathrm{e}_i^j). \end{array}$$

Operationally, the rule (variable-read) gives the desired result:

$$\langle \alpha^{\mathsf{exp}}, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \mathrm{proj}(\sigma, i), B, \sigma\rangle.$$

$[\theta = \mathsf{acc}$ **and** $B(\alpha^{\mathsf{acc}}) = i]$ Remember that acc is an abbreviation for exp $\rightarrow$ comm. We need to show for all closed lterms $[E, B]$ of type exp, that the lterm $[(\alpha^{\mathsf{acc}}\, E), B]$ has property $\mathrm{comp}_{\mathsf{comm}}$. Let $j = \mathrm{index}(B)$ and $\sigma \in X_j$. We must show that:

$$[\![\langle(\alpha^{\mathsf{acc}}\, E), B, \sigma\rangle]\!] = \sigma' \text{ implies } \langle(\alpha^{\mathsf{acc}}\, E), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \sigma'\rangle$$

As

$$\begin{array}{rcll} [\![\langle \alpha^{\mathsf{acc}}, B, \sigma\rangle]\!] & = & [\![(\alpha^{\mathsf{acc}}\, E)]\!] X_j([\![B]\!]j)\,\mathrm{id}_{X_j}\,\sigma & (\text{def of } [\![\langle\cdot,\cdot,\cdot\rangle]\!]) \\ [\![(\alpha^{\mathsf{acc}}\, E)]\!] X_j([\![B]\!]j) & = & \mathrm{a}_i^j\,\mathrm{id}_{X_j}\,([\![E]\!] X_j([\![B]\!]j)) & \\ & & \quad (\text{def of application and } [\![B]\!]), & \end{array}$$

we have $\sigma' = \mathrm{a}_i^j\,\mathrm{id}_{X_j}\,([\![E]\!] X_j([\![B]\!]j))\,\mathrm{id}_{X_j}\,\sigma$. The definition of a_i^j implies that there is an $n \in \mathbb{N}$ such that $[\![E]\!] X_j([\![B]\!]j)\,\mathrm{id}_{X_j}\,\sigma = n$ and $\sigma' = \mathrm{update}(\sigma, i, n)$. We assumed that $[E, B]$ was a computable closed lterm with property $\mathrm{comp}_{\mathsf{exp}}$, so $\langle E, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle n, B, \sigma\rangle$. Finally by a simple induction on the length of the rewriting sequence (using the rule (assign-eval-arg)), followed by an application of the rule (variable-write) we have

$$\langle(\alpha^{\mathsf{acc}}\, E), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle \alpha^{\mathsf{acc}}\, n, B, \sigma\rangle \rightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \mathrm{update}(\sigma, i, n)\rangle$$

giving $\langle(\alpha^{\mathsf{acc}}\, E), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, B, \mathrm{update}(\sigma, i, n)\rangle$.

Constants other than Y_θ are also relatively straightforward to handle. The following examples represent all of the interesting issues.

$[n, B]$ Let $i = \text{index}(B)$ and $\sigma \in X_i$. Since $[\![\langle n, B, \sigma\rangle]\!] = [\![n]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma = n$, and $\langle n, B, \sigma\rangle \xrightarrow{0}_{\mathcal{L}} \langle n, B, \sigma\rangle$ we are done.

$[\mathsf{seq}, B]$ Let $i = \text{index}(B)$ and $\sigma \in X_i$. We need to show for all closed computable lterms $[P_1, B] : \mathsf{comm}$ and $[P_2, B] : \mathsf{comm}$ that the closed lterm $[(\mathsf{seq}\ P_1\ P_2), B]$ has property $\text{comp}_{\mathsf{comm}}$. Specifically,

$$[\![\langle(\mathsf{seq}\ P_1\ P_2), B, \sigma\rangle]\!] = \sigma' \quad \text{implies} \quad \langle(\mathsf{seq}\ P_1\ P_2), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle\mathsf{skip}, B, \sigma'\rangle.$$

Suppose $[\![(\mathsf{seq}\ P_1\ P_2)]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma = \sigma'$. From the definition of $[\![\mathsf{seq}]\!]$ we can conclude that there is a $\sigma'' \in X_i$ such that $[\![P_1]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma = \sigma''$ and $[\![P_2]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma'' = \sigma'$. Our original assumption was that both $[P_1, B]$ and $[P_2, B]$ were closed lterms with property $\text{comp}_{\mathsf{comm}}$. Thus, $\langle P_1, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle\mathsf{skip}, B, \sigma''\rangle$ and $\langle P_2, B, \sigma''\rangle \twoheadrightarrow_{\mathcal{L}} \langle\mathsf{skip}, B, \sigma'\rangle$ An easy induction on the length of the first rewriting sequence (using the rule (seq-eval-arg1)) followed by an application of the rule (seq-discharge) then gives

$$\langle(\mathsf{seq}\ P_1\ P_2), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle(\mathsf{seq}\ \mathsf{skip}\ P_2), B, \sigma''\rangle \rightarrow_{\mathcal{L}} \langle P_2, B, \sigma''\rangle \twoheadrightarrow_{\mathcal{L}} \langle\mathsf{skip}, B, \sigma\rangle,$$

giving $\langle(\mathsf{seq}\ P_1\ P_2), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle\mathsf{skip}, B, \sigma'\rangle$.

$[\mathsf{plus}, B]$ Let $i = \text{index}(B)$ and $\sigma \in X_i$. We need to show that for all closed computable lterms $[E_1, B] : \mathsf{exp}$ and $[E_2, B] : \mathsf{exp}$ that the closed lterm $(\mathsf{plus}\ E_1\ E_2)$ has property $\text{comp}_{\mathsf{exp}}$. Specifically, $[\![\langle(\mathsf{plus}\ E_1\ E_2), B, \sigma\rangle]\!] = n \neq \perp$ implies $\langle(\mathsf{plus}\ E_1\ E_2), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle n, B, \sigma\rangle$. So, suppose

$$[\![(\mathsf{plus}\ E_1\ E_2)]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma = n.$$

By the definition of $[\![\mathsf{plus}]\!]$, we know that there must be $n_1, n_2 \in \mathbb{N}$ such that:

$$\begin{aligned} [\![E_1]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma &= n_1 \\ [\![E_2]\!]X_i([\![B]\!]i)\,\text{id}_{X_i}\,\sigma &= n_2 \\ n_1 + n_2 &= n \end{aligned}$$

As $[E_1, B]$ and $[E_2, B]$ are computable, we have $\langle E_1, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle n_1, B, \sigma\rangle$ and $\langle E_2, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle n_2, B, \sigma\rangle$ Another two simple inductions on the length of these rewriting sequences (one using the rule (plus-eval-arg1) and the other using (plus-eval-arg2)) followed by an application of the rule (plus-do) give

$$\langle(\mathsf{plus}\ E_1\ E_2), B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle n, B, \sigma\rangle.$$

Applications fall out easily from comp_θ being a logical relation. The case of $[(M\ N), B]$ closed is immediate. If $[(M\ N), B]$ is open, Let $M : \theta \to \tau$ and $N : \theta$ and let $[L, B]$ be a closed instantiation of $[(M\ N), B]$ by terms $N_1, \ldots, N_k$ such that each $[N_j, B]$ (for $1 \leq j \leq k$) is computable. L must have the form $(M'\ N')$ where M' and N' are instantiations of M and N respectively such that both $[M', B]$ and $[N', B]$ are closed lterms. By induction $[M, B]$ and $[N, B]$ were computable, and so $[M', B]$ has the property $\mathrm{comp}_{\theta\to\tau}$ and $[N', B]$ has the property comp_θ. By definition of $\mathrm{comp}_{\theta\to\tau}$, $[(M'\ N'), B]$ has property comp_τ exactly as required.

λ-abstractions, $[(\lambda\alpha^\theta.\, M), B] : \theta \to \tau$. Let $[L, B]$ be an arbitrary closed instantiation of $[(\lambda\alpha^\theta.M), B]$ by computable closed lterms (same general methodology as applications). In order to show that $[L, B]$ has the property $\mathrm{comp}_{\theta\to\tau}$ we observe that $L \equiv (\lambda\alpha^\theta.M')$ where M' is an instantiation of all of the free identifiers of $[M, B]$ other than α^θ. The type $\theta \to \tau$ can be uniquely rewritten to be of the form $\theta_0 \to \theta_1 \to \ldots \theta_k \to \beta$ for some $k \geq 0$ and $\beta \in \{\mathsf{exp}, \mathsf{comm}\}$. To show that $[L, B]$ is computable, it suffices to show, for arbitrary computable closed lterms $[N_0, B] : \theta_0, \ldots, [N_k, B] : \theta_k$, that the ground term $[(L\ N_0\ N_1\ \ldots\ N_k), B]$ has the property comp_β. For concreteness, we show the result for $\beta = \mathsf{exp}$. Let $i = \mathrm{index}(B)$ and $\sigma \in X_i$. Suppose $[\![L\ N_0 \cdots N_k]\!] X_i([\![B]\!]i)\,\mathrm{id}_{X_i}\,\sigma = n \neq \bot$, then by Soundness

$$[\![(M'[N_0/\alpha^\theta])\ N_1 \cdots N_k]\!] X_i([\![B]\!]i)\,\mathrm{id}_{X_i}\,\sigma = [\![L\ N_0 \cdots N_k]\!] X_i([\![B]\!]i)\,\mathrm{id}_{X_i}\,\sigma = n.$$

But $[M'[N_0/\alpha^\theta], B]$ is a computable closed lterm, therefore the closed lterm $[((M'[N_0/\alpha^\theta])\ N_1 \cdots N_k), B]$ is also computable. We now have:

$$\langle L\ N_0 \cdots N_k, B, \sigma\rangle \to_{\mathcal{L}} \big\langle (M'[N_0/\alpha^\theta])\ N_1 \cdots N_k, B, \sigma\big\rangle \twoheadrightarrow_{\mathcal{L}} \langle n, B, \sigma\rangle.$$

New-declarations, $[(\mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P), B]$. Without loss of generality, suppose this lterm is closed. Let $i = \mathrm{index}(B)$ and $\sigma \in X_i$. By the identifier convention we may assume that $\iota \notin \mathrm{Dom}(B)$ and $\kappa \notin \mathrm{Dom}(B)$. We must show that

$$[\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P, B, \sigma\rangle]\!] = \sigma'$$
$$\text{implies}$$
$$\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, \sigma'\rangle.$$

We make the following abbreviations:

$$\begin{aligned} B' &= B[(i+1)/\iota, (i+1)/\kappa] \\ n &= [\![E]\!] X_i([\![B]\!]i)\,\mathrm{id}_{X_i}\,\sigma = [\![\langle E, B, \sigma\rangle]\!] \end{aligned}$$

Suppose $[\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P, B, \sigma\rangle]\!] = \sigma'$. The fact that the New-block is defined tells us that $n \neq \bot$. By induction $[E, B]$ has property $\mathrm{comp}_{\mathsf{exp}}$. Since $[E, B]$ is a closed lterm, we have $\langle E, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle n, B, \sigma\rangle$. Then

$$\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E\ \mathsf{in}\ P, B, \sigma\rangle \to_{\mathcal{L}} \langle \mathsf{New}\ \iota,\ \kappa \leftarrow n\ \mathsf{in}\ P, B, \sigma\rangle.$$

By soundness, $[\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E \,\mathsf{in}\, P, B, \sigma\rangle]\!] = [\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow n \,\mathsf{in}\, P, B, \sigma\rangle]\!] = \sigma'$. By Lemma 17,

$$[\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow n \,\mathsf{in}\, P, B, \sigma\rangle]\!] = S(\times\mathbb{N})[\![\langle P, B', \sigma.n\rangle]\!] = \sigma'.$$

By definition of $S(\times\mathbb{N})$, there must exist an n' such that

$$[\![\langle P, B', \sigma.n\rangle]\!] = \sigma'.n'.$$

But $[P, B']$ is a closed lterm with property $\mathrm{comp}_{\mathsf{comm}}$. Note that we were not doing an induction on the structure of lterms, but an induction on the structure of (ordinary) terms. As, P is subterm of $(\mathsf{New}\ \iota,\ \kappa \leftarrow E \,\mathsf{in}\, P)$, this is a sound application of our induction hypothesis. Thus using the computability of $[P, B']$, we have that:

$$\langle P, B', \sigma.n\rangle \twoheadrightarrow_{\mathcal{L}} \langle \mathsf{skip}, B', \sigma'.n'\rangle$$

An easy induction on the length of the rewriting sequence (using the rule (New-eval)) followed by an application of the rule (New-discharge) gives

$$\begin{array}{rll} \langle \mathsf{New}\ \iota,\ \kappa \leftarrow E \,\mathsf{in}\, P, B, \sigma\rangle & \twoheadrightarrow_{\mathcal{L}} & \langle \mathsf{New}\ \iota,\ \kappa \leftarrow n \,\mathsf{in}\, P, B, \sigma\rangle \\ & \twoheadrightarrow_{\mathcal{L}} & \langle \mathsf{New}\ \iota,\ \kappa \leftarrow n' \,\mathsf{in}\, \mathsf{skip}, B, \sigma'\rangle \\ & \rightarrow_{\mathcal{L}} & \langle \mathsf{skip}, B, \sigma\rangle \end{array}$$

4.2.2 Handling Y

As with PCF [18], the main difficulty in proving all terms computable is caused by the recursion operators, Y_θ. Before we handle this last case, we need to do some substantial preliminary work.

We make the obvious generalization of Plotkin's work to EoA. We approximate Y_θ by terms $\mathsf{Y}_\theta^{(n)}$, (for $n \geq 0$). To do so we first define terms $\Omega_{\mathsf{exp}} \equiv \mathsf{Y}_{\mathsf{exp}}(\lambda\alpha^{\mathsf{exp}}.\alpha^{\mathsf{exp}})$, $\Omega_{\mathsf{comm}} \equiv \mathsf{Y}_{\mathsf{comm}}(\lambda\alpha^{\mathsf{comm}}.\alpha^{\mathsf{comm}})$, and $\Omega_{\theta\to\tau} = (\lambda\alpha^{\theta}.\Omega_\tau)$. We then define $\mathsf{Y}_\theta^{(n)}$ by induction on n as follows:

$$\mathsf{Y}_\theta^{(0)} \equiv \Omega_{(\theta\to\theta)\to\theta} \text{ and } \mathsf{Y}_\theta^{(n+1)} \equiv (\lambda\alpha^{\theta\to\theta}.(\alpha^{\theta\to\theta}\,(\mathsf{Y}_\theta^{(n)}\,\alpha^{\theta\to\theta})))$$

We then have the following useful combinatorial properties for all $\mathbf{W}$-worlds X, $\mathbf{W}$-morphisms $f : X \to Y$, $u \in \mathrm{Env}(X)$, and $m \in [\![\theta \to \theta]\!]Y$

$$\begin{array}{rcl} [\![\mathsf{Y}_\theta]\!]Xufm & = & \bigsqcup_{n\geq 0} (m\,\mathrm{id}_Y)^n\,(\bot_{[\![\theta]\!]X}), \\ [\![\Omega_\theta]\!]Xu & = & \bot_{[\![\theta]\!]X}, \\ [\![\mathsf{Y}_\theta^{(n)}]\!]Xufm & = & (m\,\mathrm{id}_Y)^n\,(\bot_{[\![\theta]\!]X}), \\ [\![\mathsf{Y}_\theta]\!]Xu & = & \bigsqcup\{[\![\mathsf{Y}_\theta^{(n)}]\!]Xu : n \geq 0\}. \end{array}$$

We define $\preceq$ to be the least relation between terms such that:

- $\Omega_\theta \preceq M:\theta$ and $\mathsf{Y}_\theta^{(n)} \preceq \mathsf{Y}_\theta$ for all θ, $n \geq 0$,
- $M \preceq M$,
- If $M:(\theta \to \tau) \preceq M':(\theta \to \tau)$ and $N:\theta \preceq N':\theta$ then $(\lambda\alpha^{\theta'}.N) \preceq (\lambda\alpha^{\theta'}.N')$, and $(MN) \preceq (M'N')$,
- If $E:\mathsf{exp} \preceq E':\mathsf{exp}$ and $P:\mathsf{comm} \preceq P':\mathsf{comm}$ then

$$(\mathsf{New}\,\iota,\ \kappa \leftarrow E\,\mathsf{in}\,P) \preceq (\mathsf{New}\,\iota,\ \kappa \leftarrow E'\,\mathsf{in}\,P').$$

The following is the appropriate analogue of Plotkin's Lemma 3.2 [18]

Lemma 21 *If $M \preceq N$, $\langle M, B, \sigma\rangle \in \mathcal{C}$ and $\langle M, B, \sigma\rangle \to_{\mathcal{L}} \langle M', B, \sigma'\rangle$ then either $M' \preceq N$ and $\sigma = \sigma'$ or else for some N' and σ', $\langle N, B, \sigma\rangle \to_{\mathcal{L}} \langle N', B, \sigma'\rangle$ and $M' \preceq N'$.*[1]

Proof: In order to prove the above Lemma we must first establish the analogous result for $M \to_{\mathcal{L}} M'$, specifically

> If $M \preceq N$, $M \to_{\mathcal{L}} M'$ then either $M' \preceq N$ or else for some N' $N \to_{\mathcal{L}} N'$ and $M' \preceq N'$.

The proof of the above result for $M \to_{\mathcal{L}} M'$ is a simple induction on the derivation of $M \preceq M'$. The proof of the Lemma itself is then merely a tedious induction on the derivation of $\langle M, B, \sigma\rangle \to_{\mathcal{L}} \langle M', B, \sigma'\rangle$, with the case of (interaction) coming directly from the preceding result. ■

We are now ready to prove that all lterms with term part of the form Y_θ have the property $\mathrm{comp}_{(\theta\to\theta)\to\theta}$. Let $[\mathsf{Y}_\theta, B]$ be an arbitrary such lterm. We start by observing that any type θ is of the form $\tau_1 \to \tau_2 \to \cdots \to \tau_k \to \beta$, where $\beta \in \{\mathsf{exp}, \mathsf{comm}\}$. To show that $[\mathsf{Y}_\theta, B]$ has the property $\mathrm{comp}_{(\theta\to\theta)\to\theta}$, it is enough to show that for arbitrary computable closed lterms $[N_0, B]:\theta \to \theta, [N_1, B]:\tau_1, \ldots, [N_k, B]$, the closed lterm $[(Y_\theta N_0 \cdots N_k), B]$ has property comp_β. The two cases for β are essentially similar; we argue the case of $\beta = \mathsf{exp}$.

Let $i = \mathrm{index}(B)$, and $\sigma \in X_i$. We must show that

$$[\![(\mathsf{Y}_\theta N_0 \cdots N_k)]\!] X_i([\![B]\!]i)\,\mathrm{id}_{X_i}\,\sigma = c \neq \perp$$

implies that

$$\langle (\mathsf{Y}_\theta N_0 \cdots N_k), B, \sigma\rangle \to\!\!\!\!\to_{\mathcal{L}} \langle c, B, \sigma\rangle.$$

So assuming the hypothesis, we have:

$$[\![\mathsf{Y}_\theta]\!] X_i([\![B]\!]i)\,\mathrm{id}_{X_i}\,m_0\,\mathrm{id}_{X_i} \cdots \mathrm{id}_{X_i}\,m_k\,\mathrm{id}_{X_i}\,\sigma = c$$

[1] In order to avoid an endless profusion of "primes" and subscripts, we temporarily abandon our metavariable convention (for M and N) for the statement and proof of this Lemma.

Where $m_j = [\![N_j]\!]X_i([\![B]\!]i)$ (for $0 \leq j \leq k$). We remarked earlier that

$$[\![\mathsf{Y}_\theta]\!]X_i([\![B]\!]i) = \bigsqcup_{n\geq 0} [\![\mathsf{Y}_\theta^{(n)}]\!]X_i([\![B]\!]i).$$

This will give us the following:

$$\begin{aligned}\left(\bigsqcup_{n\geq 0} [\![\mathsf{Y}_\theta^{(n)}]\!]X_i([\![B]\!]i)\right) \mathrm{id}_{X_i}\, m_0\, \mathrm{id}_{X_i} \cdots \mathrm{id}_{X_i}\, m_k\, \mathrm{id}_{X_i}\, \sigma &= c \\ \bigsqcup_{n\geq 0} \{[\![Y_\theta^{(n)}]\!]X_i([\![B]\!]i)\, \mathrm{id}_{X_i}\, m_0\, \mathrm{id}_{X_i} \cdots \mathrm{id}_{X_i}\, m_k\, \mathrm{id}_{X_i}\, \sigma\} &= c\end{aligned}$$

Since $c \in \mathbb{N}$, and $\mathbb{N}$ is flat, then there is a least n such that:

$$\begin{aligned}[\![\mathsf{Y}_\theta^{(n)}]\!]X_i([\![B]\!]i)\, \mathrm{id}_{X_i}\, m_0\, \mathrm{id}_{X_i} \cdots \mathrm{id}_{X_i}\, m_k\, \mathrm{id}_{X_i}\, \sigma &= c \\ [\![\mathsf{Y}_\theta^{(n)} N_0 \cdots N_k]\!]X_i([\![B]\!]i)\sigma &= c\end{aligned}$$

All lterms with term part $Y_\theta^{(n)}$ are computable. A simple induction on n shows why. For the basis we have that $[Y_\theta^{(0)}, B] \equiv [\Omega_{(\theta\to\theta)\to\theta}, B]$ is computable (for any binding B), since $\Omega_{(\theta\to\theta)\to\theta}$ denotes $\perp_{[(\theta\to\theta)\to\theta]}$. For the inductive case, $[\mathsf{Y}_\theta^{(n+1)}, B] \equiv [(\lambda\alpha^{\theta\to\theta}.\, (\alpha^{\theta\to\theta}(\mathsf{Y}_\theta^{(n)}\alpha^{\theta\to\theta}))), B]$, we use the earlier cases in proving that all terms are computable (applications and λ-abstractions) to show that if $[\mathsf{Y}_\theta^{(n)}, B]$ is computable, then so is $[Y_\theta^{(n+1)}, B]$.

Consequently, $(\mathsf{Y}_\theta^{(n)} N_0 \cdots N_k)$ has property $\mathrm{comp}_{\mathbf{exp}}$. Thus,

$$\left\langle \mathsf{Y}_\theta^{(n)} N_0 \cdots N_k, B, \sigma \right\rangle \twoheadrightarrow_{\mathcal{L}} \langle c, B, \sigma\rangle$$

Finally, we apply Lemma 21 and so we can say that:

$$\langle \mathsf{Y}_\theta N_0 \cdots N_k, B, \sigma\rangle \twoheadrightarrow_{\mathcal{L}} \langle c, B, \sigma\rangle$$

and thus conclude that $[\mathsf{Y}_\theta, B]$ is computable.

4.3 A General Adequacy Theorem for Functor Category Models

Until now we have focused our attention to Tennent's model of EoA. We abstract away the details of Tennent's model and decribe those properties of the model upon which the adequacy proof of the preceeding Sections depended. We limit our attention to models of EoA which use a functor category of the form $\mathbf{D}^{\mathbf{A}}$ where $\mathbf{A}$ is now permitted to be any small category. We assume $[\![\cdot]\!]$ is

standard c.c.c. interpretation of EoA in $\mathbf{D}^{\mathbf{A}}$. That is it is a specification of the category, objects for the meanings of base types, and morphisms for the meanings of constants. We assume that $[\![\theta \to \tau]\!]$ is the exponent object $[\![\theta]\!] \Rightarrow [\![\tau]\!]$. We assume that meanings for applications and λ-abstractions are found in the standard ways.

Our conditions will provide constraints on what $[\![\mathsf{exp}]\!]$ and $[\![\mathsf{comm}]\!]$ must look like, and how the constants must fit together. It will also place some constraints between the interpretation of New blocks and the meanings for base type. Finally, we must also be able to find appropriate entites in the model to interpret bindings and states so that everything will fit together.

Our first condition is a restriction on $\mathbf{A}$. We must be able to find an appropriate sequence of worlds in $\mathbf{A}$ to model configurations with state part of length 0, and length 1 and length 2, ... To state this precisely we define the category $\mathbf{NAT} = \langle \mathbb{N}, \geq \rangle$. $\mathbf{NAT}$ has $\mathbb{N}$ as objects, and for n greather than or equal to m, it has a morphism $n \geq m : m \to n$. We require that there exist a covariant functor $L : \mathbf{NAT} \to \mathbf{A}$, in order to get off the ground. All of our constraints will be phrased in terms of L. In a generalized adequacy proof, we would use the object $L(i)$ of $\mathbf{A}$ wherever we used the object X_i of $\mathbf{W}$ in the old proof. In place the morphism $\times\mathbb{N}^k : X_i \to X_{i+k}$, we now use the morphism $L(i+k \geq i) : L(i) \to L(i+k)$.

We require that $[\![\mathsf{exp}]\!](L(i))$ "looks like" $\mathbb{N}^i \to \mathbb{N}_\perp$, and that $[\![\mathsf{comm}]\!](L(i))$ looks like $\mathbb{N}^i \to \mathbb{N}_\perp^i$. So we require that $\mathbf{ap}_{\mathsf{exp}}\langle e, \sigma\rangle$, when viewed as a function of σ, is isomorphic to $\mathbb{N}^i \to \mathbb{N}_\perp$ (wlog we assume it is equal). Similarly we require that $\mathbf{ap}_{\mathsf{comm}}\langle c, \sigma\rangle$, when viewed as a function of σ, is isomorphic to $\mathbb{N}^i \to \mathbb{N}_\perp^i$ (wlog we assume it is equal). We should really index $\mathbf{ap}_{\mathsf{exp}}$ and $\mathbf{ap}_{\mathsf{comm}}$ by the world $L(i)$ in which the first argument lives, but this should always be evident from context. We also require that $[\![\mathsf{exp}]\!](L(m \geq n))$ and $[\![\mathsf{comm}]\!](L(m \geq n))$ behave sensibly. Specifically, we require:

$$\mathbf{ap}_{\mathsf{exp}}\langle [\![\mathsf{exp}]\!](L(m \geq n))(e), \langle v_1, \ldots v_m\rangle\rangle = \mathbf{ap}_{\mathsf{exp}}\langle e, \langle v_1, \ldots v_n\rangle\rangle,$$

and

$$\mathbf{ap}_{\mathsf{comm}}\langle [\![\mathsf{exp}]\!](L(m \geq n))(c), \langle v_1, \ldots v_m\rangle\rangle = \begin{cases} \langle v_1', \ldots v_n', v_{n+1}, \ldots v_m\rangle & \text{if } \mathbf{ap}_{\mathsf{exp}}\langle c, \langle v_1, \ldots v_n\rangle\rangle = \langle v_1', \ldots, v_n'\rangle \\ \perp & \text{otherwise.} \end{cases}$$

We require that all of the constants other than Y_θ (n, skip, $\mathsf{IFexp}_{\mathsf{exp}}$, seq, $\mathsf{IFexp}_{\mathsf{comm}}$, succ, and plus) satisfy the obvious equations. For Y_θ, we require that

$$[\![\mathsf{Y}_\theta]\!](L(i))(f) = \bigsqcup_{n \geq 0} (f\ \mathrm{id}_{L(i)})^n(\perp_{[\![\theta]\!](L(i))}).$$

Finally, we must be able to find "locations" in the model which fit together properly with each other and with L. For each i and $k \leq i$ we need to find

an acceptor $\mathrm{a}_k^i \in [\![\mathsf{acc}]\!](L(i))$ which corresponds to the acceptor for the k-th component of the state. Similarly we need an expressor $\mathrm{e}_k^i \in [\![\mathsf{exp}]\!](L(i))$. These must fit together as they did for Tennent's model, specifically,

$$\begin{aligned} \mathrm{a}_k^i &= [\![\mathsf{acc}]\!](L(i \geq k))\,\mathrm{a}_k^k, \\ \mathrm{e}_k^i &= [\![\mathsf{exp}]\!](L(i \geq k))\,\mathrm{e}_k^k. \end{aligned}$$

Finally, not only must they fit together properly, but they must behave like the right locations. The following condition is sufficient:

$$\begin{aligned} \mathbf{ap}_{\mathsf{comm}}\langle \mathbf{ap}_{\mathsf{acc}} L(k)\langle \mathrm{a}_k^k, e\rangle, \sigma\rangle &= \begin{cases} \mathrm{update}(\sigma, n, k) & \text{if } \mathbf{ap}_{\mathsf{exp}}\langle e, \sigma\rangle = n \neq \bot, \\ \bot & \text{otherwise} \end{cases} \\ \mathbf{ap}_{\mathsf{exp}}\langle \mathrm{e}_k^k, \sigma\rangle &= \mathrm{proj}(\sigma, k) \end{aligned}$$

We can then define the meanings for bindings as in Section 3.5.2. Finally we define the meaning of a configuration $\langle M, B, \sigma\rangle$ with $\mathrm{length}(\sigma) = i$ as follows:

$$\begin{array}{ll} [\![M]\!](L(i))([\![B]\!]i) & \text{for } M \text{ of higher type,} \\ [\![M]\!](L(i))([\![B]\!]i)\,\mathrm{id}_{X,}\,\sigma & \text{for } M \text{ of base type.} \end{array}$$

Lastly we need the appropriate analogue of Lemma 17, which is: for all program configurations $\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E \text{ in } P, B, \sigma\rangle$, if $[\![\langle E, B, \sigma\rangle]\!] = n \neq \bot$, $l = 1 + \mathrm{length}(\sigma)$, and $B' = B[l/\iota, l/\kappa]$.

$$\begin{aligned} &[\![\langle \mathsf{New}\ \iota,\ \kappa \leftarrow E \text{ in } P, B, \sigma\rangle]\!] \\ &\quad = \begin{cases} \langle v_1, \ldots, v_l\rangle & \text{if } [\![\langle P, B', \sigma{\cdot}n\rangle]\!] = \langle v_1, \ldots, v_l, v_{l+1}\rangle, \\ \bot & \text{otherwise.} \end{cases} \end{aligned}$$

And that's all.

Chapter 5

Comparing Models of Block Structure

Although the field of denotational semantics has made many strides in its history, there is still great difficulty in finding reasonable models of imperative features and variable allocation. The most commonly used model of imperative features is the "marked store" model, which is little more than an encoding of the usual operational semantics—providing little added insight to justify the additional technical background required to understand the denotational semantics. A guiding concern in the design of a denotational semantics is for the semantics to be "fully abstract." A fully abstract semantics would by definition, contain precisely the information about a term that we require in order to understand its behavior in all contexts. In other words we would have "abstracted away" from all of the irrelevant details. As discussed in [9] the marked stores model fails very early on in the quest to full abstraction for block structured local variables. The failure results from an inability to fully understand the behavior of commands.

5.1 Advanced Models of Block Structure

During the 1980's several more promising models of block structure have been presented. Reynolds and Oles developed a model of block structure based on a category of functors from a category of state shapes to a category of bottomless cpos [15–17, 21]. The work of Reynolds and Oles has been simplified in Tennent's model of Reynolds's Specification Logic [25]. Tennent separated the issues of implicit coercions and block structure, which had been intertwined in the model of Reynolds and Oles. Tennent also found a few other technical simplifications of the appropriate category of state shapes which should be used for modeling block structure. In addition, to properly model interference (which helps in

understanding how higher order objects behave) Tennent further modified the category of state shapes and chose an "ad hoc" definition for $[\![\mathsf{comm}]\!]$, rather than taking the functor $S \rightharpoonup S$. Finally, in order to model a principle of "non-interference abstraction," Tennent and O'Hearn [13] adjusted the category of state shapes which Tennent originally used in [25].

Along a different line, Halpern, Meyer and Trakhtenbrot developed a model [5] (now referred to as the HMT store model), which Meyer and Sieber [9] summarize as follows:

> Halpern-Meyer-Trakhtenbrot proposed a formal definition of the *support* of a function from *Stores* to *Stores*. Intuitively the support of a store transformation p is the set of locations which p can read or write. In the HMT store model [5], *Prog* [comm] is taken to be the set of p with *finite* support. To model local variables, the notion of support is extended to the type $Loc \rightarrow Prog$ of block bodies regarded as a function of their free location identifier. The semantical space used to interpret such block body functions is again restricted to be the elements in $Loc_{\perp} \rightarrow_c Prog$ with finite support. Since there are an infinite number of locations, this restriction guarantees that a location can be found which is not in the support of any given block body. Then local storage allocation for a block **begin new** x; *body* **end** is (uniquely) determined by the rule that x be bound to *any* location not in the support of the function denoted by $\lambda x.body$.

Meyer and Sieber developed what they called "The invariant-preserving model," which is now coming to be called the Meyer-Sieber model. This model validates the following reasoning principle:

> Let Q be of type $\mathsf{comm} \rightarrow \mathsf{comm}$, and P of type comm. Let r be a property of states such that $support(r) \cap support(Q) = \emptyset$. If r is an invariant of P, then r is also an invariant of $Q(P)$.

None of the above models is fully abstract for EoA. We will now examine the relative merits of later models—the Tennent model, the O'Hearn-Tennent model and the Meyer-Sieber model. It is not obvious whether the Tennent model and the O'Hearn-Tennent differ in their equational theory. In fact it is not obvious which, if either, has the stronger equational theory.

5.2 The Meyer-Sieber Examples

Our discussion focuses on the Meyer-Sieber Examples [9]. Specifically, we examine the statement of these examples in EoA and address how the Tennent and O'Hearn-Tennent models handle them. All but the last of the Meyer-Sieber Examples hold for the Meyer-Sieber model. All of the examples which can be

stated in EoA hold for the Tennent and O'Hearn-Tennent models. Since Example 6 critically uses the fact that the sample language is based on locations rather than variables, this example cannot be translated into EoA and thus it is not handled by the Tennent and O'Hearn-Tennent models. We will see in Section 5.3, slight perturbations of the problems will lead to either open questions or outright failures.

Example 1 *The block below is replaceable simply by the call P.*

```
begin
    new x;
    P;                                  % P is declared elsewhere
end
```

The translation of this block into EoA would something like:

$$\mathsf{New}\ \iota,\ \kappa \leftarrow 0\ \mathsf{in}\ P$$

Contrary to the claim at the end of [25, p. 158], the Tennent model (and also the Tennent-O'Hearn model) *does* satisfy this equivalence—without additional constraints on command meanings. A discussion of this equality appears in [13, §5, Example 1], the bulk of the justification being a proof of our Lemma 13.

Example 2 *The block below always diverges.*

```
begin
    new x;
    x := 0;
    P;                                  % P is declared elsewhere
    if contents(x) = 0 then diverge fi
end
```

The translation of this block into EoA would be something like:

$$\begin{array}{l}\mathsf{New}\ \iota, \kappa \leftarrow 0\ \mathsf{in} \\ \quad P; \qquad\qquad \%\ P \textit{ is declared elsewhere} \\ \quad \mathsf{IFexp}_{\mathsf{exp}}(\kappa = 0)\ \textit{diverge}\ \mathsf{skip}\end{array}$$

This Example is not explicitly discussed in [13], but it is quite simple to prove. Specifically, let C be the above EoA command. We show for arbitrary worlds X and Y, environments $u \in \mathrm{Env}(X)$, morphisms $f : X \to Y$ and states $y \in Y$, that $[\![C]\!]Xufy$ is undefined. Let $p = u(P)$. Since $\times\mathbb{N}; f \times \mathbb{N} = f; \times\mathbb{N}$, we have $p(\times\mathbb{N}; f \times \mathbb{N}) = p(f; \times\mathbb{N})$. Suppose that $p(f; \times\mathbb{N})(y.0)$, is defined, say

it is $y'.n$. We must show that n is 0. Let $Y' = Y \times \{0\}$. By the uniformity condition on elements of $[\![\mathsf{comm}]\!]$:

$$\underbrace{\Big(S(\lceil Y' \rceil; p(f; \times\mathbb{N})\Big)(y.0)}_{\| \; y'.n} = \Big(p(f; \times\mathbb{N}; \lceil Y' \rceil; S(\lceil Y' \rceil)\Big) y.0.$$

Any state which is equal to the right hand side must lie in the set Y'. Thus, $y'.n$ is of the form $y'.0$, and so $n = 0$. ■

Example 3 *The blocks*

$$\textbf{begin new } x\textbf{; new } y\textbf{; } x := 0; y := 0; Q(x, y) \textbf{ end}$$

and

$$\textbf{begin new } x\textbf{; new } y\textbf{; } x := 0; y := 0; Q(y, x) \textbf{ end}$$

are equivalent.

The EoA version of the first block is:

$$\mathsf{New}\ \iota_1,\ \kappa_1 \leftarrow 0\ \mathsf{in}\ (\mathsf{New}\ \iota_2,\ \kappa_2 \leftarrow 0\ \mathsf{in}\ Q(\iota_1)(\kappa_1)(\iota_2)(\kappa_2))$$

The EoA version of the second block is:

$$\mathsf{New}\ \iota_1,\ \kappa_1 \leftarrow 0\ \mathsf{in}\ (\mathsf{New}\ \iota_2,\ \kappa_2 \leftarrow 0\ \mathsf{in}\ Q(\iota_2)(\kappa_2)(\iota_1)(\kappa_1))$$

where Q is an identifier of type $\mathsf{acc} \to \mathsf{exp} \to \mathsf{acc} \to \mathsf{exp} \to \mathsf{comm}$. This equality was also discussed in [13, §5, Example 2]. They say: "The equivalence can be shown by a straightforward calculation using an endomorphism that exchanges the two $\mathbb{N}$-valued components in a world of the form $X \times \mathbb{N} \times \mathbb{N}$, thus exchanging the declared variables." Properly using this endomorphism will require repeatedly moving it across the commutativity conditions required by the functor $\Rightarrow$. We then observe that this endomorphism is also an isomorphism in the category of possible worlds, thus we can use Lemma 13 to move the endomorphism across the command. ■

Example 4 *The block below always diverges.*

```
begin
    new x; new y;
    procedure Twice; begin y := 2*contents(y) end;
    x := 0; y := 0
    Q(Twice);                          % Q is declared elsewhere
    if contents(x) = 0 then diverge fi
end
```

For the translation of this block into EoA, we beta-reduce the desugared version of the procedure declaration to get:

$$\begin{array}{l}\mathsf{New}\ \iota_1, \kappa_1 \leftarrow 0\ \mathsf{in}\\ \quad \mathsf{New}\ \iota_2, \kappa_2 \leftarrow 0\ \mathsf{in}\\ \qquad Q(\iota_2\,(2 * \kappa_2))\,;\\ \qquad \mathsf{IFexp}_{\mathsf{comm}}(\kappa_1 = 0)\ \mathit{diverge}\ \mathsf{skip}\end{array}$$

This example is more complicated than the earlier ones, and is not specifically addressed in [13], but it is easily handled using the methods developed there. Specifically it is handled by the semantic definition of non-interference (#) which they give in Section 3. One merely needs to formalize the usual intuition. We have the following:

$$\begin{array}{rl} Q\#\kappa_1 & \text{(since } Q \text{ is non-local)}\\ \iota_2(2 * \kappa_2)\#\kappa_1 & \text{(obvious)}\\ Q(\iota_2(2 * \kappa_2))\#\kappa_1 & \end{array}$$

where the last step comes from the semantic version of the following "procedure-call law": if $C : \theta \rightarrow \theta'$ and an identifier c is not free in C or E then $C\#E \Rightarrow (\forall c{:}\theta.c\#E \rightarrow C(c)\#E)$. From this we can conclude that after the execution of $Q(\iota_2(2 * \kappa_2))$ the value of κ_1 is unchanged, namely 0. ■

Example 5 *The block below always diverges.*

```
begin
    new x;
    procedure Add_2;   %Add_2 is the ability to add 2 to x
        begin x := contents(x) + 2 end
    x := 0;
    Q(Add_2);                % Q is declared elsewhere
    if contents(x) mod 2 = 0 then diverge fi
end
```

For the translation of this block into EoA we again beta-reduce the desugared version of the procedure declaration getting:

$$\begin{array}{l}\mathsf{New}\ \iota, \kappa \leftarrow 0\ \mathsf{in}\\ \quad Q(\iota\,(\kappa + 2))\,;\\ \quad \mathsf{IFexp}_{\mathsf{exp}}(\kappa \bmod 2 = 0)\ \mathit{diverge}\ \mathsf{skip}\end{array}$$

O'Hearn and Tennent also discuss this example in [13, §6]. Their argument uses specification logic, rather than a direct semantic proof. Moreover, their argument relies on the principle of non-interference abstraction. Whether or not

this principle is sound for the Tennent model is an open question. The O'Hearn-Tennent model was introduced specifically to overcome this shortcoming. There does, however, exist a semantic proof of this equality for the original Tennent model. Furthermore, the reasoning carries over directly to the O'Hearn-Tennent model. To see how to prove this in the Tennent model, we make the informal observation that

$$Q(\iota\,(\kappa+2))\#(\kappa \bmod 2).$$

The additional constraint on elements of $[\![\mathsf{comm}]\!]$, requires the equivalence class component of the endomorphism $[\kappa \bmod 2]$ must be respected by $[\![Q(\iota\,(\kappa+2))]\!]$. Thus the resulting state (if any) of $Q(\iota\,(\kappa+1))$ must have the same value of $(\kappa \bmod 2)$ as the initial state. ■

As we discuss in the next Section, there is a slight perturbation of the above block whose divergence (in the Tennent model) is an open question.

Example 6 *The block*

```
begin
    new x;
    procedure AlmostAdd_2;
    begin if z = x then x := 1 else x :=contents(x) + 2 fi end
    x := 0;
    Q(Add_2);                          % Q is declared elsewhere
    if contents(x) mod 2 = 0 then diverge fi
end
```

always diverges.

A test for equality of locations is not a part of EoA, nor is it possible to simulate such a test in EoA. Therefore Example 6 is not relevant to EoA.

Example 7 *The block*

begin new x; **procedure** *Add_1*; **begin** $x := Contents(x) + 1$ **end**; $P(Add\_1)$ **end**

is observationally congruent to the block

begin new x; **procedure** *Add_2*; **begin** $x := Contents(x) + 2$ **end**; $P(Add\_2)$ **end**.

The desugared EoA encodings of these blocks are:

$$\mathsf{New}\ \iota,\ \kappa \leftarrow 0\ \mathsf{in}\ (\lambda Add\_1.\, P(Add\_1))(\iota\,(\kappa+1))$$

and

$$\mathsf{New}\ \iota,\ \kappa \leftarrow 0\ \mathsf{in}\ (\lambda Add\_2.\, P(Add\_2))(\iota\,(\kappa+2)).$$

To see why this equality holds, we let $V = \mathbb{N}$ and $V' = \{n \in \mathbb{N} \mid n \text{ is even}\}$. Given any world X, we can define a $\mathbf{W}$-isomorphism $(i, T_{X\times V'}) : X \times V \to X \times V'$, where $i(x.n) = x.(2n)$, and $T_{X\times V'}$ is the universally true binary relation on $X \times V'$. As with the argument for Example 5, we know that $P(Add\_2)\#Parity(\kappa)$, thus the behavior of $P(Add\_2)$ in world $X \times V'$ completely projects into its behavior on states of the form $x.2n$ in world $X \times V$. In other words, for the u that arises in interpreting the second block, we have:

$$[\![P(Add\_2)]\!](X \times V)(u)(\mathrm{id}_{X\times V})(x.n) = [\![P(Add\_2)]\!](X \times V)(u_2))(\mathrm{id}_{X\times V'})(x.n)$$

where $u_2 = \mathrm{Env}(\lceil(X \times V')\rceil)u$. From this we need to show:

$$[\![P(Add\_1)]\!](X \times V)(u_1)(\mathrm{id}_{X\times V})(x.n) = [\![P(Add\_2)]\!](X \times V)(u_2)(\mathrm{id}_{X\times V'})(x.2n)$$

using

$$\begin{aligned} u_1(P) &= [\![\mathsf{comm} \to \mathsf{comm}]\!](\times\mathbb{N})p = p_1 \\ u_2(P) &= [\![\mathsf{comm} \to \mathsf{comm}]\!](\times\mathbb{N}; \lceil(X \times V')) = p_2 \\ c_1 &= u_1(Add\_1) \\ c_2 &= u_2(Add\_2). \end{aligned}$$

It is sufficient to prove the following equality, which follows via an argument analogous to that for Example 3, using the isomorphism $(i, T_{X\times V'})$.

$$p\left(\mathrm{id}_{\times\mathbb{N}}\right)(c_1)(\mathrm{id}_{X\times V})(x.n) = p\left(\mathrm{id}_{\times\mathbb{N};\lceil(X\times V')}\right)(c_2)(\mathrm{id}_{X\times V'})(x.2n)$$

■

There is one other example of an equivalence which the Tennent model satisfies, which is interesting[13, §5,Example 3]. This does not appear as one of the Meyer-Sieber examples, and Sieber claims that this example fails in the Meyer-Sieber model. It is essentially

Example 8 *The following blocks are equivalent:*

```
begin
    new x;
    x := 1;
    P(x);                          % P is declared elsewhere
end
```

and

```
P(1)
```

Operationally, the idea is that P does not have write access to X, so that whenever x is evaluated as part of the call $P(x)$, x must produce 1.

5.3 Failure of Full Abstraction

The fact that the Tennent and O'Hearn-Tennent models correctly handle all of the relevant Meyer-Sieber examples demonstrates the substantial power of these models. Looking at these examples alone, however, is quite misleading. There are slight perturbations of the Meyer-Sieber examples which the Tennent or O'Hearn-Tennent models do not handle (or where their performance is an open problem). Furthermore, there is a simple example which the Meyer-Sieber model (and most other models) handles but the Tennent and O'Hearn-Tennent models do not.

We start with the Example due to O'Hearn [13] which first showed the failure of full abstraction (for both the Tennent and O'Hearn-Tennent models) [13].

Example 9 *The block*

$$\textbf{begin new } x\textbf{; } P(\mathsf{skip})\textbf{ end}$$

is observationally congruent to the block

$$\textbf{begin new } x\textbf{; procedure } \mathit{Add\_1}\textbf{; begin } x := \mathit{Contents}(x) + 1\textbf{; end;}\ P(\mathit{Add\_1})\textbf{ end.}$$

This is just a slight perturbation of Example 7. Intuitively the Tennent and O'Hearn Tennent models fail because even though P does not have complete access to x from *Add_1*, P can check whether or not *Add_1* has a side-effect on x. Specifically, they let $p \in [\![\mathsf{comm} \to \mathsf{comm}]\!]X$ be defined as follows: for any $f : X \xrightarrow{\mathbf{W}} Y$, $c \in [\![\mathsf{comm}]\!]$, $g : Y \xrightarrow{\mathbf{W}} Z$, and $s \in S(Z)$,

$$pfcgs = \begin{cases} s & \text{if } cgs = s, \\ \mathit{undefined}, & \text{otherwise.} \end{cases}$$

Then

$$p(\mathrm{id}_X)([\![\mathsf{skip}]\!]Xu)(\mathrm{id}_X)(s) = s,$$

whereas

$$p(\times\mathbb{N})\Big([\![x := x + 1]\!](X \times \mathbb{N})(u')\Big)(\mathrm{id}_{X\times V})(s.0)$$

is undefined, where u and u' are the appropriate non-local and local environments, respectively. ■

From O'Hearn's counterexample to full abstraction, we manage to construct an example which is handled by most other models, but neither the Tennent model nor the O'Hearn-Tennent model handle.

Example 10 *The block*

$$\textbf{begin new } x\textbf{; } P(x := x + 1; x := x + 1)\textbf{ end}$$

is observationally congruent to the block

$$\textbf{begin new } x;\ P(x := x + 2)\ \textbf{end}$$

Intuitively, this failure is because these models are sensitive to intermediate states encountered during program execution. Specifically, we modify the proof for the preceding example and take instead

$$pfcgs = c(g; \lceil\{z, c(g)z\})z.$$

Then we have

$$p(\times\mathbb{N})\Big([\![x := x + 2]\!](X \times \mathbb{N})(u)\Big)(\mathrm{id}_{X\times V})(s.0) = s.2$$

whereas

$$p(\times\mathbb{N})\Big([\![x := x + 1; x := x + 1]\!](X \times \mathbb{N})(u)\Big)(\mathrm{id}_{X\times V})(s.0)$$

is undefined, where u is the appropriate local environment. ■

Finally, we have a perturbation of Example 5 which is handled by the O'Hearn-Tennent model, but whether or not the Tennent model handles is it an open problem.

Example 11 *The block below always diverges.*

```
begin
    new x;
    procedure Add_2;   %Add_2 is the ability to add 2 to x
        begin x :=contents(x) + 1; x :=contents(x) + 1; end
    x := 0;
    Q(Add_2);                % Q is declared elsewhere
    if contents(x) mod 2 = 0 then diverge fi
end
```

O'Hearn and Tennent justified this example by showing that the O'Hearn-Tennent model satisfies a principle called "Non-Interference Abstraction," which has the following as a special case

$$P\#R\ \&\ \{R\}C\{R\} \Rightarrow \{R\}P(C)\{R\}.$$

In English, this says that if P does not interfere with property R and property R is preserved by C, then property R is preserved by the call $P(C)$. This principle is sufficient to justify the above Example. ■

It is an open problem as to whether or not the Tennent model satisfies the principle of Non-Interference Abstraction. It is also an open problem as to whether or not the Tennent model and the O'Hearn-Tennent model have the same equational theories. Furthermore, in the event that the equational theories differ, it is not obvious whether one must be stronger than the other, or if they could be incomparable.

Chapter 6

Conclusion

We have given the first complete exposition of a Structured Operational Semantics for an ALGOL-like language, and have shown that the Tennent and O'Hearn-Tennent functor category models are adequate with respect to this semantics. As the Tennent and O'Hearn-Tennent models were themselves designed to provide a model of Reynolds's Specification Logic [22], the adequacy result of this thesis can be taken to be an adequacy result for Specification Logic. Thus this thesis provides a concrete connection between Specification Logic and our operational understanding of how ALGOL-like code should behave.

One novel feature used in defining the operational semantics is the handling of New-blocks. Specifically, the use of the initialization for the new variable and the rule (New-eval) seemed to make the technical work in the Adequacy proof flow cleanly.

Having established the adequacy of these models, the next question to ask is "Are they fully-abstract?" Section 5.3 shows that they are not. Another reasonable question to ask is how close do they come? In other words, what "partial full-abstraction" results hold. For example, the model is fully abstract for the equational theory of EoA commands with free identifiers restricted to be of type exp or comm. On the other hand, the counterexamples of Section 5.3 reflect a failure of full abstraction for the equational theory of EoA terms with free identifiers restricted to be of type $\mathsf{comm} \to \mathsf{comm}$ or below. Related to lower-order full abstraction results are "half-full abstraction" results. Specifically, for what collection of C (if any) is the model fully abstract for equalities of the form $C = P$ where P is a program? What about when P is a closed command?

Aside from taxonomy, the natural question is to ask as to whether or not we can "repair" these models so as to either achieve full abstraction, or at least come closer to it. It might be possible to obtain models with equational theories closer to full abstraction by perturbing the model in subtle ways. The generalization of the adequacy proof given in Section 4.3 provides some loose

conditions on these perturbations to insure that they preserve symmetry.

Bibliography

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, second edition, 1984.

[2] H. Bekic. Definable operations in general algebras and the theory of automata and flowcharts. Technical report, IBM Laboratory, Vienna, 1969. Also, pp. 30-55 in *Programming Languages and their Definition*, H. Bekic (ed. C. B. Jones), Lecture Notes in Computer Science **177**, Springer-Verlag, Berlin (1984).

[3] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132. Cambridge Univ. Press, 1985.

[4] Joseph Y. Halpern, Albert R. Meyer, and Boris A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 245–257, 1983.

[5] Joseph Y. Halpern, Albert R. Meyer, and Boris A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? Technical Report MIT/LCS/TM-258, Massachusetts Institute of Technology, April 1984.

[6] Trevor Jim and Albert R. Meyer. Full abstraction and the context lemma. Technical report, MIT Laboratory for Computer Science, 1991.

[7] Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101; 158–165, 1965.

[8] Arthur F. Lent. The category of functors from state shapes to bottomless cpos is adequate for block structure. Master's thesis, Massachusetts Institute of Technology, February 1992.

[9] Albert R. Meyer and Kurt 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, 1988.

[10] Peter D. Mosses and Gordon D. Plotkin. On proving limiting completeness. *SIAM Journal on Computing*, 16:179–194, 1987.

[11] Evelyn Nelson. On exponentiating exponentiation. *Journal of Pure and Applied Algebra*, 20:79–91, 1981.

[12] Peter O'Hearn and Robert Tennent. Semantics of local variables. In P.T. Johnstone M.P. Fourman and A.M.Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1992. Conference held in 1991.

[13] Peter O'Hearn and Robert Tennent. Semantical analysis of specification logic: Part 2. *Information and Computation*, 1993. In Press.

[14] Peter W. O'Hearn. *The Semantics of Non-Interference: a Natural Approach*. PhD thesis, Queen's University, Kingston, Canada, 1990.

[15] Frank J. Oles. *A category-theoretic approach to the semantics of programming languages*. PhD thesis, Syracuse Univ., 1982.

[16] Frank J. Oles. Type algebras, functor categories, and block structure. Technical Report DAIMI PB-156, Aarhus Univ., Computer Science Dept., Denmark, 1982. Proc. U.S.-French Joint Symp. Applications of Algebra to Language Definition and Compilation, to appear.

[17] Frank J. Oles. Type algebras, functor categories, and block structure. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 15, pages 543–573. Cambridge Univ. Press, 1985.

[18] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, December 1977.

[19] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[20] John C. Reynolds. *The Craft of Programming*. Prentice-Hall, Inc., 1981.

[21] John C. Reynolds. The essence of ALGOL. In Jaco W. de Bakker and van Vliet, editors, *Int'l. Symp. Algorithmic Languages*, pages 345–372. IFIP, North-Holland, 1981.

[22] John C. Reynolds. Idealized ALGOL and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

[23] R. D. Tennent. Semantical analysis of specification logic (preliminary report). In Rohit Parikh, editor, *Logics of Programs: Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 373–386. Springer-Verlag, 1985.

[24] R. D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[25] Robert Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, April 1990.

[26] Boris A. Trakhtenbrot, Joseph Y. Halpern, and Albert R. Meyer. From denotational to operational and axiomatic semantics for ALGOL-like languages: an overview. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs: Workshop*, volume 164 of *Lecture Notes in Computer Science*, pages 474–500. Springer-Verlag, 1984. Conference held in 1983.

[27] Glynn Winskel. Introduction to the formal semantics of programming languages. Unpublished Manuscript, 1991.