# Logic from Programming Language Semantics

Thesis by

Young-il Choo

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1987

(Submitted May 21, 1987)

# Acknowledgements

First thanks go to Jim Kajiya for the initial impetus to delve into the issues of logic and the semantics of high-level programming languages, and for the introduction to powerful mathematics. I have benefited much from his insights and have appreciated his patient support over the years as my advisor.

To Alain Martin for many discussions on program verification. The work on infinite lists came out of his class on functional languages with lazy evaluation. I do appreciate his efforts to keep my feet on the ground while my head was intoxicated in abstract mathematics.

I would like to thank Alexander Kechris, Fred Thompson, and Hugh Woodin for being on my reading committee. It was in classes given by Professor Kechris and Professor Thompson that I was introduced to mathematical logic and mathematical theory of languages, respectively.

To Lenny Rudin, my officemate of many years, thanks for the many stimulating late night talks on mathematics and fundamental issues in our science. His work on $\lambda$-logic introduced me to the model of type-free logic which has been a guiding principle for this thesis.

Special appreciation to C.F.F. for gentle encouragement and firm exhortation to keep at the task of writing. A toast to the Sunday evening B.C. crowd for the change of scenery, good food, and fellowship.

Finally, I wish to thank my parents for their support and encouragement during these past several years even while moving between three continents.

# Abstract

Logic for reasoning about programs must proceed from the programming language semantics. It is our thesis that programs be considered as mathematical objects that can be reasoned about directly, rather than as linguistic expressions whose meanings are embedded in an intermediate formalism.

Since the semantics of many programming language features (including recursion, type-free application, infinite structures, self-reference, and reflection) require models that are constructed as limits of partial objects, a logic for dealing with partial objects is required.

Using the $D_\infty$ model of the $\lambda$-calculus, a logic (called *continuous logic*) for reasoning about partial objects is presented. In continuous logic, the logical operations (negation, implication, and quantification) are defined for each of the finite levels and then extended to the limit, giving us a model of type-free logic.

The triples of Hoare Logic are interpreted as partial assertions over the domain of partial states, and contradictions arising from rules for function definitions are analyzed. Recursive procedures and recursive functions are both proved using mathematical induction.

A domain of infinite lists is constructed as a model for languages with lazy evaluation and it is compared to an ordinal-heirarchic construction. A model of objects and multiple inheritance is constructed where objects are self-referential states and multiple inheritance is defined using the notion of product of classes. The reflective processor for a language with environment and continuation reflection is constructed as the projective limit of partial reflective processors of finite height.

# Contents

Chapter 1

# Introduction

Logic for reasoning about programs must proceed from the programming language semantics. It is our thesis that programs be considered as mathematical objects that can be reasoned about directly, rather than as linguistic expressions whose properties are embedded in an intermediate formalism. To achieve this, a clear semantics must be given of the programming language. Once the semantics is given, the logic for programs becomes the informal logic of everyday mathematics.

In this thesis, we explore the relationship of logic to programming language semantics by first showing how a model of the type-free $\lambda$-calculus can be a model of a type-free logic. Then, using this model as our guide, we can interprete the axioms and inference rules of a proof system for an imperative programming language and resolve contradictions that arise from the introduction of inference rules for functions. Next, the semantics of three high-level programming languages are given by constructing their semantic domains.

Three themes underlie this work:

1. Programs are mathematical objects satisfying their defining equations.
2. A logic of programs requires a logic of partial objects.
3. Semantic domains of high-level programming languages with infinitary properties can be constructed using the projective limit.

Our approach to semantics of programming language is known as *denotational semantics*, where the meaning of a program, its denotation, is the composition of the meanings of its parts, and the denotation is given as some object in some semantic domain. The construction of appropriate semantic domains is crucial, and, for unusual programming languages, can be quite non-trivial.

In the development of denotational semantics of programming languages, partial objects play a fundamental role. A logic for dealing with partial objects in their full generality as encountered in denotational semantics will allow us to reason about programs in an equational way.

In object oriented languages, the state becomes denotable and manipulable, joining the other data types as first class citizens. In higher-order applicative languages, all functions are first class citizens. In procedural reflection, even the current environment and continuation are made accessible to the program. The models for these types of programming languages require projective limit construction to handle the self-referential aspects.

In this Introduction, we survey very informally the different types of programming languages and consider a logic for one class of languages. The limitations of such a logic are discussed.

In conclusion, an overview of the thesis is presented.

## 1. Programming Languages and Logics

Programming languages express computation. Logic captures the principles of reasoning. In the following, we consider programming languages and their logics.

### Languages

Programming languages have developed significantly since the advent of the digital computer. Currently one can identify several distinct styles of programming languages. These include the imperative languages where the state is a data structure that is created and changed, the applicative or functional languages where the desired result is constructed from other structures, the logic programming style where a set of first-order formulas specify the constraints that are used to obtain the desired result. In addition, objects and class definitions combine the data and procedural abstraction, the lazy evaluation mechanism allows the manipulation of potentially infinite structures, and finally, the reflection construct makes the program's current environment and other internal entities visible and a part of the manipulable domain.

The development of programming languages seems to move toward the implementation of more sophisticated mechanisms and giving first class status to entities that were previously implicit. The latter has also been termed the reactive principle (Kay [69]). The development of logic programming languages demonstrate the first trend while the object oriented languages and the reflective languages demonstrate the other. Undoubtedly, new language constructs will be created and new styles will continue to develop.

## Logics

Along with the design of programming languages is the issue of reasoning about programs. The need for rigorous methods of reasoning about programs has led to developments in what is called "logics of programs." These logics have been formal systems based on the syntactic structure of the programming language. Due to their syntactic nature, they cannot easily deal with the increasingly sophisticated features of modern high-level programming languages.

One such logic, known as Hoare Logic, is a formal deductive systems based on the partial correctness assertions of Floyd [67] and Hoare [69]. It is designed for reasoning about programs written in imperative assignment based programming languages. Even for this class of languages, the logic is inadequate to deal with complex constructs.

One of the most developed of these logics is the *specification logic* of Reynolds [82], a deductive system for most of the features of an idealized Algol. (A practical introduction may be found in Reynolds [81].)

## Limitations

One crucial deficiency of the partial assertions method is its inability to handle recursively defined functions. We quote from Reynolds [82], p. 122:

> This limitation is an inevitable (though regrettable) consequence of using the predicate calculus for assertions, along with the vital assumption that any expression of the programming language can occur as a term in such assertions. Recursive definition of expressions or function procedures would impose a domain structure upon their meanings that would be incompatible with the free use of quantifiers, because of their lack of continuity. In other words, we cannot deal with nonterminating expressions and function procedures because we cannot deal with nonterminating assertions.

The language of assertions is usually the predicate calculus, and so any assertion is assumed to be well defined. Allowing recursively defined functions means that these functions must be allowed to appear in the assertions, resulting in nonterminating assertions. It is not immediately clear how to provide a suitable semantics for these types of assertions.

One way of getting around this limitation is to systematically translate a recursive function into a state transforming procedure with the same functional behavior, as was done in de Bakker et al. [81]. Further limitation comes from the fact that assertions are implicitly about the underlying state.

The lack of continuity for quantifiers probably comes from models where true and false are ordered, and where lattice join and meet are used to model logical conjunction and disjunction. In this thesis we present a way of dealing with nonterminating assertions in the framework of what we have called *continuous logic*. Once assertions are allowed to be non-terminating, they require a semantics just as programming languages.

Another deficiency is that since the logic is implicitly about the value of the variables in the state, it cannot deal with applicative languages or with object-oriented languages where local states are distributed among the objects.

## Semantics

Rather than develop logics based on the syntactic structure of a programming language, we propose to reason directly with the objects that make up the meaning of a program. At the semantic level the analysis of different types of programming languages can be handled in a uniform manner.

The first step in the semantic analysis of programs is formulating a clean semantics of the programming language. Not only should the semantics deal with all the features of the programming language but also it must be robust and manipulable. The second step is to formalize the properties of the semantic objects. This will give the proper logic.

As pointed out by Lehmann [76], the fixed point semantics allows us to treat recursive definitions as equations to be used in reasoning about programs. We view a programming language as a linguistic tool for representing the objects and functions of our domain of discourse.

# 2. Overview of Thesis

The mathematical notions and tools that underlie the semantics of programming languages are presented in Chapter 2. These are lattices, semilattices and continuous functions, categories and functors, the projective limit construction, and the construction of semantic domains.

In Chapter 3, the $\lambda$-calculus is presented along with its extensional model $D_\infty$. Logical connectives are defined on the model to give us a type-free logical calculus, called Continuous Logic, based on the $\lambda$-calculus. Using this logical calculus we analyze and resolve couple of classic paradoxes.

Chapter 4 deals with the semantics of imperative programming languages with partial states. Using the framework of continuous logic, we examine the proof rules for recursive functions in a Hoare-style logic that lead to contradictions.

In Chapter 5, a domain of infinite lists is constructed using the projective limit of spaces of finite partial lists. We show that this domain contains more lists than a domain of lists constructed using the ordinal hierarchy.

Chapter 6 presents the construction of the semantic domain for object oriented programming languages that contains objects that are self-referencing. Semantics of multiple inheritance is given by using the notion of product of classes.

In Chapter 7, we describe an operational definition of procedural reflection implemented in a version of Lisp. We construct a reflective processor as the limit of reflective processors of finite levels.

Chapter 2

# Mathematical Foundations

*Wherein the necessary mathematical notions are introduced and the foundation laid for the denotational semantics of programming languages of different flavors.*

The notion of "approximation" lies at the heart of the mathematical foundation for denotational semantics. The difference between programs and their denotations is that programs are usually recursively defined, while their denotations are essentially the "unwound" analogs. The approximation relation is a partial ordering, and so it is treated in the framework of lattices and continuous functions over them. For programming language semantics, a slightly weaker notion of semilattices turns out to be more useful.

Semantic domains are defined to be semilattices in which all objects can be defined as limits of "finite" approximations, the algebraic and the continuous semilattices.

For constructing models of programming languages with high-level features such as self-application and self-reference, a powerful technique known as the projective limit (also known as the inverse limit) is presented. The projective limit provides a way of extending finite properties to the limit. It is a way of completing or compactifying a space.

## 1. Lattices and Continuous Functions

The fundamental notion is that of a partial ordering. To it are added notions of directedness and completeness. Basic notions are introduced. For further reading, the first chapter of Gierz et al. [80] provides a comprehensive set of definitions.

**Definition.** A *partially ordered set*, or *poset* for short, is a set $P$ with a binary relation $\sqsubseteq$, which is

(i) reflexive: for all $x \in P$, $x \sqsubseteq x$,

(ii) antisymmetric: if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$,

(iii) transitive: if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$.

A set with a binary relation that is only reflexive and transitive is called a *preorder*. A *least upper bound* or *supremum* of a subset $X$ of a poset, written $\bigsqcup X$ or $\sup X$, if it exists, is the smallest element that is larger than all the elements in $X$. A poset $P$ is *directed* if for any $x$, $y \in P$, there exists a $z \in P$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$; i.e., any finite number of elements from $P$ has a supremum. The dual notions are *greatest lower bound* or *infimum*, written $\inf X$, and *filtered* subsets.

**Notation.** For directed $\{x_i\}$, the following abbreviations shall be used:

$$\bigsqcup_{i=0}^{\infty}\{x_i\} \equiv \bigsqcup_{i}\{x_i\} \equiv \bigsqcup\{x_i\} \equiv \bigsqcup x_i.$$

**Definition.** A function $f : S \to T$ between two posets is called *order preserving* or *monotonic* if $x \sqsubseteq y$ always implies $f(x) \sqsubseteq f(y)$. We say that $f$ *preserves*

(i) *finite sups*, or (ii) *arbitrary sups*, or (iii) *directed sups*

if whenever $X \subseteq S$ is

(i) finite, or (ii) arbitrary, or (iii) directed,

and $\sup X$ exists in $S$, then $\sup f(X)$ exists in $T$ and equals $f(\sup X)$. A *continuous function* is one that preserves directed sups.

**Definition.** A *semilattice* is a poset $S$ in which every nonempty finite subset has an inf. A *sup-semilattice* is a poset $S$ in which every nonempty finite subset has a sup. A poset that is both a semilattice and a sup-semilattice is called a *lattice*. The empty inf (which, if it exists, is the same as $\sup S$, the maximum element of $S$) is called the *top* element of $S$ and is written as $\top$. The empty sup (which, if it exists, is the same as $\inf S$, the minimum element of $S$) is called the *bottom* element of $S$ and is written as $\bot$.

**Definition.** A poset is said to be *complete with respect to directed sets* (shorter: *up-complete*) if every directed set has a sup. A *complete semilattice* is a poset in which every nonempty(!) subset has an inf and every directed subset has a sup. A *complete lattice* is a poset in which *every* subset has a sup and an inf.

Next, we introduce an additional relation on lattices that enables us to capture the notion of "finiteness" of elements.

**Definition.** Let $L$ be a complete lattice. We say that $x$ *is way below* $y$, written $x \ll y$, if for directed subset $D \subset L$ the relation $y \sqsubseteq \sup D$ always implies the existence of a $d \in D$ with $x \sqsubseteq d$. An element satisfying $x \ll x$ is said to be *isolated from below*, *compact* or *finite*.

**Definition.** A lattice $L$ is *algebraic* if it is complete and for all $d \in L$, the set

$$S_d = \{z \mid z \sqsubseteq d \text{ and } z \text{ is compact}\}$$

is directed and $d = \sup S_d$. If the set of compact elements is countable, then $L$ is $\omega$*-algebraic*.

Since for us, we only deal with $\omega$-algebraic lattices, they just be called algebraic.

**Definition.** A lattice is called a *continuous lattice* if $L$ is complete and satisfies the axiom of approximation:

$$(A) \qquad x = \sup\{u \in L \mid u \ll x\}.$$

**Remark.** By definition, an algebraic lattice is a continuous lattice. Also, the criteria for algebraic and continuous can be applied to semilattices.

We state without proof an essential property of complete lattices, the fixed-point theorem of Tarski [55].

**Theorem.** *Let $f : L \to L$ be a monotone self-map on a complete lattice $L$. Then the set $M = \{x \in L : x = f(x)\}$ of fixed points of $f$ forms a complete lattice in itself.*

A consequence of the fixed-point theorem is that for any monotone self-map $f$, the least fixed-point of $f$ is given by $f^{\alpha}(\bot)$ for some ordinal $\alpha$, where composition into the transfinite is done in the usual way. For continuous self-maps, the least fixed-point is reached at $f^{\omega}(\bot)$, where $\omega$ is the first infinite ordinal.

Clearly, the fixed-point theorem also works for complete semilattices.

**Notation.** Since our interest is with semilattices, from this point on, when we use the term "lattice" informally, we shall mean both lattices and semilattices in the strict sense.

# 2. Categories

The notion of categories is introduced to define the appropriate mathematical structures that will serve as our semantic domains. The categorial concepts introduced here have been taken from Barr and Wells [85] and Mac Lane [71].

## Definition of Category

A *category* $C$ consists of two collections, $\mathrm{Ob}\, C$, whose elements are the *objects* of $C$, and $\mathrm{Ar}\, C$, the *arrows* (or *morphisms* or *maps*) of $C$. To each arrow is assigned a pair of objects, called the *domain* and the *codomain* of the arrow. The notation $f : a \longrightarrow b$ means that $f$ is an arrow with domain $a$ and codomain $b$. If $f : a \longrightarrow b$ and $g : b \longrightarrow c$ are two arrows, there is an arrow $g \circ f : a \longrightarrow c$ called the *composite* of $f$ and $g$. For each object $a$, there is an arrow $1_a$ (or just 1, depending on the context), called the *identity* of $a$, whose domain and codomain are both $a$. These data are subject to the following axioms:

(i) For $f : a \longrightarrow b$,

$$f \circ 1_a = 1_a \circ f = f;$$

(ii) For $f : a \longrightarrow b$, $g : b \longrightarrow c$, and $h : c \longrightarrow d$,

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

**Examples.** The prime example of a category is **Set**, whose objects are ordinary sets and whose arrows are set functions. A poset is also a category where the objects are the elements of the poset and there exits exactly one arrow $f : x \longrightarrow y$ if and only if $x \leq y$ in the partial order. The axioms are satisfied since the partial order is reflexive and transitive. Notice that since the symmetric property of the partial order is not used, any pre-order can be regarded as a category.

**Definition.** An arrow $f : a \longrightarrow b$ in a category is an *isomorphism* if it has an *inverse*, namely, an arrow $g : b \longrightarrow a$ for which $f \circ g = 1_b$ and $g \circ f = 1_a$.

**Examples.** In a group considered as a category, every arrow is invertible, whereas in a poset regarded as a category, the only invertible arrows are the identity arrows (which are invertible in any category).

**Definition.** If $C$ is a category, then we define $C^{op}$ to be a category with the same objects and arrows as $C$, but an arrow $f : a \to b$ in $C$ is regarded as an arrow from $b$ to $a$ in $C^{op}$. $C^{op}$ is called the *opposite category* of $C$.

**Example.** Let $\omega$ be the free category generated by the graph

$$0 \to 1 \to 2 \to 3 \to \cdots$$

(i.e., closed under the identity arrow and composition of arrows); then the opposite category $\omega^{op}$ is the free category generated by the graph

$$0 \leftarrow 1 \leftarrow 2 \leftarrow 3 \leftarrow \cdots.$$

**Definition.** An object $s$ is an *initial object* of a category if for any other object $x$ in the category there is a unique arrow $! : s \to x$. The dual notion is a *terminal object* $t$, where for any object $x$, there exists a unique arrow $! : x \to t$. These unique arrows are usually written $!$.

## Functors

Like other mathematical structures, categories also have a notion of morphism. It is natural to define a morphism of categories to be a map that takes objects to objects, arrows to arrows, and preserves domain, codomain, identities and composition.

**Definition.** If $C$ and $D$ are categories, a *functor* $F : C \to D$ is a map for which

(i) If $f : a \to b$ is an arrow of $C$, then $Ff : Fa \to Fb$ is an arrow of D;

(ii) $F1_a = 1_{Fa}$; and

(iii) If $g : b \to c$, then $F(g \circ f) = Fg \circ Ff$.

**Examples.** The monotone maps between posets are functors between posets regarded as categories. There is a large class of forgetful functors that "forget" some of the properties of a category, such as a group becoming a set, or a ring becoming a group.

## Natural Transformations

A functor $F : C \to D$ can be thought of as embedding a "picture" of category $C$ inside category $D$. A natural transformation can be thought of as transforming one "picture" into another.

**Definition.** If $F : C \to D$ and $G : C \to D$ are two functors, $\lambda : F \to G$ is a *natural transformation* from $F$ to $G$ if $\lambda$ is a collection of arrows $\lambda c : Fc \to Gc$, one for each object $c$ of $C$, such that for each arrow $g : c \to c'$ of $C$ the following diagram commutes:

$$\begin{tikzcd} Fc \arrow[r, "\lambda c"] \arrow[d, "Fg"] & Gc \arrow[d, "Gg"] \\ Fc' \arrow[r, "\lambda c'"] & Gc' \end{tikzcd}$$

The arrows $\lambda c$ are called the *components* of $\lambda$. The natural transformation $\lambda$ is a *natural equivalence* if each component of $\lambda$ is an isomorphism in $D$.

## 3. Projective Limits

Projective limits (also known as inverse limits) provide a way of constructing infinite objects that are a "natural" extension of finite ones. They play a crucial role in the construction of semantic domains for programming languages where infinitary properties, such as self-application or infinite lists, enter.

The following is from Dugundji [66].

**Definition.** Let $A$ be a pre-ordered set and $\{Y_\alpha \mid \alpha \in A\}$ be a family of spaces indexed by $A$. For each pair of indices $\alpha$, $\beta$ satisfying $\alpha \sqsubseteq \beta$, assume there is a continuous map $\mu_{\alpha\beta} : Y_\beta \to Y_\alpha$ and that these maps satisfy the following condition: If $\alpha \sqsubseteq \beta \sqsubseteq \gamma$, then $\mu_{\gamma\alpha} = \mu_{\beta\alpha} \circ \mu_{\gamma\beta}$. Then the family $\{Y_\alpha; \mu_{\beta\alpha}\}$ is called a *projective spectrum* (or *inverse spectrum*) over $A$ with spaces $Y_\alpha$ and connecting maps $\mu_{\beta\alpha}$.

Each projective spectrum yields a limit space:

**Definition.** Let $\{Y_\alpha; \mu_{\beta\alpha}\}$ be a projective spectrum over $A$. Form

$$\prod\{Y_\alpha \mid \alpha \in A\},$$

and for each $\alpha$, let $p_\alpha$ be its projection onto the $\alpha$th factor. The subspace

$$\{y \in \prod_\alpha Y_\alpha \mid \forall \alpha, \beta : [\alpha \sqsubseteq \beta] \Rightarrow [p_\alpha(y) = \mu_{\beta\alpha} \circ p_\beta(y)]\}$$

is called the *projective limit* (or *inverse limit*) space of the spectrum and is denoted by $Y_\infty$ or $\varprojlim Y_\alpha$.

According to this definition, a point $y \equiv \langle y_\alpha \rangle \in \prod Y_\alpha$ belongs to $Y_\infty$ whenever its coordinates "match" in the sense that if $\alpha \sqsubseteq \beta$, then $y_\alpha = \mu_{\beta\alpha}(y_\beta)$. This is known as the *consistency condition.*

For more detail on the topological aspects of the projective (or inverse) limit construction, good references are Dugundji [66] and Nagata [85].

The abstract categorical definition is given below.

**Definition.** Let $C$ and $J$ be categories. The *diagonal functor*

$$\Delta : C \rightarrow C^J$$

sends each object $c$ to the constant functor $\Delta c$, the functor that has the value $c$ at each object $i \in J$ and the value $1_c$ at each arrow of $J$. A *limit* for a functor $F : J \rightarrow C$ consists of an object $r$ of $C$, usually written $r \equiv \varprojlim F \equiv F_\infty$ and called the *limit* object (or "inverse limit" or "projective limit") of the functor $F$, together with a natural transformation $\nu : \Delta c \dot{\rightarrow} F$, which is universal among natural transformations $\tau : \Delta c \dot{\rightarrow} F$, for objects $c$ of $C$.

Since $\Delta c : J \rightarrow C$ is the functor constantly $c$, this natural transformation $\tau$ consists of one arrow $t_i : c \rightarrow F_i$ of $C$ for each object $i$ of $J$ such that for every arrow $u : i \rightarrow j$ of $J$ one has $\tau_j = Fu \circ \tau_i$. We call $\tau : c \dot{\rightarrow} F$ a *cone* to the base $F$ from the vertex $c$.

The universal property of $\nu$ says that

(i) It is a cone to the base $F$ from the vertex $F_\infty$;

(ii) For any cone $\tau$ to $F$ from an object $c$, there is a unique arrow $t : c \rightarrow F_\infty$ such that $\tau_i = \nu_i \circ t$ for all $i$. The situation may be pictured as

$$\begin{tikzcd} c \arrow[r, "t"] \arrow[dr, "\tau_i"'] & F_\infty \arrow[d, "\nu_i"] \\ & F_i \end{tikzcd} .$$

It means that the limit $F_\infty$ is the "smallest" object that can be the vertex of a cone with the base $F$.

**Definition.** The dual notion to limit is *co-limit*, where all the arrows in the previous definition are reversed.

# 4. Semantic Domains

In this section we introduce the notions of semantic domains, the mathematical structures for the denotation of programs. The choice of semantic domain depends on the type of characteristic one needs for the particular programming language as well as the mathematical properties one wants for the semantics.

## Categories of Semantic Domains

Semantic domains will be defined and constructed using simpler domains of similar type. It is convenient to think of them as forming categories in which the domain equations defining a specific semantic domain will be solved.

**Definition.** By a *semantic domain* we will mean any algebraic or continuous semilattice or lattice. The category of these lattices with continuous functions as arrows will be denoted: **ASem**, **ALat**, **CSem**, or **CLat**.

In this thesis, the semantic domains will turn out to be algebraic, and most of the time the extra lattice property of having a top element is unnecessary. We will also refer to "semantic domains" as just "domains." The context should make clear whether a semantic domain or the domain of a function is meant.

**Example.** The simplest and yet the most important example of an **ASem** object is the *domain of truth values*, $T$, which looks like

$$\begin{tikzcd}
\mathsf{t} & & \mathsf{f} \\
& \bot \arrow[ul] \arrow[ur] &
\end{tikzcd},$$

where the arrows indicate the partial ordering.

In general, algebraic semilattices can be constructed from sets by adjoining the undefined element as the bottom element, and making them into a "flat" semilattice.

**Example.** The *domain of natural numbers* is an algebraic semilattice where all the natural numbers are greater than $\bot$, and all the numbers are mutually incomparable.

**Notation.** In the computer science literature complete semilattices are known as *complete partial orders*.

## Constructions of Semantic Domains

Given semantic domains, we can construct new ones in a variety of ways.

**Definition.** Let $D$ and $E$ be domains. The *function space domain*, $[D \rightarrow E]$, consists of all continuous functions from $D$ to $E$ with the ordering defined componentwise; i.e., for $f, g \in [D \rightarrow E]$,

$$f \sqsubseteq g \quad \text{if} \quad \forall x \in D f(x) \sqsubseteq g(x)$$

The *product domain*, written $D \times E$, of two domains is a domain consisting of all pairs $(d, e)$ with $d \in D$ and $e \in E$ with the componentwise ordering.

The *sum* or *coproduct*, is a domain

$$D + E \equiv \{\perp, (d, 0), (e, 1) \mid d \in D \text{ and } e \in E\}$$

with $\perp \sqsubseteq (d, 0)$ and $\perp \sqsubseteq (e, 1)$ and the ordering of each injection determined by the separate summands, and two elements from different summands are incomparable.

It is easy to show that the categories of complete lattices and complete semilattices are closed under the constructions described above. It is a little more involved to show that algebraic and continuous lattices and semilattices are also closed under these construction.

Let **CLat** be the category of continuous lattices and **ALat** the category of algebraic lattices. The following result is from Gierz et al. [80], Theorem 2.8 of Chapter II.

**Theorem.** *The functor* $[\cdot \rightarrow \cdot]$ *maps* $\mathbf{CLat}^{op} \times \mathbf{CLat}$ *into* **CLat** *and* $\mathbf{ALat}^{op} \times \mathbf{ALat}$ *into* **ALat**. *In particular,* $[S \rightarrow T]$ *is a continuous (respectively, algebraic) lattice if* $S$ *and* $T$ *are.*

The proof is in Gierz et al. [80]. A more revealing proof that can be easily modified for semilattices can be found in Scott [72].

## 5. Notation

The substitution operation will be a postfix operator, and $N[x/y]$ will denote the term $N$ with all free occurrences of $x$ replaced by $y$. In mathematical logic this is usually written $[y/x]N$ in prefix notation. Barendregt [81] uses the notation $N[x := y]$ which is the clearest to programmers, but is a little too long.

For any function $f$, let $f[x : e]$ denote the function that is identical to $f$ in all arguments, except that at $x$ its value is $e$.

In general, we shall use square brackets to denote postfix functions.

Chapter 3

# Continuous Logic

*Wherein the λ-calculus is presented and a model constructed, motivating a logic for partially defined objects, thus allowing us to resolve certain paradoxes of type-free logic.*

Logic is an attempt to formalize ways of reasoning about objects in a certain domain. In dealing with denotations of programs, we encounter objects that are partially defined or are given as limits of partially defined objects. For these partially defined objects, the notion of equality is not clear and becomes quite unintuitive when higher order functions over complex domains are involved.

This chapter begins with a short discussion on logic of partial objects and then leads to a quick introduction to the syntax and semantics of Church's λ-calculus (Church [41]). The construction of the model, $D_\infty$, due to Scott, is presented using the projective limit construction.

Next, logical connectives are defined over the base domain $D_0$, and then extended inductively over all $D_n$. Continuous quantifiers are defined by introducing the notion of proper elements in a semantic domain. These logical operations are shown to satisfy the consistency condition, meaning that they are valid operations over $D_\infty$. This gives us a model of a type-free logical calculus. Using this model, we present resolutions to paradoxes due to Curry and to Russell.

## 1. Logic of Partial Objects

Classical logic cannot deal with undefined or partially defined objects. There have been logics developed to specifically handle the notion of undefinedness. Scott [79] formulates a logic where existence can be tested by a predicate. Beeson [85] proposes a different logic of partial objects where the notion of definedness is introduced as a formula constructor with the meaning that a term is defined. In Barringer, Cheng and Jones [84], a natural deduction style of logic for dealing with undefinedness in program proofs is presented. These approaches all assume domains where objects are either defined or not defined.

Goldblatt [79] describes a similar approach where two functions are defined to be equal if they have the same domain of definition and they agree on all the elements of their domain.

These approaches are sufficient for typed objects or functions, but when the domain and codomain of a function may contain partial objects of higher order, like in the $\lambda$-calculus, it is not clear how to determine what the domain of definition should be.

### The Undefined Object

The undefined object was introduced so that a function undefined at an argument may be given some special value that represents undefinedness. In programming language semantics, the undefined object is used to represent lack of information that may come about by non-termination. In denoting infinite objects as the limit of a sequence of partially defined finite objects, the undefined object represents incomplete information at a certain point which may become more defined at a later stage.

Computationally, we think of the undefined object as representing non-termination. Therefore, the test to see whether two non-terminating programs are the same should also be non-terminating, that is, undefined. In a later section, these ideas about undefinedness form the basis for logical connectives that are continuous.

### The Overdefined Object

The overdefined object plays no essential role in the semantics of programming languages, though it may be useful in modeling inconsistency. It was introduced as part of the lattice structure of semantic domains, where it is the inf of the empty set of elements, larger than any other. In Stoy [77] a theorem is proved that says that the overdefined object cannot appear in the semantics of a program unless it is explicitly introduced. Therefore, in this thesis, all the semantics is done with complete semilattices. We will point out any definitions and constructions that are significantly different for lattices.

# 2. The $\lambda$-Calculus

Church's $\lambda$-calculus was developed in order to provide a foundation for mathematics with the function, as a rule for computing the value given an argument, as the fundamental notion. Two key ideas in $\lambda$-calculus are abstraction and application. These give the calculus its ability to model the intuitive notion of computation.

In this section we briefly introduce the syntax and semantics of $\lambda$-calculus. The construction of the model provides insights into construction of other programming language features. For further exposition, Scott [71] presents the original motivation and construction of a model, Stoy [77] contains a concise presentation of the syntax and semantics, while Barendregt's [81] is an encyclopedic treatment of many topics.

## Syntax

The terms of $\lambda$-calculus are generated inductively. Let $M$ and $N$ be metavariables ranging over $\lambda$-terms.

**Definition.** The language of the $\lambda$-calculus consists of variables ($x$, $y$, ...) and special symbols: $\lambda$ and the left and right parentheses.

A *$\lambda$-term* is inductively defined by:

(a) A variable is a $\lambda$-term;

(b) If $x$ is a variable and $M$ is a $\lambda$-term, then $(\lambda x M)$ is a term (called a *$\lambda$-abstraction*);

(c) If $M$ and $N$ are terms, then $(MN)$ is a term (called an *application*).

The standard convention of left association in the application and the Peano dot are adopted to eliminate unreadable parentheses. For example, $ABC \equiv ((AB)C)$ and $\lambda x.xy \equiv \lambda x(xy)$. The usual notions of bound and free variables are assumed with respect to the $\lambda$-abstraction.

## Conversion Rules

Terms embody information. Terms may be converted to others according to the following rules. These conversion rules represent computation.

**Definition.** A *conversion rule* specifies how one $\lambda$-term may be transformed to another. There are three rules *reduction rules*:

($\alpha$) $\lambda x.M \ \triangleright_\alpha \ \lambda y.M[x/y]$ provided $y$ is not free in $M$;

($\beta$) $(\lambda x.M)N \ \triangleright_\beta \ M[x/N]$;

($\eta$) $\lambda x.Mx \ \triangleright_\eta \ M$ if $x$ is not free in $M$.

These reductions are called, respectively, $\alpha$, $\beta$, and $\eta$-reduction, and the left hand side of the above rules are called, respectively, $\alpha$, $\beta$, and *$\eta$-redexes*. The reflexive, symmetric, and transitive closure of the above reductions will be denoted by $=$, and two terms that are related by $=$ are called *interconvertible*.

**Definition.** A $\lambda$-term is said to be *in normal form* (n.f.) if it does not contain a $\beta$ or $\eta$-redex as a subterm. A term is said to *have a normal form*, if there is a sequence of reductions from that term to a term in normal form.

Church originally considered any term not having a normal form to be meaningless. In the $\lambda I$-calculus, where abstraction is allowed only for free variables in a term, this interpretation is reasonable, but for the more general $\lambda K$-calculus, interpretation of all terms not having normal forms to be meaningless is actually contradictory (Wadsworth [76]). Wadsworth [76] introduced a weaker notion known as head normal form. An equivalent notion known as solvability was independently introduced by Barendregt [81].

**Definition.** A $\lambda$-term is in *head normal form* (h.n.f.) if it is of the form

$$\lambda x_1 x_2 \cdots x_n . z X_1 X_2 \cdots X_m, \qquad (n \geq 0, m \geq 0),$$

with $z$ a variable and $X_i$ are arbitrary terms. The variable $z$ is known as the *head variable* and $X_i$ as its $i$th argument.

Terms that do not have a h.n.f. reduce to the following form

$$\lambda x_1 x_2 \cdots x_n . ((\lambda x.M)N) X_1 X_2 \cdots X_m, \qquad (n \geq 0, m \geq 0),$$

where $((\lambda x.M)N)$ is known as the *head redex*. All terms of this form are said to be *not in head normal form*.

Terms without h.n.f. will be considered meaningless since no amount of $\beta$-reduction will produce a head variable.

**Example.** Let $\Delta \equiv (\lambda x.xx)$. Then

$$M \equiv \Delta\Delta \ \triangleright_\beta \ \Delta\Delta \ \triangleright_\beta \ \cdots$$

does not have a h.n.f. since it always has a head redex $\Delta\Delta$.

**Example.** On the other hand, let $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Then

$$Y \triangleright_\beta \lambda f.f(Y) \triangleright_\beta \lambda f.f(f(Y)) \triangleright_\beta \cdots$$

has a h.n.f. with head variable $f$. This $\lambda$-term turns out to be very meaningful as an example of a fixed point combinator. Given any term $M$,

$$YM \equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M = M(YM),$$

its fixed point under $=$.

## Construction of $D_\infty$

The unique character of the $\lambda$-calculus that kept it from being taken too seriously in the past was its type-freeness — that any term could be applied to any other term, including itself. This was a major stumbling block in coming up with an extensional model.

What is needed is a model in which an object can be treated both as an argument and as a function that is defined for all objects. Specifically, a domain that is isomorphic to its own function space, one that satisfies the following domain equation:

$$(*) \qquad D \cong [D \to D].$$

Scott [71] first constructed such a domain using the projective limit of a spectrum of finite domains and projection functions between them.

The following has been gleaned from Scott [71], Wadsworth [76], and Rudin [81]. Note that unlike in Scott [71], we use the three element semilattice without the top element. Except for this difference, everything else follows through without any modification.

The construction proceeds in two stages. First, the finite domains are constructed inductively, followed by the projective maps.

**Definition.** Let $D_0 \equiv T$, the semilattice of truth values, and let

$$D_{n+1} \equiv [D_n \to D_n],$$

the function space of continuous functions over $D_n$.

Next, define a pair of mappings $(\phi_n, \psi_n)$, with

$$\phi_n : D_n \rightarrow D_{n+1} \quad \text{and} \quad \psi_n : D_{n+1} \rightarrow D_n,$$

for $n = 0, 1, 2, \ldots$.

Inductively define the following.

$$\begin{aligned} \phi_0(x) &\equiv \lambda y \in D_0.x && (x \in D_0) \\ \psi_0(x') &\equiv x'(\bot_0) && (x' \in D_1, \bot_0 \equiv \inf D_0) \end{aligned}$$

$$\begin{aligned} \phi_{n+1}(x) &\equiv \phi_n \circ x \circ \psi_n && (x \in D_{n+1}) \\ \psi_{n+1}(x') &\equiv \psi_n \circ x' \circ \phi_n && (x' \in D_{n+2}) \end{aligned}$$

$$\begin{tikzcd} \cdots & D_n \arrow[l] \arrow[d, "x"] & D_{n+1} \arrow[l, "\psi_n"'] & D_{n+2} \arrow[l, "\psi_{n+1}"'] & \cdots \arrow[l] \\ \cdots \arrow[r] & D_n \arrow[r, "\phi_n"] & D_{n+1} \arrow[r, "\phi_{n+1}"] \arrow[u, "x'"'] & D_{n+2} \arrow[r] & \cdots \end{tikzcd}$$

The function $\phi_0$ embeds any element into the constant function with that value, while $\psi_0$ gives us the minimum value of the (monotonic) function $x'$, which is its best approximation. For the inductive cases, an element $x$ in $D_n$ is made into an element of $D_{n+1}$ by first projecting its argument down one level so that $x$ can be applied to it, then it is injected back up to give it an answer at the right level. Similarly for the projection, the argument is injected up a level, then the function is applied, and then projected down a level.

**Definition.** The set $\{D_n; \psi_n\}$ is a projective spectrum and its limit is $D_\infty$, which can be represented as

$$D_\infty \equiv \{\langle x_n \rangle_{n=0}^{\infty} \mid x_n = \psi_n(x_{n+1}), x_n \in D_n\}.$$

The selection criterion, $x_n = \psi_n(x_{n+1})$, is known as the *consistency condition.*

The consistency condition singles out unique sequences for each limit. Without the consistency condition there would be many sequences with the same limit and many sequences without any limits at all.

In categorial terms, $D_\infty$ is a limit for the functor $D : \omega^{op} \rightarrow$ **CSem** defined by $D(n) \equiv D_n$ and $D(! : n+1 \rightarrow n) \equiv \psi_n$.

**Notation.** Let $\psi_{nn}$ and $\phi_{nn}$ be identity functions on $D_n$. Then $\psi_{nm} : D_n \rightarrow D_m$ and $\phi_{mn} : D_m \rightarrow D_n$, for $n > m$, are inductively defined by

$$\psi_{n+1\,n} \equiv \psi_n,$$
$$\phi_{n\,n+1} \equiv \phi_n,$$

and, for $k > 1$,

$$\psi_{n+k+1\,n} \equiv \psi_{n+k\,n} \circ \psi_{n+k},$$
$$\phi_{n\,n+k+1} \equiv \phi_{n+k} \circ \phi_{n\,n+k}.$$

Each element of $D_\infty$ defines a function over $D_\infty$ by the following operation.

**Definition.** Let $x \equiv \langle x_n \rangle_{n=0}^{\infty}$ and $y \equiv \langle y_n \rangle n = 0^\infty$ be elements of $D_\infty$. The *application* $x(y)$ is defined by

$$x(y)_n \equiv \bigsqcup_{m=n}^{\infty} \psi_{mn}(x_{m+1}(y_m)).$$

We will denote by $\Phi(x)$ the function defined by the element $x$. This provides an injection

$$\Phi : D_\infty \rightarrow [D_\infty \rightarrow D_\infty].$$

It turns out that just taking $x(y)_n \equiv x_{n+1}(y_n)$ is not sufficient, for this sequence may fail to satisfy the consistency condition for $D_\infty$.

Next, we define the injection and projection functions connecting the limit, $D_\infty$, with each $D_n$.

**Definition.** Let $\phi_{n\infty} : D_n \rightarrow D_\infty$ and $\psi_{\infty n} : D_\infty \rightarrow D_n$ be defined inductively by the following.

For $n = 0$:

$$\phi_{0\infty}(x_0) \equiv \langle x_0, \phi_0(x_0), \phi_{02}(x_0), \ldots, \phi_{0n}(x_0), \ldots \rangle$$
$$\psi_{\infty 0}(x) \equiv x_0, \qquad x \equiv \langle x_n \rangle_{n=0}^{\infty};$$

and, for $n + 1$,

$$\phi_{n+1\infty}(x_{n+1}) \equiv \phi_{n\infty} \circ x_{n+1} \circ \psi_{\infty n}, \qquad x_{n+1} \in D_{n+1},$$
$$\psi_{\infty n+1}(x) \equiv \psi_{\infty n} \circ \Phi(x) \circ \phi_{n\infty}, \qquad x \in D_\infty.$$

Having constructed $D_\infty$, the limit of all the finite $D_n$'s, the question remains of whether we need to continue and construct

$$D_{\infty+1} \equiv [D_\infty \rightarrow D_\infty]?$$

It turns out that these functions are already represented in $D_\infty$ because they are continuous.

**Definition.** For $f \in [D_\infty \to D_\infty]$, its *representation* is an element $u$ in $D_\infty$, defined by,

$$u_0 \equiv \psi_{\infty 0}(f(\phi_{0\infty}(\bot_0))),$$
$$u_{n+1} \equiv \psi_{\infty n} \circ f \circ \phi_{n\infty}.$$

This operation, denoted $\Psi(f)$, provides an injection

$$\Psi : [D_\infty \to D_\infty] \to D_\infty.$$

## Semantics

The semantics of the $\lambda$-calculus is given by the following meaning function from the free algebra of terms to a domain satisfying the domain equation (*).

---

Syntactic Categories:

| | | |
|---|---|---|
| | $I \in \mathsf{Ide}$ | the usual variables |
| | $E \in \mathsf{Exp}$ | $\lambda$-expressions |

Value Domains:

| | | |
|---|---|---|
| | $\epsilon \in D = [D \to D]$ | values of expressions |
| | $\rho \in U = [\mathsf{Ide} \to D]$ | environments |

Semantic Function: $\mathcal{E} : \mathsf{Exp} \to [U \to D]$

$$\mathcal{E}[\![I]\!]\rho = \rho[\![I]\!]$$
$$\mathcal{E}[\![\lambda I.E]\!]\rho = \lambda\epsilon.\mathcal{E}[\![E]\!](\rho[I : \epsilon])$$
$$\mathcal{E}[\![E_0 E_1]\!]\rho = (\mathcal{E}[\![E_0]\!]\rho)(\mathcal{E}[\![E_1]\!]\rho)$$

Semantics of the $\lambda$-Calculus

---

To deal with the $\lambda$-calculus with atoms, the construction is similar, except that the domain of atoms should be summed with $D_n$ in the inductive definitions, and the projections should preserve atoms between levels.

## 3. Continuous Logical Connectives

The notion of monotonic logical connectives is not new (Stoy [77] calls them doubly additive). They have been used in a logic for dealing with undefinedness in Barrenger [84].

In this section we define the logical connectives and the equality as functions over the truth lattice $T$,

$$\begin{tikzcd}
\mathsf{t} & & \mathsf{f} \\
& \bot \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

using the conditional.

The new direction that is pursued in the following sections is the extension of these functions to $D_\infty$. They are defined inductively for all $D_n$, and then shown to satisfy the consistency condition.

**Definition.** The *continuous conditional* extends the standard conditional in the minimal manner and is defined by the following:

$$T \times T \times T \longrightarrow T$$

$$(x \rightarrow y, z) \equiv \begin{cases} y, & \text{if } \mathsf{t} \sqsubseteq x \text{ and } \mathsf{f} \not\sqsubseteq x; \\ z, & \text{if } \mathsf{t} \not\sqsubseteq x \text{ and } \mathsf{f} \sqsubseteq x; \\ y \sqcap z, & \text{if } \mathsf{t} \not\sqsubseteq x \text{ and } \mathsf{f} \not\sqsubseteq x. \end{cases}$$

This definition is the same as the one given in Scott [71] except that it is defined over the three element truth semilattice, not the four element lattice with a top.

**Definition.** The *continuous conjunction* ($\dot{\wedge}$) is defined using the continuous conditional to be

$$x \mathbin{\dot{\wedge}} y \equiv (x \rightarrow y, \mathsf{f})$$

with its truth table:

$$\begin{tikzcd}
\mathsf{t} & {[\mathsf{t}]} & \mathsf{f} & \bot & {[\bot]} & \mathsf{f} & \mathsf{f} & {[\mathsf{f}]} & \mathsf{f} \\
& \bot \arrow[ul] \arrow[ur] & & & \bot \arrow[ul] \arrow[ur] & & & \mathsf{f} \arrow[ul] \arrow[ur] & .
\end{tikzcd}$$

The truth table should be read as giving the value of the operation whose first argument is the value inside $[\cdot]$, and the second value is the location on the table with the value located at that location. For example, in the above, the value of $\mathsf{t} \mathbin{\dot\wedge} \mathsf{f}$ is the right value of the left table.

**Proposition.** *The continuous conjunction is commutative and associative.*

*Proof:* Commutativity is shown by examination of the truth table. Associativity is by exhaustive checking of the different possibilities.

($\dot\wedge$-A) $$x \mathbin{\dot\wedge} (y \mathbin{\dot\wedge} z) = (x \mathbin{\dot\wedge} y) \mathbin{\dot\wedge} z$$

Clearly, ($\dot\wedge$-A) holds when: (1) all three values are $\mathsf{t}$, since then both sides are equal to $\mathsf{t}$; (2) if any one of them is $\mathsf{f}$, in which case both sides are $\mathsf{f}$. The remaining case is when none is $\mathsf{f}$. This means at least one is $\perp$. In this case both sides of ($\dot\wedge$-A) are $\perp$, and so we are done. $\square$

Note that once the four values for $\mathsf{t}$ and $\mathsf{f}$ have been defined, the other values are forced by continuity considerations. For example, in the left diamond, we begin with $\mathsf{t} \mathbin{\dot\wedge} \mathsf{t} = \mathsf{t}$ and $\mathsf{t} \mathbin{\dot\wedge} \mathsf{f} = \mathsf{f}$. This requires us to make $\mathsf{t} \mathbin{\dot\wedge} \perp$ less than either $\mathsf{t}$ or $\mathsf{f}$, since $\perp = \mathsf{t} \sqcup \mathsf{f}$, namely $\perp$.

**Definition.** The *continuous disjunction* ($\dot\vee$) is defined

$$x \mathbin{\dot\vee} y \equiv (x \rightarrow \mathsf{t}, y)$$

with truth table:

$$\begin{tikzcd}
\mathsf{t} & {[\mathsf{t}]} & \mathsf{t} & \mathsf{t} & {[\perp]} & \perp & \mathsf{t} & {[\mathsf{f}]} & \mathsf{f} \\
& \mathsf{t} \arrow[ul] \arrow[ur] & & & \perp \arrow[ul] \arrow[ur] & & & \perp \arrow[ul] \arrow[ur] &
\end{tikzcd}.$$

**Proposition.** *The continuous disjunction is commutative and associative*

*Proof:* By examination of the truth table, and by exhaustive case analysis. $\square$

**Proposition.** *The continuous disjunction and conjunction satisfy the following distributive laws:*

($\dot\wedge$-D) $$x \mathbin{\dot\wedge} (y \mathbin{\dot\vee} z) = (x \mathbin{\dot\wedge} y) \mathbin{\dot\vee} (x \mathbin{\dot\wedge} z)$$

($\dot\vee$-D) $$x \mathbin{\dot\vee} (y \mathbin{\dot\wedge} z) = (x \mathbin{\dot\vee} y) \mathbin{\dot\wedge} (x \mathbin{\dot\vee} z)$$

*Proof:* The equations hold when the three values are either tor f, since they are then the same as for standard connectives. When one or more of them are undefined, we need to check them separately.

(1) If $x = \perp$, then both sides of ($\dot\wedge$-D) equal $\perp$ and both sides of ($\dot\vee$-D) equal $(y \dot\wedge z)$.

(2) If $y = \perp$, then both sides of ($\dot\wedge$-D) equal $x \dot\wedge z$ and both sides of ($\dot\vee$-D) equal $x$. By symmetry, same holds for $z = \perp$. □

**Definition.** The *continuous implication* is defined

$$x \dot\Rightarrow y \equiv (x \rightarrow y, \mathsf{t})$$

with truth table:

$$\begin{tikzcd}
\mathsf{t} & {[\mathsf{t}]} & \mathsf{f} & \mathsf{t} & {[\perp]} & \perp & \mathsf{t} & {[\mathsf{f}]} & \mathsf{t} \\
& \perp \arrow[ul] \arrow[ur] & & & \perp \arrow[ul] \arrow[ur] & & & \mathsf{t} \arrow[ul] \arrow[ur] & .
\end{tikzcd}$$

For continuous implication, we assert that false implying anything is always true, whether that object is total or not, whereas when true implies something, the outcome is very much dependent on the second argument.

**Definition.** Just as in intuitionistic logic, *continuous negation* ($\dot\neg$) is defined in terms of implication as

$$\dot\neg\, x \equiv x \dot\Rightarrow \mathsf{f} \quad \text{or} \quad \dot\neg\, x \equiv (x \rightarrow \mathsf{f}, \mathsf{t})$$

with truth table:

$$\begin{tikzcd}
\mathsf{f} & & \mathsf{t} \\
& \perp \arrow[ul] \arrow[ur] & .
\end{tikzcd}$$

Note that the negation of undefined is undefined.

**Proposition.** *The continuous conjunction, disjunction, and negation satisfy De Morgan's Laws:*

(M1) $$\dot\neg(x \dot\wedge y) = \dot\neg\, x \dot\vee \dot\neg\, y$$

(M2) $$\dot\neg(x \dot\vee y) = \dot\neg\, x \dot\wedge \dot\neg\, y$$

*Proof:* By checking all the cases. If $x$ and $y$ are both defined or undefined, the equations are satisfied.

For (M1), suppose $x = \perp$. If $y = \mathsf{t}$, then both sides are $\perp$; if $y = \mathsf{f}$, then both sides are $\mathsf{t}$.

For (M2), suppose $x = \perp$. If $y = \mathsf{t}$, then both sides are $\mathsf{f}$; if $y = \mathsf{f}$, then both sides are $\perp$. $\square$

## Equality

In the literature on logic of partial objects, there are usually two notions of equality. One is monotonic on both its arguments, usually called equality; the other is not monotonic, producing the value true when both sides are undefined. The monotonic equality is the equality of the object language, while the non-monotonic equality is what we use in the metalanguage when dealing with the elements of the semantic domains.

**Definition.** The *continuous equality* over the lattice of truth values is defined to be

$$x \doteq y \equiv x \to (y \to \mathsf{t}, \mathsf{f}), (y \to \mathsf{f}, \mathsf{t}),$$

or,

$$x \doteq y \equiv \begin{cases} \mathsf{t}, & \text{if } x = y = \mathsf{t} \text{ or } x = y = \mathsf{f}, \\ \mathsf{f}, & \text{if } (x = \mathsf{t} \text{ and } y = \mathsf{f}) \text{ or } (x = \mathsf{f} \text{ and } y = \mathsf{t}), \\ \perp, & \text{otherwise.} \end{cases}$$

and its truth table is given below.

$$\begin{tikzcd}
\mathsf{t} & {[\mathsf{t}]} & \mathsf{f} & \perp & {[\perp]} & \perp & \mathsf{f} & {[\mathsf{f}]} & \mathsf{t} \\
& \perp \arrow[ul] \arrow[ur] & & & \perp \arrow[ul] \arrow[ur] & & & \perp \arrow[ul] \arrow[ur] &
\end{tikzcd} .$$

**Proposition.** *The continuous equality is associative, symmetric and transitive, but not reflexive.*

*Proof:* Associativity holds when all three values are defined. When any one of them is not defined, then both sides are undefined.

Since $\perp \doteq \perp = \perp$, reflexitivity fails. By the truth table, continuous equality is symmetric.

Assume $x \doteq y$ and $y \doteq z$. If $x = \mathsf{t}$, then $y = \mathsf{t}$, which implies $z = \mathsf{t}$. Similarly, for $x = \mathsf{f}$. $\square$

In this section, we presented the logical connectives and the equality over $T$, and showed that they satisfy the usual algebraic properties, namely, the conjunction and disjunction are commutative and associative, and they distribute over each other, and with negation, they satisfy de Morgan's laws.

## 4. Extension to $D_\infty$

This section presents our new work in the definition of logical connectives over $D_\infty$.

With the logical connectives defined over $D_0$, we want to extend it for all $D_n$'s, and eventually to $D_\infty$. Like almost everything else in this thesis, this will be done inductively. For brevity, let $\odot$ represent any one of $\dot\wedge$, $\dot\vee$, or $\dot\Rightarrow$.

For all elements $x_0$ and $y_0$ in $D_0$, $\odot_0$ was defined such that $x_0 \odot_0 y_0$ is in $D_0$ and it is continuous in both arguments. Inductively we extend the logical connectives to all $D_n$.

**Definition.** Assume that $\odot_n$ has been defined over $D_n$, and $x_{n+1}$ and $y_{n+1}$ are elements of $D_{n+1}$. For all $n$,

$$x_{n+1} \odot_{n+1} y_{n+1} \equiv \lambda d \in D_n . x_{n+1}(d) \odot_n y_{n+1}(d).$$

For this definition to be useful, $x_{n+1} \odot_{n+1} y_{n+1}$ should again be an element of $D_{n+1}$ and the connective should be continuous in both of its arguments.

**Lemma.** *For each $n$,*

(i) *if $x_n$ and $y_n$ are elements of $D_n$, then so is $x_n \odot_n y_n$, and*

(ii) *the logical connectives, $\odot_n : D_n \times D_n \to D_n$, are continuous in both arguments.*

*Proof:* The proof will be by mutual induction on $n$.

For $n = 0$, (i) and (ii) are both true by definition of the connectives.

For $n+1$, to prove (i) we need to show that if $x_{n+1}$ and $y_{n+1}$ are in $D_{n+1}$, then $x_{n+1} \odot_{n+1} y_{n+1}$ is a continuous function over $D_n$. Assume that $\odot$ is continuous over $D_n$. Let $\{d^i\}$ be a chain in $D_n$. First we show that $x_{n+1} \odot_{n+1} y_{n+1}$ is a continuous function over $D_n$.

$$\begin{aligned}
[x_{n+1} \odot_{n+1} y_{n+1}](\bigsqcup_i d^i) &\equiv x_{n+1}(\bigsqcup_i d^i) \odot_n y_{n+1}(\bigsqcup_i d^i) \\
&= \bigsqcup_i x_{n+1}(d^i) \odot_n \bigsqcup_i y_{n+1}(d^i) \\
&\qquad \{\text{by continuity of } x_{n+1} \text{ and } y_{n+1}\} \\
&= \bigsqcup_i [x_{n+1}(d^i) \odot_n y_{n+1}(d^i)] \\
&\qquad \{\text{by hypothesis of continuity of } \odot_n\} \\
&\equiv \bigsqcup_i [x_{n+1} \odot_{n+1} y_{n+1}](d^i).
\end{aligned}$$

Next, for (ii), assume that $x_{n+1} \odot_{n+1} y_{n+1}$ is in $D_{n+1}$. Let $\{x^i_{n+1}\}$ be a chain and $y_{n+1}$ be an element of $D_{n+1}$. Then

$$\begin{aligned}
\bigsqcup_i x^i_{n+1} \odot_{n+1} y_{n+1} &\equiv \lambda d \in D_n.[\bigsqcup_i x^i_{n+1}](d) \odot_n y_{n+1}(d) \\
&\qquad \{\text{by definition of } \odot_{n+1}\} \\
&= \lambda d \in D_n. \bigsqcup_i x^i_{n+1}(d) \odot_n y_{n+1}(d) \\
&\qquad \{\text{by definition of supremum}\} \\
&= \lambda d \in D_n. \bigsqcup_i [x^i_{n+1}(d) \odot_n y_{n+1}(d)] \\
&\qquad \{\text{by hypotheses of continuity of } \odot_n\} \\
&= \bigsqcup_i [\lambda d \in D_n.[x^i_{n+1}(d) \odot_n y_{n+1}(d)]] \\
&\qquad \{\text{by continuity of abstraction}\} \\
&\equiv \bigsqcup_i [x^i_{n+1} \odot_{n+1} y_{n+1}]. \quad \square
\end{aligned}$$

Next, we extend the connectives to $D_\infty$. We write $\langle x_n \rangle$ to mean $\langle x_n \rangle_{n=0}^{\infty}$.

**Definition.** Let $x \equiv \langle x_n \rangle$ and $y \equiv \langle y_n \rangle$ be elements of $D_\infty$. Define

$$x \odot y \equiv \langle x_n \odot_n y_n \rangle_{n=0}^{\infty} .$$

From the previous Lemma, each component is in the correct domain, but is the whole thing in $D_\infty$? To show that it is, we need to show that the consistency condition holds: $x_n \odot_n y_n = \psi_n(x_{n+1} \odot y_{n+1})$.

**Theorem.** *If $x$ and $y$ are in $D_\infty$, then so is $x \odot y$. Equivalently, for all $n$, the following diagram commutes:*

$$\begin{tikzcd} D_{n+1} \times D_{n+1} \arrow[r, "\odot_{n+1}"] \arrow[d, "\psi_n \times \psi_n"] & D_{n+1} \arrow[d, "\psi_n"] \\ D_n \times D_n \arrow[r, "\odot_n"] & D_n \end{tikzcd} .$$

*Proof:* For $n = 0$:

$$\begin{aligned} \psi_0(x_1 \odot_1 y_1) &\equiv [x_1 \odot_1 y_1](\bot_0) \\ &= x_1(\bot_0) \odot_0 y_1(\bot_0) \\ &\equiv \psi_0(x_1) \odot_0 \psi_0(y_1). \end{aligned}$$

For $n + 1$, assume the diagram commutes for $n$:

$$\begin{aligned} \psi_{n+1}(x_{n+2} \odot_{n+2} y_{n+2}) &\equiv \psi_n \circ [x_{n+2} \odot_{n+2} y_{n+2}] \circ \phi_n \\ &\qquad \{\text{by definition of } \psi_{n+1}\} \\ &= \lambda d \in D_n.[\psi_n \circ [x_{n+2} \odot_{n+2} y_{n+2}] \circ \phi_n](d) \\ &\qquad \{\text{by } \eta\text{-abstraction}\} \\ &= \lambda d \in D_n.\psi_n([x_{n+2} \odot_{n+2} y_{n+2}](\phi_n(d))) \\ &= \lambda d \in D_n.\psi_n(x_{n+2}(\phi_n(d)) \odot_{n+1} y_{n+2}(\phi_n(d))) \\ &\qquad \{\text{by definition of } \odot_{n+2}\} \\ &= \lambda d \in D_n.\psi_n(x_{n+2}(\phi_n(d))) \odot_{n+1} \psi_n(y_{n+2}(\phi_n(d))) \\ &\qquad \{\text{by induction hypothesis}\} \\ &= \lambda d \in D_n.[\psi_n \circ x_{n+2} \circ \phi_n](d) \odot_n [\psi_n \circ y_{n+2} \circ \phi_n](d) \\ &= [\psi_n \circ x_{n+2} \circ \phi_n] \odot_{n+1} [\psi_n \circ y_{n+2} \circ \phi_n] \\ &\qquad \{\text{by definition of } \odot_{n+1}\} \\ &\equiv \psi_{n+1}(x_{n+2}) \odot_{n+1} \psi_{n+1}(y_{n+2}). \quad \Box \end{aligned}$$

The key point is that $x_n \odot y_n$ is an element of $D_n$ for each $n$, and therefore the projections, $\psi_n$, are defined for them. The componentwise definition is what makes the consistency condition simple to verify.

**Proposition.** *The logical connectives over $D_\infty$ satisfy the same algebraic properties as the ones defined over $D_0$.*

*Proof:* By the propositions in the previous section, the logical connectives over $D_0$ satisfy the algebraic properties.

For the inductive case, we demonstrate the proof using the distributive law. Let $x$, $y$, and $z$ be elements of $D_{n+1}$. Then

$$\begin{aligned} x \,\dot{\wedge}(y \,\dot{\vee}\, z) &= \lambda d \in D_n . x(d) \,\dot{\wedge}(y(d) \,\dot{\vee}\, z(d)) \\ &= \lambda d \in D_n .(x(d) \,\dot{\wedge}\, y(d)) \,\dot{\vee}(x(d) \,\dot{\wedge}\, z(d)) \\ &= (x \,\dot{\wedge}\, y) \,\dot{\vee}(x \,\dot{\wedge}\, z). \end{aligned}$$

Similarly for all the other properties. $\square$

## 5. Continuous Quantification

So far we have defined the connectives for each level, and then shown the extension to $D_\infty$ to be well defined. Next, we tackle quantification.

The standard definition of quantifiers, where the bound variable ranges over all the elements of a set, is not very meaningful when applied to a domain containing partially defined objects. It ignores the approximation relation between objects, and the undefined element will tend to dominate in assigning the value of the quantifier.

The key idea in continuous quantification over a domain is that since predicates are continuous, the value of a predicate at a partial object should be an approximation of the value of the predicate at some "proper" object. Then, the quantifiers can be defined so that the bound variable ranges over only "proper" objects, for some notion of "proper."

In the truth lattice, $T$, it is clear that the proper elements are the constants t and f, and the only improper element is $\perp$. For functions over domains, the most natural definition would be that a function is proper if it maps proper values to proper values.

**Definition.** For a domain $D$, let $\pi D$ denote the set of *proper elements*. We set $\pi T \equiv \{\mathsf{t}, \mathsf{f}\}$. In general, for any flat semilattice, the proper elements are the defined elements.

The proper elements of constructed domains are given below.

**Definition.** For domains $D$ and $E$:

(1) $\pi(D \times E) \equiv \pi D \times \pi E$;

(2) $\pi(D + E) \equiv \pi D + \pi E$;

(3) $\pi[D \to E] \equiv \{f \in [D \to E] \mid x \in \pi D \text{ implies } f(x) \in \pi E\}$.

Continuous quantification is defined for each $D_n$ inductively. By considering a predicate as a function of one variable (Currying if there are many arguments),

$$p \in [A \to B],$$

quantification is defined as operators of type

$$\dot{\forall}, \dot{\exists} : [[A \to B] \to B].$$

**Definition.** Let $p_{n+1}$ be in $D_{n+1}$.

(1) The *universal quantification* of $p_{n+1}$ is given by

$$\dot{\forall} p_{n+1} \equiv \bigwedge \{p_{n+1}(y_n) \mid y_n \in \pi D_n\};$$

i.e., the continuous conjunction of all the $p_{n+1}(y_n)$, $y_n \in \pi D_n$.

(2) The *existential quantification* of $p_{n+1}$ is given by

$$\dot{\exists} p_{n+1} \equiv \bigvee \{p_{n+1}(y_n) \mid y_n \in \pi D_n\};$$

i.e., the continuous disjunction of all the $p_{n+1}(y_n)$, $y_n \in \pi D_n$.

Next, the notion of proper element for $D_\infty$ is defined that allows us to define quantification over $D_\infty$.

**Definition.** An element, $x \equiv \langle x_n \rangle$, in $D_\infty$ is said to be *proper*, written $x \in \pi D_\infty$, if

(1) for some $k$, $x_n$ is proper for all $n \geq k$, or

(2) it is maximal among all non-finitary elements under the product ordering.

The elements satisfying criterion (1) are known as the *finitary proper elements.*

The elements where each $x_n$ is improper, are like the partial functions whose limit is a total function.

**Note.** The above definition is for semilattices. For lattices, an element is proper if it is maximal among all potentially proper lists. A list is potentially proper if each component is less than some proper element.

**Notation.** At this point we need to show that this definition is consistent with the ones for each $D_n$. We shall use $>$ and $<$ to mean "strictly greater" and "strictly less than" in the lattice ordering.

**Proposition.** *If $x$ and $y$ are proper in $D_\infty$, then $x(y)$ is also proper in $D_\infty$.*

*Proof:* Suppose $x(y)$ is not proper. Then there exists a proper $u$, such that $u > x(y)$; i.e., for some $k$, $u_n > x(y)_n$, for all $n \geq k$. Let

$$t_{n+1} \equiv \lambda z.(z = y_n \rightarrow u_n \,,\, x(y)_n),$$

an element that behaves like $x$ for all arguments, except that at $y$, it gives $u$ as its value.

Since $\langle t_n \rangle$ may not satisfy the consistency condition, define $x'$, where

$$x'_n \equiv \bigsqcup_{m=n}^{\infty} \psi_{mn}(t_m).$$

Then, $x'$ does satisfy the consistency condition, and furthermore, is strictly greater than $x$, contradicting our assumption that it is proper. $\square$

Next, we extend the definition of properness to functions over $D_\infty$, and show that the representation in $D_\infty$ remains proper.

**Definition.** A function in $[D_\infty \rightarrow D_\infty]$ is said to be *proper* if its restriction to proper elements of $D_\infty$ gives proper elements.

**Proposition.** *If $f$ is a proper function over $D_\infty$, then its representation, $\langle u_n \rangle$, in $D_\infty$, given by*

$$u_0 \equiv \psi_{\infty 0}(f(\phi_{0\infty}(\bot_0))),$$
$$u_{n+1} \equiv \psi_{\infty n} \circ f \circ \phi_{n\infty}$$

*is proper.*

*Proof:* The element $u$ was constructed so that, for all $x$ in $D_\infty$, $f(x) = u(x)$, and, since $f$ is proper, so is $u(x)$, for all proper $x$.

Assume that $u$ is not proper in $D_\infty$. Then there exists a $u'$ such that $u' > u$; i.e., for some $k$, $u'_n > u_n$, for all $n \geq k$. This means that for some $x$ in $D_\infty$, $u'(x)_n > u(x)_n$, contradicting the assumption that $u(x)$ is proper. $\square$

We are now ready to define quantification over $D_\infty$.

**Definition.** Let $p \equiv \langle p_n \rangle_{n=0}^{\infty}$ be in $D_\infty$.

(1) The *universal quantification* of $p$ over $D_\infty$ is given by

$$\dot{\forall} p \equiv \bigwedge\{p(y) \mid y \in \pi D_\infty\} \equiv \langle \bigwedge\{p_{n+1}(y_n) \mid y \in \pi D_\infty\}\rangle_{n=0}^{\infty}.$$

(2) The *existential quantification* of $p$ over $D_\infty$ is given by

$$\dot{\exists} p \equiv \bigvee\{p(y) \mid y \in \pi D_\infty\} \equiv \langle \bigvee\{p_{n+1}(y_n) \mid y \in \pi D_\infty\}\rangle_{n=0}^{\infty}.$$

We prove that the sequences so defined satisfy the consistency condition.

**Proposition.** *Let $p$ be in $D_\infty$. For all $n$,*

$$\psi_n((\dot{\forall} p)_{n+1}) = (\dot{\forall} p)_n \quad \textit{and} \quad \psi_n((\dot{\exists} p)_{n+1}) = (\dot{\exists} p)_n.$$

*Proof:* By straight forward calculation:

$$\begin{aligned}
\psi_n((\dot{\forall} p)_{n+1}) &= \psi_n(\bigwedge\{p_{n+2}(y_{n+1}) \mid y \in \pi D_\infty\}) \\
&= \bigwedge\{\psi_n(p_{n+2}(y_{n+1})) \mid y \in \pi D_\infty\} \\
&= \bigwedge\{\psi_{n+1} p_{n+2}(\psi_n y_{n+1}) \mid y \in \pi D_\infty\} \\
&= \bigwedge\{p_{n+1}(y_n) \mid y \in \pi D_\infty\} \\
&= (\dot{\forall} p)_n.
\end{aligned}$$

The proof for the existential quantifier is identical. $\square$

In Scott [79], quantification is defined to be only over objects that satisfy the existence predicate. What we have done is defined the existence predicate for $D_\infty$.

The value of the predicates as well as everything else are all elements of $D_\infty$, giving us type-free quantification.

## 6. A Type-Free Logical Calculus

Having defined operations over $D_\infty$ that correspond to the logical connectives and quantification, we are now in a position to provide the semantics for a type-free logical calculus based on the $\lambda$-calculus with the terms extended by allowing logical connectives and quantification.

## Syntax

We first define the new terms of this logic, and then define the semantic function for them.

**Definition.** *Continuous Logic* is $\lambda$-calculus with new rules for generating terms:

(d) If $X$ and $Y$ are terms, then so are $X = Y$, $X \Rightarrow Y$ and $\neg X$.

(e) If $P$ is a term, then so are $\forall P$ and $\exists P$.

## Semantics

The semantics of all the formulas are inductively given below for some environment $\rho$.

$$\begin{aligned}
\mathcal{E}[\![X = Y]\!]\rho &\equiv \mathcal{E}[\![X]\!]\rho \mathrel{\dot{=}} \mathcal{E}[\![Y]\!]\rho \\
\mathcal{E}[\![X \Rightarrow Y]\!]\rho &\equiv \mathcal{E}[\![X]\!]\rho \mathrel{\dot{\Rightarrow}} \mathcal{E}[\![Y]\!]\rho \\
\mathcal{E}[\![\neg X]\!]\rho &\equiv \dot{\neg}\, \mathcal{E}[\![X]\!]\rho \\
\mathcal{E}[\![\forall P]\!]\rho &\equiv \dot{\forall}\, \mathcal{E}[\![P]\!]\rho \\
\mathcal{E}[\![\exists P]\!]\rho &\equiv \dot{\exists}\, \mathcal{E}[\![P]\!]\rho
\end{aligned}$$

Semantics of Continuous Logic

In the sequel, we shall simply write $[\![M]\!]$ for $\mathcal{E}[\![M]\!]\rho$.

## Validity

Having defined the denotation of the terms and formulas of the type-free $\lambda$-calculus, the next issue is that of validity. Classically, a formula is valid for a model if its interpretation in that model is true, i.e., the denotation is the truth value "true." As long as the denotations of the formulas are either "true" or "false," this is fine, but for the type-free theory with terms denoting partial elements of $D_\infty$, there are weaker notions that deserve to be considered. These give us a notion of partial validity.

The strong notion of validity, close to the classical one, has formulas valid if they denote $\mathbf{t}$. Under this notion, formulas whose denotations are less than $\mathbf{t}$ in the lattice ordering can be considered to be "partially valid," their validity dependent on how close they are to **t**. Similarly, formulas denoting **f** are totally false, and those whose denotations are less than f are "partially invalid." In this framework, the totally undefined element $\perp$, is the only element that is both partially valid and invalid.

**Definition.** A formula $M$ is *strongly valid*, written $\models_T M$, if $[\![M]\!] = \mathrm{t}$.

Under strong validity, only formulas that in some way already incorporate the constants **t** or **f** can be be valid. Since we are dealing with a domain of partial elements, we consider another notion of validity that uses the identity function over $D_\infty$ as a measure of truth.

First, let $I$ be the identity function over $D_\infty$, and define

$$\mathbf{t}_I \equiv I \doteq I.$$

Then, this element $\mathbf{t}_I$ is a value that says an element is true in so far as it is defined in $D_\infty$. For example, we have that $\mathbf{t}_I(\perp) = \perp$, $\mathbf{t}_I(\mathrm{t}) = \mathbf{t}$, and $\mathbf{t}_I(\mathbf{t}_I) = \mathbf{t}_I$.

**Definition.** A formula $M$ is *partially valid*, written $\models_I M$, if $[\![M]\!] \sqsupseteq \mathbf{t}_I$.

Partial validity allows formulas containing finite elements to be valid. But so far only finitary elements can be valid. Since there are non-finite elements that have been defined to be proper, we would like a notion that allows formulas containing these elements also to be valid.

Since the non-finite proper elements are less than $\mathbf{t}_I$ componentwise, we weaken the requirement to allow formulas that become progressively more "true." This is done by injecting the finite components of $\mathbf{t}_I$ into higher spaces and comparing them with the components of the formula.

**Definition.** The formula $M$ is *valid in the limit* (or *limit valid*), written $\models_\infty M$, if for all $n$, there is a $k \geq n$, such that $[\![M]\!]_k \sqsupseteq \phi_{nk}((\mathbf{t}_I)_n)$. Here the subscripts denote the components.

Clearly, strong and partial validity imply limit validity.

## 7. Paradoxes of Russell and Curry

The quest for a type-free logical calculus ran into a quagmire of paradoxes. As soon as logical notions were defined for the $\lambda$-calculus, a paradox appeared. Attempts were made to prevent paradoxes either by weakening the logic, or by syntactic restrictions to prevent paradoxical formulas from being written. These methods have not been very satisfactory.

In this section we present the kinds of paradoxes that appear when logical connectives are added to the equational theory of the $\lambda$-calculus.

## Equational Theory of the $\lambda$-Calculus

As the framework for Russell's [02] and Curry's paradox (Hindley et al. [72]), we need to formulate an equational theory for the $\lambda$-calculus.

The equational theory of the $\lambda$-calculus consists of the $\lambda$-terms and the following equations. Let $M$, $N$ and $L$ be $\lambda$-terms.

$$(\rho) \qquad M = M$$

$$(\delta) \qquad \frac{M = N}{N = M}$$

$$(\tau) \qquad \frac{M = L \land L = N}{M = N}$$

For all contexts $C[\ ]$, where $C[\ ]$ is a term except for one missing subterm,

$$\text{(Substitution)} \qquad \frac{M = N}{C[M] = C[N].}$$

For the three conversions, the equality is defined for interconvertible terms:

$$(\alpha') \qquad \lambda x.M = \lambda y.M[x/y]$$

$$(\beta') \qquad (\lambda x.M)N = M[x/N]$$

$$(\eta') \qquad \lambda x.Mx = M$$

where in $(\alpha')$, $y \notin FV(M)$; and in $\eta'$, $x \notin FV(M)$.

For any $\lambda$-term $X$, we may assert the validity of $X$, written $\vdash X$, if it is axiomatically true or may be derived according to rules of inference.

$$\text{(Equality Rule)} \qquad \frac{X = Y \land \vdash X}{\vdash Y}$$

We shall abbreviate

$$\frac{\vdash A_1 \land \ldots \land \vdash A_n}{\vdash X}$$

by

$$A_1, \ldots, A_n \vdash X.$$

To the $\lambda$-terms we adjoin two new term formation rules:

If $X$ and $Y$ are terms, then so are $X \Rightarrow Y$ and $\neg X$.

These represent implication and negation, respectively. By allowing such terms, we are quickly led into paradoxes.

## Russell's Paradox

This is the same set theoretic antimony formulated functionally. Define

$$F(f) \equiv \neg f(f).$$

Then immediately we have that

$$F(F) = \neg F(F),$$

and inconsistency.

We claim that the term $F(F)$ must be given the value $\perp$. Then, since in continuous logic, negation of $\perp$ is still $\perp$, there is no contradiction.

Let $M \equiv F(F)$. Then $M = \neg M$, a recursive equation. The standard way of solving for such is by taking the limit of partial applications of the functional defined by the recursive equation.

$$[\![M]\!] \equiv \bigsqcup_{k=0}^{\infty} \tau^k,$$

where

$$\tau^0 \equiv \perp,$$
$$\tau^{k+1} \equiv \dot{\neg}\, \tau^k = \perp$$

Therefore, $[\![M]\!] = \perp$.

Another way of looking at this is by considering the notion of head normal form for $\lambda$-terms introduced by Wadsworth [76]. Since we have extended the language with the logical connectives, we need to know when $\neg A$ is in h.n.f. We propose the following

**Definition.** A term $\neg A$ is in h.n.f if $A$ is in h.n.f.

Then, with this definition we have the next

**Proposition.** *$F(F)$ does not have an h.n.f.*

*Proof:* Since $F(F) \rhd_\beta \neg F(F) \rhd_\beta \neg\neg F(F) \rhd_\beta \cdots$, so it cannot have an h.n.f. $\square$

In the metalanguage we can know when a program will not terminate by analyzing its structure. For example,

$$\ulcorner\textbf{while true do } x := x + 1 \textbf{ od}\urcorner$$

will not terminate, since the constant **true** will never change. Similarly,

$$\ulcorner(\lambda x.xx)(\lambda x.xx)\urcorner$$

has no head normal form, since $\beta$-reductions cannot change its intension. This prompts further rules:

(1) If $A$ is a term with one $\beta$-redex and $A \rhd_\beta A$ and $A$ is not in h.n.f., then $A$ has no h.n.f.

(2) If $A \rhd_\beta \neg A$ and $A$ is not in h.n.f., then $A$ has no h.n.f.

## Curry's Paradox

Curry's paradox begins with the term

$$Z \equiv \lambda z.(zz \Rightarrow (zz \Rightarrow X)), \tag{1}$$

where $X$ is any $\lambda$-term not containing $z$ as a free variable. Next let

$$M \equiv ZZ. \tag{2}$$

Then by $\beta$-reduction we get that

$$\begin{aligned} M &\equiv ZZ \\ &\equiv (\lambda z.(zz \Rightarrow (zz \Rightarrow X)))Z \\ &= ZZ \Rightarrow (ZZ \Rightarrow X) \\ &\equiv M \Rightarrow (M \Rightarrow X). \end{aligned} \tag{3}$$

We use the following axiom from propositional logic

$$\vdash (A \Rightarrow (A \Rightarrow B)) \Rightarrow (A \Rightarrow B) \tag{$*$}$$

and the standard modus ponens

$$X, X \Rightarrow Y \vdash Y. \tag{$MP$}$$

Replacing $X$ by $M$ and $Y$ by $X$ in the axiom $(*)$, we get

$$\vdash (M \Rightarrow (M \Rightarrow X)) \Rightarrow (M \Rightarrow X). \tag{4}$$

But $(M \Rightarrow (M \Rightarrow X)) = M$ by (3). Then using the Equation Rule we get

(5) $$\vdash M \Rightarrow (M \Rightarrow X).$$

From (5) and (4) and modus ponens we have

(6) $$\vdash M \Rightarrow X,$$

and by (5) and (3) we get

(7) $$\vdash M.$$

Finally, by modus ponens we have

$$\vdash X.$$

So we have derived $X$, an arbitrary term occurring in the definition of $Z$ above. The logic is inconsistent.

## Resolution of Curry's Paradox

Consider the term $M$. It turns out to be the fixed point of the function $H \equiv \lambda a.(a \Rightarrow (a \Rightarrow X))$; i.e., $M = YH$, since

$$\begin{aligned} YH &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))H \\ &= (\lambda x.H(xx))(\lambda x.H(xx)) \\ &= (\lambda x.(xx \Rightarrow (xx \Rightarrow X))(\lambda x.(xx \Rightarrow (xx \Rightarrow X)) \\ &\equiv ZZ \equiv M. \end{aligned}$$

What is the value of $M$? Since it is the fixed point of $H$, we can represent it as the limit

$$[\![M]\!] \equiv \bigsqcup_{k=0}^{\infty} \tau^k,$$

where

$$\begin{aligned} \tau^0 &\equiv \bot, \\ \tau^1 &\equiv (\tau^0 \Rightarrow (\tau^0 \Rightarrow [\![X]\!])) \\ &= (\bot \Rightarrow (\bot \Rightarrow [\![X]\!])), \\ \tau^{k+1} &\equiv (\tau^k \Rightarrow (\tau^k \Rightarrow [\![X]\!])) \end{aligned}$$

are the approximations.

The $\tau^k$'s are the partial approximations introduced by Wadsworth [76,78] to give values to terms that do not possess a normal form but do possess a head normal form. The $\tau^k$'s form a chain approximating the value of $M$.

If we interpret the implication as continuous implication, then the value of

$$\tau^1 \equiv \begin{cases} \bot, & \text{if } [\![X]\!] \text{ is } \bot \text{ or f;} \\ \text{t}, & \text{if } [\![X]\!] \text{ is t.} \end{cases}$$

By induction, for all $k$,

$$\tau^k \equiv \begin{cases} \bot, & \text{if } [\![X]\!] \text{ is } \bot \text{ or f,} \\ \text{t}, & \text{if } [\![X]\!] \text{ is t.} \end{cases}$$

Therefore, the value of $M$ will be the same as $\tau^k$, which depends on the value of $X$.

Just as for Rudin [81], our resolution to the paradox depends on the value of the terms being used. We claim that the axiom (∗) can be used only when the appropriate substitution instance evaluates to **t**. This can happen when either the value of $B$ is t, or when both $A$ and $B$ are proper values. Since the value of $M$ is already t when $X$ is already t, deriving $X$ in this case is not inconsistent. When $X$ is not t, then the value of $M$ is $\bot$, and the formula in the axiom (∗) evaluates to $\bot$.

A formula that is always true, is only conditionally true in continuous logic. The axiom (∗) must be rewritten as two axioms:

$$B \vdash (A \Rightarrow (A \Rightarrow B)) \Rightarrow (A \Rightarrow B)$$
$$A \vee \neg A, B \vee \neg B \vdash (A \Rightarrow (A \Rightarrow B)) \Rightarrow (A \Rightarrow B),$$

and read as saying that if the left hand sides are true, then so are the right hand sides. The disjunctions indicate that the values are proper.

The modification necessary for the above axiom illustrates the general difference between classical logic and continuous logic. In continuous logic we may not conclude that something that is not true must be false. Due to this, it is intuitionistic in that objects must be shown to exist, but once that has been done, the logical operators act quite classically.

## 8. Summary

Continuous logic was defined with the principle of keeping all operations continuous. This was so that all expressions in continuous logic would have fixed point, and therefore would be computationally meaningful.

In this section we compare continuous logic with Rudin's $\lambda$-logic, which also has $D_\infty$ as its model.

## Comparison with Rudin's $\lambda$-Logic

Rudin [81] defined a type-free logic, called *$\lambda$-logic*, based on the $\lambda$-calculus. He showed that $D_\infty$ was a complete relatively pseudocomplemented lattice—better known as a complete Heyting algebra—and therefore, a model for $\lambda$-logic. Conjunction was interpreted as lattice meet, disjunction as lattice join, and most significantly, implication was interpreted as relative pseudocomplement. The constants "true" and "false" were interpreted using the top and the bottom elements of $D_\infty$, respectively.

By considering the semantics, Rudin resolved the paradoxes of Russell and Curry by showing that abstraction is meaningful only when done over continuous objects, that is, over elements of $D_\infty$, and that the paradoxes arose in the attempt to get the least fixed point of non-continuous functions. In the paradoxes considered, the heart of the difficulty lay in the fact that implication, interpreted as relative pseudocomplementation, is continuous in the second argument but *not* in the first.

In continuous logic, we defined all the logical connectives, including implication, to be continuous in all arguments. Therefore, discontinuity could be used as a criterion to resolve the paradoxes. As in Rudin's investigation, our resolution to the paradoxes was also achieved by considering the semantics of the terms.

Unlike Rudin, we could not rule out the term $Z$ as invalid on the ground that it was abstracting over a non-continuous object. Our method for resolving the paradoxes was to show that certain formulas (or terms, since we are in a type-free universe) that were thought to be valid were not.

## Limitations of Continuous Logic

Barringer [84] presents a logic for dealing with undefinedness in program proofs. Their definition of the logical connectives is the same as ours over $T$. They list a set of axioms and inference rules that they claim are sound and complete for the three valued truth values. Their proof system uses both monotonic and non-monotonic equality as well as the non-monotonic definedness predicate. The difference is that they only deal with flat lattices and typed functions.

Having defined a logic where all the operations are continuous, the question remains of its significance and applicability. Because of the continuity assumption, it is a weaker system than classical, or even intuitionistic logics. We don't even have $A \Rightarrow A$, for all $A$.

Continuous logic as it stands models the logic we intuitively use when confronted with programs and other mathematical objects that we have defined but are not sure whether they exist. Until there is some proof, one way or another, any proposition we have about such an object must be conditional. So, in so far as the $\lambda$-calculus models what we can construct, continuous logic models the logic we use intuitively.

As to its applicability, continuous logic is a logic of denotations in their extension. When we regard a computer program, however, we know more than its extension, we are able to prove a great deal by analyzing its structure, i.e., its intension. Therefore, it is quite reasonable to add to continuous logic non-continuous operators such as an existence predicate or a properness predicate, as we have done in our meta-language in order to define continuous operators.

The next step would be to try and formalize how we deal with intensions. Some work by Moschovakis [86] indicates possible directions, with the formalization of algorithms as higher order functionals. The difficulty comes in trying to decide when two algorithms are the same.

Chapter 4

# Imperative Programs

*Wherein a formal system for reasoning about imperative programming languages is introduced and found wanting, motivating an analysis in continuous logic using partial states and predicates.*

Imperative programming languages represent computations by defining a state and then specifying a sequence of state transformations, using the assignment statement and various control structures. One approach to formal reasoning of imperative programs is collectively known as Hoare Logic (Floyd [67] and Hoare [69]). We begin this chapter by introducing the axioms and inference rules of Hoare Logic for simple programs, and then present one of its limitations, pointed out by O'Donnell [82], when the programming language is extended by recursive function definitions.

Next, we present a new semantics of Hoare Logic using the ideas of Continuous Logic. We introduce the notion of partial state structures and partial predicates over them. Using this framework, we analyze the axioms and inference rules of Hoare Logic, and resolve contradictions in proofs of recursive functions.

We conclude the chapter with several examples of reasoning about programs.

## 1. Hoare Logic and Its Limitations

Hoare Logic, also known as "axiomatic semantics," is a logic for dealing with programs in a syntax-directed, compositional way. The axioms describe the primitive elements of the language, and the inference rules describe the compositions.

## Axioms and Inference Rules for while-programs

Assume we have a first-order language as the specification language and an imperative programming language. For definiteness, let the specification language be the standard predicate calculus language, and the programming language be Algol 68 (Lindsey and Meulen [80]) with modifications in the syntax when convenient.

**Definition.** A *Hoare triple* is an expression of the form $\{A\}\ P\ \{B\}$, where $A$ and $B$ are formulas of a specification language and $P$ is a well-formed program. The triple is to be interpreted as saying:

> If $A$ is true of a state before the execution of the program $P$ and $P$ terminates, then $B$ will be true of the resulting state.

Because of the assumption regarding the termination of $P$, the Hoare triples are known as *partial correctness assertions.*

The axioms and inference rules of Hoare Logic for while-programs are listed below. They are presented in the natural deduction style whereby the formula below the line can be derived in one step, given the formulas above the line. The inferences without any formulas above the line are the axioms.

These are the rules for a simple fragment of an imperative programming language. There are no rules for functions or procedures. Procedures can be accommodated in an expanded framework where induction rules are added for recursive procedures. Reynolds [81,82] presents such a logic called *specification logic.* The attempt to include rules for recursive functions has been problematic because of the potentially undefinedess of the function, as we shall see below.

## Function Rules

In O'Donnell's critique [82] of Hoare logic, the rules above are used to show that these rules are not inferentially sound. That is, by adding correct new axioms it is possible to derive contradictions.

Add to the above set of rules an inference rule to deal with the occurrence of defined functions in the assertions. This rule is from Clint and Hoare [72].

(Function-1) $$\frac{\{A\}\ P\ \{B\}}{\forall x(A \Rightarrow B[y/f(x)])}$$

(Empty) $$\frac{}{\{A\}\ \{A\}}$$

(Null) $$\frac{}{\{A\}\ \mathsf{null}\ \{A\}}$$

(Fail) $$\frac{}{\{A\}\ \mathsf{fail}\ \{B\}}$$

(Assignment) $$\frac{}{\{A[x/E]\}\ x := E\ \{A\}}$$

(Composition) $$\frac{\{A\}\ P\ \{B\},\ \{B\}\ Q\ \{C\}}{\{A\}\ P;Q\ \{C\}}$$

(Conditional) $$\frac{\{A \wedge B\}\ P\ \{C\},\ \{A \wedge \neg B\}\ Q\ \{C\}}{\{A\}\ \mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{fi}\ \{C\}}$$

(While) $$\frac{\{A \wedge B\}\ P\ \{A\}}{\{A\}\ \mathsf{while}\ B\ \mathsf{do}\ P\ \mathsf{od}\ \{\neg B \wedge A\}}$$

(Consequence) $$\frac{A \Rightarrow B,\ \{B\}\ P\ \{C\},\ C \Rightarrow D}{\{A\}\ P\ \{C\}}$$

Hoare Logic Axioms and Inference Rules

where

$$f = \mathsf{proc}(x)\text{: }\mathsf{local}\ z_1, \ldots, z_n\text{; } P\text{; }\mathsf{return}(y)\ \mathsf{end}.$$

Given these rules, O'Donnell proceeds to derive a contradiction. First define the function

(∗) $$f = \mathsf{proc}(x)\text{: }\mathsf{fail}\text{; }\mathsf{return}(y)\ \mathsf{end}.$$

Then we get the following derivation.

| | | |
|---|---|---|
| (1) | $\{\mathsf{true}\}\ \mathsf{fail}\ \{\mathsf{false}\}$ | Fail |
| (2) | $\forall x.\mathsf{true} \Rightarrow \mathsf{false}$ | Function-1, (1) |

(3) false

O'Donnell remarks that this contradiction is due to the function rule assuming the existence of all the variables occurring in its predicates. To get around this, he introduces a variation, Function-2, introduced by Musser (referenced in O'Donnell [82]), that forces the existence by having it bound by a quantifier.

(Function-2)
$$\frac{\exists y(A[x/E] \Rightarrow B[x/E]) \quad \{A\}\ P\ \{B\}}{A[x/E] \Rightarrow B[x/E, y/f(x)]}$$

Using this function rule, it takes us two derivations to get a contradiction.

| | | |
|---|---|---|
| (1) | $\{\mathsf{true}\}\ \mathsf{fail}\ \{y = 0\}$ | Fail |
| (2) | $\exists y(\mathsf{true} \Rightarrow y = 0)$ | predicate calculus |
| (3) | $\mathsf{true} \Rightarrow f(0) = 0$ | Function-2, (1), (2) |
| (4) | $f(0) = 0$ | predicate calculus, (3) |

| | | |
|---|---|---|
| (1) | $\{\mathsf{true}\}\ \mathsf{fail}\ \{y \neq 0\}$ | Fail |
| (2) | $\exists y(\mathsf{true} \Rightarrow y \neq 0)$ | predicate calculus |
| (3) | $\mathsf{true} \Rightarrow f(0) \neq 0$ | Function-2, (1), (2) |
| (4) | $f(0) \neq 0$ | predicate calculus, (3) |

O'Donnell states that "It is weakly consistent only because of the peculiar restriction that Function-2 may be applied to each function for only one choice of $A$ and $B$." Then he presents a strongly consistent rule by forcing the function to be used in an assignment rule so that, should it be undefined, then the rule will be vacuously valid.

(Function-3)
$$\frac{\{A\}\ P\ \{B\}}{\{A[x/E]\}\ z := G[f(E)]\ \{B[x/E, y/f(E)]\} \quad \text{for } z \notin FV(E).}$$

Even this rule, however, is not consistent if non-strict functions are allowed to occur in $G$. So then, even more involved rules, one for each of the statements in the language, are provided to keep the proof system consistent. We list the more involved rules, one for each of the constructs (assignment, conditional, and while-loop), with the function $f$ defined as in the Function-1 rule above.

(Function-assignment)
$$\frac{\{A\}\ P\ \{B\}}{\{A[x/E] \wedge (B[x/E, y/f(E)] \Rightarrow C[z/G[f(E)]])\}\ z := G[f(E)]\ \{C\}}$$

(Function-conditional)

$$\frac{\{A\}\ P\ \{B\},\ \{C \wedge G[f(E)] \wedge B[y/f(E)]\}\ Q\ \{D\},\quad \{C \wedge \neg G[f(E)] \wedge B[y/f(E)]\}\ R\ \{D\}}{\{A[x/E] \wedge C\}\ \textsf{if}\ G[f(E)]\ \textsf{then}\ Q\ \textsf{else}\ R\ \textsf{fi}\ \{D\}}$$

(Function-while)

$$\frac{\{A\}\ P\ \{B\},\ \{C \wedge G[f(E)]\}\ Q\ \{C\}}{\{A[x/E] \wedge C\}\ \textsf{while}\ G[f(E)]\ \textsf{do}\ Q\ \textsf{od}\ \{C \wedge \neg G[f(E)]\}}$$

O'Donnell then concludes,

> The soundness of rules for function definitions is a slippery issue when function bodies fail, since the normal interpretation of the predicate calculus does not allow for partial functions. So, we consider a predicate calculus formula containing a program-defined function $f$ to be true when it is true for all total functions $f$ consistent with the values computed by the definition of $f$ (Constable and O'Donnell [78]). If the definition fails to halt, then every total function is consistent with all the computed values (there are none), so only assertions that hold for all functions, such as $\forall x f(x) = f(x)$, are true for $f$. The assertion $f(0) = 0$ is only true when the definition of $f$ actually computes the output value 0 on input 0. Under such an interpretation and assuming that all primitive functions are strict, it is *conjectured* that Function-assignment, Function-conditional, and Function-while are inferentially sound. These rules are so inelegant that the proof of soundness is of much less interest than Cook's [78] proof of soundness for the system of Sec. 4 (Emphasis added).

The contradictions occur because the rule for the "Fail" and the rule for the "While," when the loop does not terminate, are invalid. Semantically, "Fail" denotes non-termination and the state resulting "after" the execution of "Fail" must therefore be undefined. Therefore, we cannot say that an arbitrary predicate holds on the resultant state. A similar argument can be made for the while rule; the rule can be validly applied only when termination is assured.

## 2. Reasoning with Recursive Definitions

Consider a program where a recursive function has been declared. In the proof rule for the assignment statement we have that any expression occurring on the right hand side of the assignment statement is incorporated into the assertions. The assignment rule says: For all assertion $P$, variable $x$, and expression $E$ with consistent type,

$$\{P[x/E]\}\ x := E\ \{P\},$$

where $P[x/E]$ denotes a new assertion with all occurrences of $x$ replaced with expression $E$.

Another example. Suppose *fact* is defined recursively. Then we would want to assert something like the following:

$$\{\mathit{fact}(3) = 6 \wedge y = 3\}\ x := \mathit{fact}(y)\ \{x = 6 \wedge y = 3\}$$

The problem, as Reynolds points out, is that a recursive definition may not be totally defined. So that if $\mathit{fact}(3)$ is not defined, then what is the meaning of this assertion?

Our approach is to allow states to be partial in the sense that variables may be undefined, and therefore, predicates about a state will also be partial in reflecting that. The state and the predicates must be continuous so that states and predicates may be defined as limits of partial ones.

In the example above, if

$$\mathit{fact} \equiv \bigsqcup \tau^k,$$

where $\tau$ is the defining functional for the factorial function, then

$$\{P\}\ x := \mathit{fact}(y)\ \{Q\}\ \equiv\ \bigsqcup \{P^k\}\ x := \tau^k(y)\ \{Q^k\},$$

where $\tau^k$ are partial functions, $P^k$ and $Q^k$ are partial predicates, and

$$\{P^k\}\ x := \tau^k(y)\ \{Q^k\}$$

are partial assertions for each $k$.

In the following are new formulations of state and assignment statement to make the above precise. First, a notion of state will be presented that can have undefined values, the assignment statement will be defined as state functions, and predicates will be defined as functions from states to the truth lattice.

## 3. States

In imperative programming languages, the fundamental object that a program manipulates through the assignment statement is the state. All the other constructs, such as control mechanisms or declarations, serve to aid in the modification of the state. A clear understanding of the state is critical for reasoning about the meaning of imperative programs.

A state can be considered to be a heterogeneous data structure where the identifiers play the role of selectors and the declarations the role of constructors. The assignment statement can then be considered an operator that takes an identifier and a value, modifies the implicitly given state, and gives a new state as its result.

In the literature on the semantics of the state, a distinction is usually made between the store and the environment. The store carries the binding between internal locations and values, while the environment binds identifiers to internal locations. This model can handle aliasing but has limitations. It cannot handle pointers or references since pointers are internal locations whose values are again internal locations. Nor can it handle constant declarations.

We will generalize the above model of the state by using a function with heterogeneous domain and codomain, which subsumes the two notions of store and environment.

Let $V$ be a semantic domain that includes the integers, the Booleans, the reals, the functions over them, as well as any other data structures one would like. We shall be more specific later. Let Ide be a set of *identifiers*, such as $x$, $y$, and *fact* etc. Let Loc be a flat domain of objects called *locations*. By an abuse of notation, we shall use $x$, $y$, etc. to be metavariables ranging over identifiers and locations, as well as values. The context should make clear its usage as a program identifier or as a metavariable.

**Definition.** A *state structure*, $\sigma$, is a continuous function of type

$$[\mathsf{Ide} + \mathsf{Loc} \rightarrow V + \mathsf{Loc}],$$

with the ordering

$$\sigma_1 \sqsubseteq \sigma_2 \quad \text{iff} \quad \forall x.\sigma_1(x) \sqsubseteq \sigma_2(x).$$

We quickly show that the state structures behave nicely.

**Proposition.** *The collection of state structures forms a semantic domain with the pointwise ordering induced by its codomain.*

*Proof:* Since the codomain of a state structure is a semantic domain, the supremum of a chain of state structures is the state structure that has the supremum of each of its chain of coordinates. $\square$

Since the locations are accessible only if they are bound to some identifier by a chain of locations, we do not want to distinguish state structures that differ at unbound locations.

**Definition.** A location $l$ is *bound* in state structure $\sigma$ if there is an identifier $x$ such that $l = \sigma^n(x)$, for some natural number $n$, where $\sigma^n$ denotes $n$ fold composition of $\sigma$. A location that is not bound will be called *free*.

The free locations cannot be modified and they cannot affect the computation of a program, so states differing only on their free locations are equivalent as far as the programmer is concerned. Using this notion of equivalence, we can define for each equivalence class of states the minimal state under the ordering given above. Rather than defining all these concepts and then proving the following assertion, we make it into a definition.

**Definition.** A state structure is *minimal* if the value at each free location is undefined. For any state structure $\sigma$, let $\min\sigma$ denote the minimal state structure that agrees with $\sigma$ at all identifiers and bound locations.

This minimalization operation usually goes by a more colorful name of "garbage collection"

**Definition.** A *state* is a minimal state structure. The domain of all states will be denoted by $\Sigma$.

This definition of state is sufficient for the semantics of languages like Algol 68, where the structuring of data is done in the domain of values, but is not for languages where the internal locations are used to create data structures, such as in LISP. To accommodate structures such as S-expressions, we need to change the codomain of the state to include pairs of locations:

$$\sigma : \mathsf{Ide} + \mathsf{Loc} \to V + \mathsf{Loc} + \mathsf{Loc} \times \mathsf{Loc}.$$

Since we are restricting our language to Algol 68, in the following, we shall use the simpler definition.

**Example.** A simple state, $\sigma$, with $x$ an integer variable and $y$ bound to the constant 5, can be pictured by

$$\sigma \equiv \begin{bmatrix} x \mapsto l \\ l \mapsto 3 \\ y \mapsto 5 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} x \mapsto l \mapsto 3 \\ y \mapsto 5 \end{bmatrix},$$

where $l$ is a location and everything else is undefined.

To access elements of the value domains, we need names for some of the elements. We will assume that a subset of the identifiers is already bound to certain values. These are the constants of the language. For example, the identifier "3" is always bound to the number 3 and "+" to the integer addition function. In most programming languages certain identifiers are overloaded so that they denote different objects in different contexts. For example, the identifier "+" is used to denote the addition function for integers as well as for reals. For simplicity, we shall assume that different uses of the same identifier are merely syntactic sugar that can be distinguished and replaced by distinct identifiers if necessary.

By putting restrictions on the state, we obtain the more traditional semantic structures.

**Definition.** An *environment* is a state where the identifiers are bound to values but all locations are undefined. A *store* is a state where none of the identifiers are defined.

Clearly, our notion of state is the join, in the lattice order, of an environment and a store, while, given a state, we can extract the environment and store from it. The reason for the more general notion of state is our view that all the constructs in an imperative language are functions or functionals on the state. The functionals include the control structures such as conditionals and loops, while the functions include the declaration and the block constructor as well as the assignment statement.

## 4. The Assignment Statement

The assignment statement transforms a state to a new state. To give precise semantics to it, we first need a few auxiliary notions.

The first notion we need is that of the value of an expression in a state. This is defined inductively on the level of the identifier, where the level indicates the distance to the values.

The second notion is that of replacing the value of the state at an argument by a new value. This combined with the minimalization will give us assignment.

**Definition.** For each defined identifier $x$, let the *level* of the identifier be the number of internal locations that lie between the identifier and a value.

**Example.** If $\sigma$ contains $x \mapsto l_1 \mapsto l_2 \mapsto 2$, then the level of $x$ is 2.

**Definition.** Let $E$ be an expression and $\sigma$ a state. The *value* of the expression $E$ in state $\sigma$, written $\sigma^\bullet(E)$, is inductively defined below. Examples following the definitions are written in Algol 68.

(a) $E \equiv c$, where $c$ is a constant, then

$$\sigma^\bullet(c) \equiv \sigma(c).$$

In most programming languages identifiers such as "1" and "2.34" denote the integer 1 and the real number 2.34, respectively. Also, the standard mathematical functions such as "+" are also given their usual meaning.

(b) $E \equiv k$, where $k$ is an identifier of level 0, i.e., a constant declaration, then

$$\sigma^\bullet(k) \equiv \sigma(k) \equiv \sigma(c),$$

where $c$ is the constant. Example: $\ulcorner \mathsf{int}\ k = c \urcorner$, a constant declaration.

(c) $E \equiv x$, with $x$ of level $n$, then

$$\sigma^\bullet(x) \equiv \sigma^{n+1}(x).$$

The usual variables are of level 1. Higher level variables are known as pointers or reference variables. In Algol 68 a variable is declared by $\ulcorner \mathsf{int}\ x := e \urcorner$, which is an abbreviation for

$$\ulcorner \mathsf{ref\ int}\ x = \mathsf{int\ loc};\ x := e \urcorner,$$

where $\ulcorner \mathsf{int\ loc} \urcorner$ is a function returning the first free location and $e$ is an integer expression.

(d) $E \equiv f(E_1, \ldots, E_n)$, with $E_i$'s expressions and $f$ a function, then

$$\sigma^\bullet(E) \equiv \sigma^\bullet(f)(\sigma^\bullet(E_1), \ldots, \sigma^\bullet(E_n)).$$

For $E \equiv v + 3$, if $\sigma^\bullet(v) = 2$, then

$$\sigma^\bullet(E) = \sigma^\bullet(v) + \sigma^\bullet(3) = 2 + 3.$$

Analogously, we define an operation that returns the location of a variable.

**Definition.** For each variable $x$ of level $n > 0$,

$$\sigma^\circ(x) \equiv \sigma^n(x).$$

$\sigma^\circ$ returns the last location that points to a value.

Next, we define functions that modify the state.

**Definition.** A *state transformer* is a function of type

$$(\mathsf{Ide} + \mathsf{Loc}) \times (V + \mathsf{Loc}) \rightarrow [\Sigma \rightarrow \Sigma],$$

and is written $\sigma\langle a : b\rangle$, where $a$ and $b$ are from the domain and codomain of $\sigma$, respectively, and $\sigma\langle a : b\rangle$ is a new state that differs from $\sigma$ only in that it maps $a$ to $b$.

A state transformer may modify a state to produce a state structure that is not minimal. So, we define a new function, which always minimalizes.

**Definition.** A *state function*, written $\sigma[a : b]$, is the composition of a state transformer followed by the minimalization,

$$\sigma[a : b] \equiv \min(\sigma\langle a : b\rangle).$$

Using the state function we illustrate different types of state-modifying operations.

**Example.** Let

$$\sigma \equiv \begin{bmatrix} x \mapsto l_1 \mapsto 3 \\ y \mapsto l_2 \mapsto 5 \end{bmatrix}.$$

Then we have

$$\sigma[y : \sigma(x)] = \sigma[y : l_1] = \begin{bmatrix} x \mapsto l_1 \mapsto 3 \\ y \mapsto l_1 \mapsto 3 \end{bmatrix},$$

where $y$ is aliased to be the same as $x$;

$$\sigma[\sigma(y) : \sigma(x)] = \sigma[l_2 : l_1] = \begin{bmatrix} x \mapsto l_1 \mapsto 3 & \\ y \mapsto l_2 \mapsto l_1 \mapsto 3 \end{bmatrix},$$

where $y$ is made to be a pointer;

$$\sigma[\sigma^\circ(y) : \sigma^\bullet(x)] = \sigma[l_2 : 3] = \begin{bmatrix} x \mapsto l_1 \mapsto 3 \\ y \mapsto l_2 \mapsto 3 \end{bmatrix},$$

where we have the standard assignment operation.

**Definition.** If $x$ is a variable and $E$ is an expression of the appropriate type, then the assignment statement is defined to be

$$[\![x := E]\!] \equiv \lambda\sigma.\sigma[\sigma(x) : \sigma^{\bullet}(E)],$$

and is written as a postfix operator: $\sigma[\![x := E]\!]$.

Clearly, for identifiers of higher level, one can introduce suitable functions to dereference a pointer variable.

Having defined state structures and assignment statement as state functions, it seems appropriate at this point to compare what we have with the applicative state transition (AST) systems of Backus [78]. AST systems are an attempt to combine the history of states with the cleaner semantics of applicative programming. What Backus does is construct state-like structures using the data-types of his functional programming language (FP) by adding the ability to name objects. Then, the programs written in the formal FP (FFP) are able to model environments and states.

What we have done is model states (and environments) using mathematical structures, and made imperative programs behave like applicative programs over the domain of states. If new constructs are introduced into imperative languages to allow larger scale modification of states, then the two approach will be semantically alike. Our purpose, however, has been to analyze the semantics of imperative languages, rather than try to combine the two approaches.

# 5. Control Structures and Declaration

These notions will be treated very briefly here. We assume that the standard denotational semantics can be defined for whatever we need.

**Definition.** The *conditional* will be written in a variety of ways, depending on the context. It is defined using the continuous conditional

$$\sigma[\![\mathsf{if}(B, S_1, S_2)]\!] \equiv (\sigma(B) \rightarrow \sigma[\![S_1]\!], \sigma[\![S_2]\!]),$$

where $B$ is a Boolean expression and $S_1$ and $S_2$ are imperative statements.

Another way of writing the conditional is in a more two-dimensional form representing the flow of control. For Boolean $B$ and statements $S_1$ and $S_2$,

$$\mathsf{if}\begin{bmatrix} B \rightarrow S_1 \\ \neg B \rightarrow S_2 \end{bmatrix}\mathsf{fi}$$

does the obvious.

Clearly, the conditional satisfies the following equation:

$$\mathsf{if}(B, S_1, S_2) = \mathsf{if}(B, \mathsf{if}(B, S_1, S_2), \mathsf{if}(B, S_1, S_2)).$$

The while-loop, can be defined to be the least fixed-point of a functional,

$$\begin{aligned} [\![\mathsf{while}]\!] &\equiv \mathit{fix}[\lambda g.\lambda B.\lambda S.\mathsf{if}(B, S; g(B, S), I)] \\ &\equiv \bigsqcup \tau^k, \end{aligned}$$

where

$$\begin{aligned} \tau^0 &\equiv \bot \\ \tau^{n+1} &\equiv \lambda B.\lambda S.\mathsf{if}(B, S; \tau^n(B, S), I). \end{aligned}$$

Here, fix is the fixed-point operator and the semicolon denotes composition from left to right.

## Declaration

In dealing with declarations, we will take the simplest approach and assume that we can do $\alpha$ conversion and rewrite each program so that each variable is unique. If a program uses a variable that has not been declared we will consider it to be a syntax error detectable at compile time.

For later examples, however, we consider the semantics of a declaration to consist of two functions. The *declarer* initializes the state so that the variable is bound to a location of the correct type, the *undeclarer* binds the variable to some standard location indicating that it is free.

## Procedures

We consider procedures to be state functions, once their arguments have been given. The parameter passing protocols will depend on the declaration of the parameters. If they are by value, then the function will take the values, if they are by reference, then the appropriate location will be passed.

Adding procedures adds new terms to $V$ in the defining semantic equation for state structures. An example of a new term looks like

$$[N \times \mathsf{Loc} \rightarrow [\Sigma \rightarrow \Sigma]],$$

which denotes the domain of procedures with two arguments, the first is an integer value, the second is by reference.

## 6. Semantics of Hoare Logic

In this section we present an analysis of Hoare Logic axioms and inference rules using continuous states that have been defined. First, the meaning of predicates over states is given inductively.

### Predicates over States

A predicate is a function from states to the truth lattice. It expresses properties about a state primarily be relating the values of different variables. Though no explicit notation is employed, we assume that program variables can be distinguished from the variables of the specification language.

**Definition.** The meaning of a predicate $A(t_1, \ldots, t_n)$, where the $t_i$ are program identifiers, over a state $\sigma$ is defined inductively as follows.

(a) For an atomic formula $A(t_1, \ldots, t_n)$,

$$A(t_1, \ldots, t_n)(\sigma) \equiv [\![A]\!](\sigma^\bullet(t_1), \ldots, \sigma^\bullet(t_n)),$$

where $[\![A]\!]$ is the denotation of $A$. As a special case

$$\ulcorner t_1 = t_2 \urcorner(\sigma^\bullet) \equiv \sigma^\bullet(t_1) \doteq \sigma^\bullet(t_2).$$

(b) For compound formula $A \odot B$, where $\odot$ is any logical connective,

$$(A \odot B)(\sigma) \equiv A(\sigma) \odot B(\sigma).$$

(c) For quantified formula $QxA$, where $Q$ is existential or universal quantifier,

$$(QxA)(\sigma) \equiv Qx(A(\sigma)).$$

**Example.** Let $A \equiv \ulcorner a = 1 \wedge b = 3 \urcorner$. Then

$$A(\sigma) \equiv \ulcorner a = 1 \wedge b = 3 \urcorner(\sigma) = \sigma(a) \dot{=} 1 \mathbin{\dot{\wedge}} \sigma(b) \dot{=} 3.$$

## Interpretation of Hoare Triples

The *continuous interpretation* is one where the Hoare triple is interpreted as an implication with the quantification over all proper states:

$$\{A\}\ P\ \{B\} \equiv \dot{\forall}\, \sigma . A(\sigma) \mathbin{\dot{\Rightarrow}} B(\sigma[\![P]\!]).$$

Under this interpretation, the Empty, Null, Composition, Conditional and Consequence inference rules remain valid, while the others fail in general.

## Fail Rule

Under this interpretation, the fail rule is not valid. We have

$$\begin{aligned}\{A\}\ \mathsf{fail}\ \{B\} &\equiv \dot{\forall}\, \sigma . A(\sigma) \mathbin{\dot{\Rightarrow}} B(\sigma[\![\mathsf{fail}]\!]) \\ &= \dot{\forall}\, \sigma . A(\sigma) \mathbin{\dot{\Rightarrow}} B(\bot),\end{aligned}$$

which is true if $A \equiv \mathsf{false}$ or if $B \equiv \mathbf{true}$. Therefore, the paradoxes that arose by using fail in the body of functions are avoided.

## Assignment Rule

The Assignment axiom has the following form:

$$\{A[x/E]\}\ x := E\ \{A\} \ \equiv\ \dot{\forall}\, \sigma . A[x/E](\sigma) \mathbin{\dot{\Rightarrow}} A(\sigma[\![x := E]\!]).$$

This is valid if either $E$ is proper, or the value of $A$ is independent of $x$, either because $x$ is not free in $A$, or else it occurs in a non-strict context.

## While Rule

(While) $$\frac{\{A \wedge B\}\ P\ \{A\}}{\{A\}\ \mathsf{while}\ B\ \mathsf{do}\ P\ \mathsf{od}\ \{\neg B \wedge A\}}$$

$$[\dot{\forall}\, \sigma . [A \wedge B](\sigma) \mathbin{\dot{\Rightarrow}} A(\sigma)] \ \mathbin{\dot{\Rightarrow}}\ [\dot{\forall}\, \sigma . A(\sigma) \mathbin{\dot{\Rightarrow}} [\neg B \wedge A](\sigma[\![\mathsf{while}\ B\ \mathsf{do}\ P\ \mathsf{od}]\!])]$$

In the case of the While rule, if the condition $B \equiv \mathsf{true}$, then the post condition will be $\mathsf{false}$ and, since no state can satisfy the postcondition, the rule is not valid. When $B$ is not identically true, the validity depends on the computation of the while-loop. Like for the Assignment rule, the postcondition may hold, even if the loop does not terminate.

## 7. Inductive Proofs of the Factorial Programs

For representing programs we shall use Algol 68 syntax and semantics with modifications as desired.

In this section we illustrate the correctness of two programs for computing the factorial, one an applicative version, the other an imperative. We will show the two proofs of correctness to be virtually isomorphic when the state is made explicit for the imperative program.

### Applicative Factorial

The factorial function can be defined recursively as

$$\mathsf{proc}\ \mathit{fact} = (\mathsf{int}\ n)\ \mathsf{int}\colon\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times \mathit{fact}(n-1)\ \mathsf{fi}.$$

A recursive definition specifies a continuous functional over the function space. The fixed point axiom asserts the existence of the least fixed point, a function, also named *fact.* Given the existence of the least fixed point, the definition asserts the equality of the two sides. That equals may be replaced by equals is the key to the simplicity of the proof below.

The proof that the factorial definition is correct requires us to show that the function given is defined for all non-negative integers. This we do by a simple mathematical induction on $n$. The induction predicate is simply

$$P(n) \equiv \mathit{fact}(n) = n!,$$

where $n$ ranges over the nonnegative integers and ! is the mathematical factorial.

First, we show $P(0)$:

$$\begin{aligned} \mathit{fact}(0) &\equiv [\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times \mathit{fact}(n-1)\ \mathsf{fi}](0) \\ &= \mathsf{if}\ 0 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times \mathit{fact}(0-1)\ \mathsf{fi} \\ &= 1 \end{aligned}$$

$$= 0!$$

Next, we show that for $n > 0$, $P(n-1) \Rightarrow P(n)$:

$$\begin{aligned} \mathit{fact}(n) &\equiv [\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \mathit{fact}(n-1) \text{ fi}](n) \\ &= \text{if } n = 0 \text{ then } 1 \text{ else } n \times \mathit{fact}(n-1) \text{ fi} \\ &= n \times \mathit{fact}(n-1) \\ &= n \times (n-1)! \qquad \{\text{by the induction hypothesis}\} \\ &= n! \end{aligned}$$

In the above proof we have used the equational interpretation of the definition and the non-strictness of the if function. Otherwise, we have a very straightforward verification of the factorial definition.

Next we shall verify an imperative version of the same algorithm.

## Imperative Factorial

The definition of the imperative factorial program is

```
proc $fact$ = (int $m$, ref int $f$) :
        if $m = 0$ then $f := 1$ else $fact(m-1, f)$; $f := m \times f$fi.
```

The verification will require that for any nonnegative value of $n$ in a state, after the execution of the program the value of $f$ will be the factorial of the value of $n$. Formally, the predicate is

$$P(n) \equiv n = \sigma(m) \Rightarrow \sigma[\![fact(m,f)]\!](f) = \sigma(m)!,$$

where $m$ and $f$ are program variables.

The proof proceeds as before.

First, show $P(0)$:

$$\begin{aligned} &\sigma[\![\mathit{fact}(m,f)]\!](f) \\ &= \sigma[\![\text{if } m = 0 \text{ then } f := 1 \text{ else } \mathit{fact}(m-1,f); f := m \times f \text{ fi}]\!](f) \\ &= \text{if } \sigma(m) = 0 \text{ then } \sigma[\![f := 1]\!] \\ &\qquad \text{else } \sigma[\![\mathit{fact}(m-1,f); f := m \times f]\!] \text{ fi}(f) \\ &= \sigma[\![f := 1]\!](f) \\ &= 1 \\ &= \sigma(m)!. \end{aligned}$$

Next, show $P(n-1) \Rightarrow P(n)$:

$$\begin{aligned}
&\sigma[\![fact(m,f)]\!](f)\\
&= \sigma[\![\textsf{if } m = 0 \textsf{ then } f := 1 \textsf{ else } fact(m-1,f); f := m \times f \textsf{ fi}]\!](f)\\
&= \textsf{if } \sigma(m) = 0 \textsf{ then } \sigma[\![f := 1]\!]\\
&\qquad\qquad \textsf{else } \sigma[\![fact(m-1,f); f := m \times f]\!] \textsf{ fi}(f)\\
&= \sigma[\![fact(m-1,f); f := m \times f]\!](f)\\
&= \sigma[\![fact(m-1,f)]\!][\![f := m \times f]\!](f)\\
&= \sigma[\![fact(m-1,f)]\!](f) \times \sigma(m)\\
&= (\sigma(m)-1)! \times \sigma(m)\\
&= \sigma(m)!
\end{aligned}$$

Here we have also used the easily verifiable fact that the value of $m$ is unaffected by the factorial program.

## 8. Inductive Proofs of Procedures

The equational interpretation of definitions allows us to treat the representation of programs as if they were the denotations providing us with suitable abstraction at the semantic level. In this section we apply this methodology to procedures and compare it with Martin's proof rules for procedures using weakest predicate transformers (Martin [83]).

### Example 1

Given

$$\ulcorner \textsf{proc } inc1 = (\textsf{int } x, \textsf{ref int } y)\colon y := x + 1 \urcorner,$$

find the weakest $A$ such that

$$\{A\}\ inc1(a,a)\ \{a = a_0 + 1\}.$$

In our method, we want a predicate $A$ such that

$$A(\sigma) \implies \ulcorner a = a_0 + 1 \urcorner(\sigma[\![inc1(a,a)]\!]).$$

That is, we want the weakest precondition such that after the execution of the procedure, $a = a_0 + 1$ holds on the new state. To find $A$, we solve for $\sigma$ on the right-hand side of the implication:

$$\begin{aligned} \ulcorner a = a_0 + 1\urcorner(\sigma[\![\mathit{inc1}(a,a)]\!]) &\iff \ulcorner a = a_0 + 1\urcorner(\sigma[\![a := a+1]\!]) \\ &\iff \sigma[\![a := a+1]\!](a) = \sigma[\![a := a+1]\!](a_0 + 1) \\ &\iff \sigma(a) + 1 = a_0 + 1 \\ &\iff \sigma(a) = a_0, \end{aligned}$$

so we have $A \equiv a = a_0$.

## Example 2

For the procedure *inc1* defined above, find $A$ such that

$$\{A\}\ \mathit{inc1}(a,b)\ \{b = 3 \times a\}.$$

We need to find $A$ such that

$$\begin{aligned} A(\sigma) &\implies \sigma[\![b := a+1]\!](b) = 3 \times \sigma[\![b := a+1]\!](a) \\ &\iff \sigma(a) + 1 = 3 \times \sigma(a) \\ &\iff 1 = 2 \times \sigma(a), \end{aligned}$$

so we have $A \equiv \ulcorner 2 \times a = 1\urcorner$.

## Example 3

Given

$$\ulcorner \textsf{proc}\ \mathit{swap} = (\textsf{ref int}\ x,\ y)\text{: } x, y := y, x\urcorner,$$

determine $A$ such that for any predicate $Q(a,b)$,

$$\{A\}\mathit{swap}(a,b)\{Q(a,b)\}.$$

Let $\hat\sigma \equiv \sigma[\![a, b := b, a]\!]$, then

$$\begin{aligned} Q(a,b)(\hat\sigma) &\iff Q(\hat\sigma(a), \hat\sigma(b)) \\ &\iff Q(\sigma(b), \sigma(a)), \end{aligned}$$

so we have $A \equiv Q(b,a)$.

**Example 4**

Given

$$\ulcorner \mathsf{proc}\ p = (\mathsf{ref\ int}\ z)\colon z := 1\urcorner,$$

show that there is no precondition $A$ such that

$$\{A\}\ p(c)\ \{c = 2\}.$$

Again we try to solve the state equation for the post condition:

$$\begin{aligned} \ulcorner c = 2\urcorner(\sigma[\![c := 1]\!]) &\iff \sigma[\![c := 1]\!](c) = 2 \\ &\iff 1 = 2, \end{aligned}$$

obtaining a contradiction. Therefore, no precondition $A$ can exist.

**Example 5**

Given McCarthy's 91 function in imperative form,

$$\mathsf{proc}\ p = (\mathsf{int}\ x,\ \mathsf{ref\ int}\ y)\colon \{\mathsf{int}\ z;\ \mathsf{if}\begin{bmatrix} x > 100 \to y := x - 10 \\ x \le 100 \to p(x+11, z); p(z, y) \end{bmatrix}\mathsf{fi}\},$$

we want to show that for all values of $\sigma(x)$,

$$(*) \qquad \sigma[\![p(x,y)]\!](y) = \mathsf{if}\begin{bmatrix} \sigma(x) > 100 \to \sigma(x) - 10 \\ \sigma(x) \le 100 \to 91 \end{bmatrix}\mathsf{fi}.$$

Let $d_z \equiv [\![\mathsf{int} z]\!]$, and $d_z^*$ be the undeclarer.

From the definition and the fixed point axiom, $p$ satisfies

$$(**) \qquad \sigma[\![p(x,y)]\!] = \mathsf{if}\begin{bmatrix} \sigma(x) > 100 \to \sigma d_z[\![y := x - 10]\!]d_z^* \\ \sigma(x) \le 100 \to \sigma d_z[\![p(x+11, z)]\!][\![p(z,y)]\!]d_z^* \end{bmatrix}\mathsf{fi}.$$

The proof proceeds by induction on the value of $x$ in $\sigma$. The ordering for the induction looks like:

$$\begin{array}{l} 101 \\ 102 \prec 100 \prec 99 \prec \cdots \prec 1 \prec 0. \\ \vdots \end{array}$$

For $\sigma(x) > 100$ ($\sigma(x) \prec 100$), the base cases, by $(**)$ we have

$$\begin{aligned} \sigma[\![p(x,y)]\!](y) &= \sigma d_z[\![y := x - 10]\!]d_z^*(y) \\ &= \sigma(x) - 10. \end{aligned}$$

Next, assume (∗) holds for $\sigma(x) > n$ ($\sigma(x) \prec n$). For $\sigma(x) = n \leq 100$,

$$\sigma[\![p(x,y)]\!](y)$$

$$= \text{if}\begin{bmatrix} \sigma(x) > 100 \rightarrow \sigma d_z[\![y := x - 10]\!]d_z^*(y) \\ \\ \sigma(x) \leq 100 \rightarrow \sigma d_z[\![p(x+11,z)]\!][\![p(z,y)]\!]d_z^*(y) \end{bmatrix}\text{fi}$$

$$= \sigma d_z[\![p(x+11,z)]\!][\![p(z,y)]\!]d_z^*(y)$$

{let $\hat{\sigma} = \sigma d_z[\![p(x+11,z)]\!]$ from this point on}

$$= \text{if}\begin{bmatrix} \hat{\sigma}(z) > 100 \rightarrow \hat{\sigma}(z) - 10 \\ \\ \hat{\sigma}(z) \leq 100 \rightarrow 91 \end{bmatrix}\text{fi}$$

{by induction hypothesis, since $\sigma(x) + 11 > \sigma(x)$}

$$= \text{if}\begin{bmatrix} \text{if}\begin{bmatrix} \sigma(x)+11 > 100 \rightarrow \sigma(x)+1 \\ \\ \sigma(x)+11 \leq 100 \rightarrow 91 \end{bmatrix}\text{fi} > 100 \rightarrow \hat{\sigma}(z) - 10 \\ \text{if}\begin{bmatrix} \sigma(x)+11 > 100 \rightarrow \sigma(x)+1 \\ \\ \sigma(x)+11 \leq 100 \rightarrow 91 \end{bmatrix}\text{fi} \leq 100 \rightarrow 91 \end{bmatrix}\text{fi}$$

$$= \text{if}\begin{bmatrix} \sigma(x)+11 > 100 \wedge \sigma(x)+1 > 100 \rightarrow \hat{\sigma}(z) - 10 \\ \sigma(x)+11 > 100 \wedge \sigma(x)+1 \leq 100 \rightarrow 91 \\ \sigma(x)+11 \leq 100 \wedge 91 > 100 \rightarrow \hat{\sigma}(z) - 10 \\ \sigma(x)+11 \leq 100 \wedge 91 \leq 100 \rightarrow 91 \end{bmatrix}\text{fi}$$

$$= 91.$$

The only nontrivial case is the first condition of the last if. From our assumption, $\sigma(x) \leq 100$, and from the condition of the if, $\sigma(x) + 1 > 100$, the only possible value for $\sigma(x)$ is 100. For that value, $\hat{\sigma}(z) - 10 = 91$, and so we are done.

## 9. Summary

Denotational semantics allows us to treat definitions as equations. This principle was used to prove properties of programs in an equational way.

The logics of programs like Hoare Logic implicitly deal with states. What we have done is made that reference explicit. The appeal of Hoare-like Logics is that they hide unnecessary details about programs, especially about the state. This is fine for simple while-programs without pointers and functions, but as the complexity of the language increases and the parameter passing protocols for procedures gets more involved, it becomes necessary to make more and more of the state explicit.

Sokolowski [84] introduces terms as the pre- and post-conditions, with the interpretation that the value of the pre-condition term equals the value of the post-condition term after the execution of the program.

Sieber [85] uses a function "cont" which returns the value of the variable at a certain state. This is the same as our use of the value at a state, $\sigma^{\bullet}$. However, "cont" assumes a simple model of the state where all variables are of level 1.

Tennent [85] uses notions form categorial logic and Kripke semantics to provide a model for Reynolds' specification logic. The crucial difference in his approach is on dealing with false as the bottom element of an Heyting algebra.

As the expressiveness of programming languages increases, it seems inevitable that purely syntactic characterization of program behavior will become more difficult and semantic properties must be used directly.

Chapter 5

# Infinite Lists

*Wherein a semantic domain for infinite lists of arbitrary high order is constructed and meaning given to non-terminating programs, thereby enabling equational reasoning about them.*

## 1. Introduction

Infinite lists arise when we want to give meaning to non-terminating, yet answer-producing, programs. In applicative languages lazy evaluation enables infinite objects to be progressively computed. Precise formulation of infinite lists is necessary for formal reasoning of such programs.

As a start, we might define infinite lists by specifying each element of the list, by, say, a function from the set of natural numbers to the set of elements. Unfortunately, this method is inadequate if we want lists whose elements are again infinite lists. Next, consider defining lists by levels. First, we have infinite lists of order one, the elements of which are atoms. Next, the infinite lists of order $n+1$ are obtained by allowing infinite lists of order $n$ or less as elements. Even this construction does not capture all definable infinite lists.

Consider an infinite list whose first element is an infinite list of order one, second element is a list of order two, etc. Clearly, this list is not of any finite order. We shall say a list like this has order $\omega$. The interesting point here is that such lists can actually be generated in a programming language that allows lazy evaluation. Once we have a list of order $\omega$, we can use it to define a list of order $\omega \cdot 2$ and so on (up to $\epsilon_0$).

Infinite objects can be computed only as a limit of finite ones. The notion of a limit presupposes some kind of a topology, or at least an ordering. The ordering we use is that of definedness. By generating a sequence of finite lists that become more and more defined, we can specify an infinite list.

We construct a domain of infinite lists by taking the projective limit of the chain of finite lists ordered by projection and indicate how the meaning of non-terminating programs can be defined. We show that set-theoretic constructions cannot generate all definable lists by appealing to the Axiom of Foundation.

## 2. Finite Lists

Given a set of atoms $A$, make it into a flat semilattice by the usual technique of adjoining a bottom element, $\perp_A$, which is less than all the defined elements of $A$. We indicate the empty list by $\diamond$, and the undefined list by $\perp$.

**Definition.** The *finite lists* of length $n$, $L_n$, are defined inductively:

$$\begin{aligned} L_0 &\equiv \{\diamond, \perp\} \\ L_1 &\equiv (A \cup L_0) \times L_0 \;\cup\; L_0 \\ L_{n+1} &\equiv (A \cup L_n) \times L_n \;\cup\; L_n. \end{aligned}$$

The set of all *finite lists*, $F$, is the union of all the $L_n$'s. The pair $(a, l)$ is written $a\!:\!l$ using the pairing operator, which is assumed to be right associative. For example, $a\!:\!b\!:\!c = a\!:\!(b\!:\!c)$.

The ordering on $F$, and therefore for each $L_n$, is defined as follows:

(1) $\perp \sqsubseteq x$ for all $x \in F$,

(2) $x\!:\!y \sqsubseteq u\!:\!v$ if $x \sqsubseteq u$ and $y \sqsubseteq v$.

In other words, an element of $L_{n+1}$ is either an element of $L_n$, or a pair of elements where the first component is either an atom or an element of $L_n$ and the second component is an element of $L_n$.

Next, we define the projection mapping from $L_{n+1}$ to $L_n$. This mapping acts as the identity on the lists of $L_{n+1}$ that already belong to $L_n$, while for the other lists its value is the list in $L_n$ that best approximates the argument.

**Definition.** The *projection functions*, $\psi_n : L_{n+1} \to L_n$, are defined inductively:

$$\psi_0(x) \equiv \begin{cases} \diamond & \text{if } x = \diamond; \\ \bot & \text{otherwise.} \end{cases}$$
$$\psi_n(x:y) \equiv \begin{cases} x:y & \text{if } x:y \in L_n; \\ x:\psi_{n-1}(y) & \text{if } x \in A; \\ \psi_{n-1}(x):\psi_{n-1}(y) & \text{otherwise.} \end{cases}$$

**Examples.** $L_2$ contains $\bot$, $\diamond$, $1:\diamond$, $(\diamond):\bot$, $1:\bot$, $1:2:\bot$, $(1:\bot):(\bot):\bot$, $\bot_A:\bot$, etc. $L_3$ contains all the elements of $L_2$ as well as $1:2:3:\bot, (1:2:\bot):(1:\diamond):(\diamond):\bot$, etc. Examples of the ordering and projection functions:

$$\bot \sqsubseteq \diamond, \quad \bot_A:\bot \sqsubseteq 1:\bot \sqsubseteq 1:2:\bot, \quad (1:\bot):\diamond \sqsubseteq (1:\diamond):\diamond,$$

$$\psi_2(1:2:\diamond) = 1:2:\diamond, \quad \psi_2(\,(1:2:\bot):(1:\bot):(\bot):\bot\,) = (1:\bot):(\bot):\bot.$$

## 3. Infinite Lists

Infinite lists will be constructed as sequences of finite lists where the $n$th element comes from $L_n$. Each element of the sequence will be the $\psi$-projection of the next. This is the consistency condition necessary for the projective limit construction.

**Definition.** The set of *infinite lists*, $L_\infty$, is the limit of the projective spectrum $\{L_n; \psi_n\}$, and can be represented as

$$L_\infty \equiv \{\langle s_n\rangle_{n=0}^{\infty} \mid \text{ for all } n,\ s_n \in L_n \text{ and } s_n = \psi_n(s_{n+1})\}.$$

There are projections and injections to and from each $L_n$ and $L_\infty$.

**Definition.** For each $n$, the *projection function* $\psi_{\infty n} : L_\infty \to L_n$ is

$$\psi_{\infty n}(l) \equiv l_n, \quad \text{where} \quad l \equiv \langle l_n\rangle_{n=0}^{\infty}.$$

**Notation.** Let $\psi_{nn}$ be the identity on $L_n$. For $n \geq m$, let

$$\psi_{nm} \equiv \psi_m \circ \cdots \circ \psi_{n-1} \circ \psi_n,$$

the composition of the projection functions.

**Definition.** For each $n$, the *injection function* $\phi_{n\infty} : L_n \rightarrow L_\infty$ is

$$\phi_{n\infty}(l_n) \equiv \langle \psi_{n0}(l_n), \psi_{n1}(l_n), \psi_{n2}(l_n), \ldots, \psi_{nn-1}(l_n), l_n, l_n, \ldots \rangle.$$

Next, make $L_\infty$ into a partial order by defining the order coordinatewise on the sequence.

**Definition.** Let $l_1 \equiv \langle s_n \rangle$ and $l_2 \equiv \langle t_n \rangle$ be infinite lists. The ordering on infinite lists is defined by:

$$l_1 \sqsubseteq l_2 \qquad \text{iff} \qquad \text{for all } n,\ s_n \sqsubseteq t_n.$$

## 4. Complete Semilattices

Once infinite lists are given with a partial ordering, the next step is to look at a chain of lists. We want each chain to have a supremum, which is again an infinite list, that is, a member of $L_\infty$.

**Definition.** A set of lists $\{l_i\}_{i=0}^{\infty}$ is called a *chain* if $l_i \sqsubseteq l_{i+1}$ for all $i$.

In order to show that $L_\infty$ is a complete semilattice, we first prove that each $L_n$ is.

**Lemma.** *Each $L_n$ is a complete semilattice.*

*Proof:* We show that the least upper bound exists for every chain by showing that a chain in $L_n$ can have only a finite number of distinct elements, and therefore, the maximum of the chain is the least upper bound.

We define the *rank* of a list to be the total number of occurrences of the pairing operator (:), the empty list ($\Diamond$), and defined elements of $A$. We claim that

(a) For each $L_n$ the maximum rank is bounded, and

(b) If $l_1$ is *strictly* less defined than $l_2$, then the rank of $l_1$ is less than the rank of $l_2$.

(a) is proved by induction on the rank of lists. The max of the rank in $L_0$ is 1. A list in $L_n$ consists of a finite number of lists of lower order. (b) is true because the only way to make a list strictly more defined is by either replacing $\perp$ by $\diamond$ or $x:\perp$ for some $x$, or replacing $\perp_A$ by a defined element of $A$, all of which increases the rank by one. Therefore, since the rank is bounded, every chain must be finite. $\square$

Next, we prove the main theorem that $L_\infty$ is a complete semilattice by explicitly constructing a list that is the supremum of a chain, and showing that it is a member of $L_\infty$.

**Theorem.** *$L_\infty$ is a complete semilattice.*

*Proof:* Let $\{l_i\}$ be a chain with $l_i \equiv \langle l_{ij}\rangle_{j=0}^{\infty}$. Construct a new list as follows:

$$l \equiv \langle s_n\rangle_{n=0}^{\infty} \qquad \text{where} \qquad s_n \equiv \bigsqcup_{i=0}^{\infty} l_{in}.$$

From the previous Lemma, the supremum, $s_n$, exists for each $n$, so $l$ is well defined. By definition, each $s_n$ is in $L_n$. To show that $l$ belongs to $L_\infty$, we need to show that $s_n = \psi_n(s_{n+1})$, for all $n$. This is shown by:

$$\begin{aligned}\psi_n(s_{n+1}) &\equiv \psi_n(\bigsqcup_{i=0}^{\infty} l_{i\,n+1}) \\ &= \bigsqcup_{i=0}^{\infty} \psi_n(l_{i\,n+1}) \\ &= \bigsqcup_{i=0}^{\infty} l_{in} \\ &\equiv s_n.\end{aligned}$$

The first, third and last equalities are by definitions. The second equality follows from the monotonicity of $\psi_n$ and the fact that every chain is finite.

So, $l$ is an upper bound since each component is the supremum, and it is the least because it is the least element componentwise. $\square$

## 5. Application to Program Semantics

With $L_\infty$ proved to be a complete semilattice, we can use it for the fixed point approach to program semantics. The following programs are defined using an applicative language with lazy evaluation like Turner's *KRC* [82].

## Example 1

A program to generate an infinite list of 1's can be defined by:

$$f = \tau[f] \quad \text{where} \quad \tau[f] \equiv 1 : f.$$

The meaning of $f$ is the supremum of the partial lists defined by the $\tau_i$'s:

$$[\![f]\!] \equiv \bigsqcup_{i=0}^{\infty} \tau_i,$$

where the $\tau_i$'s are given inductively by:

$$\tau_0 \equiv \bot \quad \text{and} \quad \tau_{n+1} \equiv \tau[\tau_n] = 1 : \tau_n.$$

By considering a finite list $\tau_i$ to be both a member of $L_i$ as well as its corresponding injection in $L_\infty$, the first few terms can be given by:

$$\begin{aligned}
&\tau_0 \equiv \bot \\
&\tau_1 \equiv \tau[\tau_0] = 1 : \tau_0 = 1 : \bot \\
&\tau_2 \equiv \tau[\tau_1] = 1 : \tau_1 = 1 : 1 : \bot \\
&\quad\vdots \\
&\tau_i \equiv \tau[\tau_{i-1}] = 1 : \tau_{i-1} = 1 : \cdots : 1 : \bot
\end{aligned}$$

and therefore

$$[\![f]\!] = \langle \tau_i \rangle_{i=0}^{\infty},$$

which is the object representing an infinite list of 1's.

## Example 2

Recall that a list whose $n$th element is a list of order $n$ is a list of order $\omega$. A program to generate a list of order $\omega$ can be defined in two stages. First, define a program that generates lists of all finite order by using a parameter:

$$\begin{aligned}
f0 &= 1 : f0 \\
f(n+1) &= fn : f(n+1).
\end{aligned}$$

Next, we diagonalize to get higher-order elements:

$$Fn = \tau[F]n \quad \text{where} \quad \tau[F]n \equiv fn : F(n+1).$$

If we write $\sigma^n \equiv [\![fn]\!]$ for each $n$, then

$$\begin{aligned}\sigma^0 &= 1 : 1 : 1 : \cdots, \\ \sigma^1 &= \sigma^0 : \sigma^0 : \sigma^0 : \cdots, \\ \sigma^2 &= \sigma^1 : \sigma^1 : \sigma^1 : \cdots, \\ &\vdots,\end{aligned}$$

and

$$\begin{aligned}[\![F0]\!] &= [\![f0]\!] : [\![f1]\!] : [\![f2]\!] : \cdots \\ &= \sigma^0 : \sigma^1 : \sigma^2 : \cdots.\end{aligned}$$

Though the meaning of $F0$ is precisely given, this is not quite satisfactory since, in practice, an infinite list, $\sigma^0$, cannot be completely given before giving $\sigma^1$. What we want is to produce finite approximations of all infinite lists so that eventually each infinite list will be given to an arbitrary level of accuracy. To do this we need projection functions from infinite lists to their finite approximations.

Then the meaning of $F0$ can be given as:

$$[\![F0]\!] = \bigsqcup_{i=0}^{\infty} \tau_i 0,$$

where the finite lists are defined inductively by:

$$\tau_0 k = \bot \quad \text{and} \quad \tau_{n+1} k \equiv \tau[\tau_n] k = \psi_{\infty n}(\sigma^k) : \tau_n(k+1) \quad \text{for all } k.$$

More explicitly:

$$\begin{aligned}&\tau_0 0 = \bot \\ &\tau_1 0 = \tau[\tau_0]0 = \psi_{\infty 0}(\sigma^0) : \tau_0 1 = \sigma_0^0 : \bot = (\bot) : \bot \\ &\tau_2 0 = \tau[\tau_1]0 = \psi_{\infty 1}(\sigma^0) : \tau_1 1 = \psi_{\infty 1}(\sigma^0) : \psi_{\infty 0}(\sigma^1) : \tau_0 2 = \sigma_1^0 : \sigma_0^1 : \bot \\ &\quad\vdots \\ &\tau_n 0 = \psi_{\infty n-1}(\sigma^0) : \psi_{\infty n-2}(\sigma^1) : \cdots : \psi_{\infty 0}(\sigma^{n-1}) : \tau_0 n.\end{aligned}$$

The projection functions ensure that each $\tau_n 0$ is an element of $L_n$. At each stage, not only does the length of the list grow, but each component of the list becomes more and more defined and of higher order, so that in the limit we have a list of order $\omega$.

Clearly, this process of diagonalization can be iterated an arbitrary number of times to produce lists of higher order.

## Example 3

Let *fib*$(n)$ be the $n$th Fibonacci number. A program to generate all the Fibonacci numbers can be defined by

$$f\,a\,b = a : f\,b\,(a+b).$$

Since we have defined a semantic domain of infinite lists, we can use the principle that program definitions are equations, to show that $f\,1\,1$ generates the infinite sequences of Fibonacci numbers.

$$f\,1\,1 = 1{:}f\,1\,2 = 1{:}1{:}f\,2\,3 = 1{:}1{:}2{:}f\,3\,5 = \cdots$$

## 6. Comparison with Ordinal-Hierarchic Construction

In this section we explore an ordinal-hierarchic construction of infinite lists recursively over the transfinite ordinals and show why this method cannot produce all the lists that are definable in a programming language with lazy evaluation.

**Definition.** For any set $A$, let $L(A)$ be the set of all finite and (countably) infinite lists whose elements are from $A$:

$$L(A) \equiv A^{[\omega]} \cup \bigcup_{n=0}^{\omega} A^{[n]},$$

where $A^B$ denotes the set of all set-theoretic functions from $B$ to $A$, $[n]$ the set of natural numbers less than $n$, and $\omega$ the first infinite ordinal.

Next, we construct a hierarchy of sets over the ordinals of lists, where at each stage we use the lists in the previous construction as elements of the new lists.

**Definition.** The set of infinite lists over the set of atoms $A$ is

$$\begin{aligned} K_0 &\equiv L(A) \\ K_{n+1} &\equiv K_n \cup L(K_n) \\ K_\omega &\equiv \bigcup_{\alpha<\omega} K_\alpha \\ K_{\omega+1} &\equiv K_\omega \cup L(K_\omega) \\ &\vdots \end{aligned}$$

where $\omega$ is a limit ordinal.

How do these sets of lists compare with the lists in the projective limit $L_\infty$? Did we really need to use the projective limit construction? It turns out that no matter how high we go up the ordinal hierarchy, we do not exhaust the lists in $L_\infty$. Furthermore, there is a list in $L_\infty$ that is not in any of the $K_\alpha$'s.

Since $K_\alpha$'s contain functions with finite and infinite domains, while $L_\infty$ contains infinite sequences of finite (partial) lists, a direct comparison is not possible. Instead, we show that there exists an injection from one to the other where the injection associates an infinite ordinal-hierarchic list with an infinite sequences of finite lists which approaches it in the limit.

**Theorem.** *For all ordinal $\alpha$, there is an injection $\iota_\alpha : K_\alpha \to L_\infty$.*

*Proof:* The proof is by induction on the $\alpha$'s. Let the $n$th element of $l$ be $l_n$.

For $\alpha = 0$, let $l \in K_0 \equiv L(A)$, and define

$$\iota_0(l) \equiv \langle \bot, l_0 : \bot, l_0 : l_1 : \bot, l_0 : l_1 : l_2 : \bot, \dots \rangle$$

if $l$ is an infinite list, and

$$\iota_0(l) \equiv \langle \bot, l_0 : \bot, l_0 : l_1 : \bot, \dots, l_0 : \cdots : l_{k-1} : \diamond, l_0 : \cdots : l_{k-1} : \diamond, \dots \rangle ,$$

if $l$ is a list of length $k$. To show that $\iota_0$ is injective we observe that if two lists are different, then they will differ at some $k$th element, and they will define infinite sequences differing at the $k+1$st place.

For the successor ordinal $\alpha + 1$, assume there exists an injection

$$\iota_\alpha : K_\alpha \to L_\infty,$$

and let $l$ be in $K_{\alpha+1}$. Define

$$\iota_{\alpha+1}(l) \equiv \langle \bot, \psi_{\infty 1}(\iota_\alpha(l_0) : \bot), \psi_{\infty 2}(\iota_\alpha(l_0) : \iota_\alpha(l_1) : \bot), \\ \psi_{\infty 3}(\iota_\alpha(l_0) : \iota_\alpha(l_1) : \iota_\alpha(l_2) : \bot), \dots \rangle,$$

where the $\psi_{\infty n}$'s are the projection functions from $L_\infty$ to $L_n$, defined previously.

If two lists $l$ and $l'$ in $K_{\alpha+1}$ differ, then for some $n$, $l_n$ differs from $l'_n$. By induction hypothesis, $\iota_\alpha$ is injective, so $\psi_{\infty n+1}(\iota_\alpha(l_n) : \cdots : \bot)$ will differ from $\psi_{\infty n+1}(\iota_\alpha(l'_n) : \cdots : \bot)$. Therefore, $\iota_{\alpha+1}$ is injective.

For $\beta$, a limit ordinal, let $l$ be in $K_\beta$. By definition, each $l_n$ is in $K_{\alpha_n}$, for some ordinal $\alpha_n$ less than $\beta$. In this case, we define

$$\iota_\beta(l) \equiv \langle \bot, \psi_{\infty 1}(\iota_{\alpha_0}(l_0) : \bot), \psi_{\infty 2}(\iota_{\alpha_0}(l_0) : \iota_{\alpha_1}(l_1) : \bot), \\ \psi_{\infty 3}(\iota_{\alpha_0}(l_0) : \iota_{\alpha_1}(l_1) : \iota_{\alpha_2}(l_2) : \bot), \dots \rangle.$$

Again, by induction hypothesis and the same reasoning as for the successor case, $\iota_\beta$ is injective. $\square$

The next theorem shows that the projective limit captures lists that are not well-founded set-theoretically.

**Theorem.** *There exists a list in $L_\infty$ not in any $K_\alpha$.*

*Proof:* Let $a$ be a list satisfying the following program definition:

$$a = a : \diamond \ .$$

We claim that this list has no corresponding element in any $K_\alpha$. Its representation in $L_\infty$ looks like:

$$a = \langle \bot, \bot : \diamond, (\bot : \diamond) : \diamond, ((\bot : \diamond) : \diamond) : \diamond, \ldots \rangle.$$

This means that if there were a list $l \in K_\alpha$ for some $\alpha$, then the first element of $l$, $l_0$, is a list isomorphic to $l$, which means the first element of $l_0$, $l_{00}$, is again isomorphic to $l$, and this must continue without bound. This violates the Axiom of Foundation (see Enderton [77]),

$$A \neq \emptyset \Rightarrow \exists x (x \in A \wedge x \cap A = \emptyset),$$

a consequence of which says that there cannot be an infinite descending chain of sets that are elements of the previous set. Therefore, no set theoretic list maps onto $a$. $\square$

In programming languages with lazy evaluation, and certainly in the $\lambda$-calculus, infinite lists can be defined, and approximations to an arbitrary degree can be obtained as long as one is willing to wait long enough.

## 7. Summary

In order to construct $\omega$ and higher-order infinite lists, we used the projective limit construction technique from topology. This seems to be a very general technique for going from the finite to the infinite. Our construction of the $L_n$'s was chosen because of its simplicity, and so may be less intuitive than other choices, although they all lead to isomorphic limits. The only nontrivial part of this chapter is in showing that the least upper bound belongs to the constructed domain of infinite lists. Here we used the fact that for finite chains a monotonic function can be moved inside the limit.

de Bakker and Zucker [83] modeled concurrent processes using infinite trees that were constructed by first defining a distance metric between finite trees, and then, by using the standard completion technique, obtaining the infinite trees as limits of Cauchy sequences of finite ones. By using the projective limit, we were able to construct the infinite lists directly without introducing an arbitrary metric.

The projective limit construction was used by Scott [71] to model the type-free $\lambda$-calculus. Infinite lists can be represented as certain terms in the $\lambda$-calculus, and therefore, from a theoretical point of view, the standard semantics using $D_\infty$ is sufficient. From a practical and pedagogical point of view, however, a more direct construction using finite lists gives clearer motivation.

Chapter 6

# Objects and Multiple Inheritance

*Wherein a semantic domain for objects that can refer to themselves is constructed, and multiple inheritance defined as operations over objects.*

With object-oriented languages, the reification of implicit structures continues. States and environments are now made first class citizens and henceforth renamed "objects". Objects make up the fundamental stuff of computation by being manipulable, and they can be passed to and returned from functions. Objects are created through "classes," which define the type of an object. One of the principles of object-oriented languages is that everything is an object. Also, every object can refer to itself. This circularity and self-reference makes the semantics of objects and classes non-trivial.

This chapter begins with an informal introduction to objects and classes based on the language Smalltalk (Goldberg and Robson [83], Kay [69]). Then we construct a semantic domain of objects and classes using the projective limit construction. The projective limit construction is needed to deal with the self-referential capability of each object.

The power of object-oriented languages derives from the abstraction mechanism of classes and the factoring out of common descriptions using inheritance. The semantics of multiple inheritance is given by introducing the notion of the product of two classes.

## 1. Objects and Classes

Informally, as exemplified in a language like Smalltalk, an object-oriented language is made up of the following notions.

(1) An *object* consists of some local memory and a set of operations called *methods*. Methods are procedures local to an object.

(2) A *message* is a request for an object to carry out one of its operations. The set of messages to which an object can respond is called its *interface*.

(3) A *class* describes the implementation of a set of objects that all represent the same kind of system component. The individual objects described by a class are called its *instances*.

(4) A mechanism whereby a new class definition is given using another class definition, is known as *inheritance*. The new class is called the *subclass* of the older class, while the older class is called its *superclass*.

The messages in the interface are public, while the local memory and the methods are private to each object.

In addition there are two guiding principles that underlie the design and implementation of a language like Smalltalk.

(A) Every object is an instance of some class.

(B) Classes are represented by objects.

In order to abide by these two principles, Smalltalk also has the notion of *metaclass*, instances of which are classes. Of course, metaclasses themselves have to be instances of some class, and so we have a circularity in the subclass and instance hierarchies.

In this chapter, we shall present a very simplified version, exhibiting only the essentials of classes and objects, handling the circularity by allowing self-application.

## 2. Semantic Domain of Objects

Objects will be modeled as state with self-reference; i.e., there is an identifier self in the domain of the state whose value in that state is isomorphic to itself. In general, objects will have identifiers pointing to other objects, as well as local procedures, making their semantic equation somewhat more involved than for the states of an imperative language.

**Notation.** In this chapter, “object” will have the meaning used in object-oriented programming languages, and not the general meaning of “entity” or “element.”

Let $\mathsf{Ide}$ be a set of identifiers containing $\mathsf{self}$, $\mathsf{Loc}$ be a set of locations, and $V$ be a sum of all the other domains of data types and functions over them.

Intuitively, what we want is a domain satisfying

$$O \cong [\mathsf{Ide} + \mathsf{Loc} \longrightarrow \mathsf{Loc} + V + O + [O \to O]],$$

such that for each $o$ in $O$,

$$o(\mathsf{self}) \cong o,$$

and, for all $x_1, \ldots, x_k$ in $\mathsf{Ide}$, if $o(x_1)\ldots(x_k)$ is an object, then

$$o(x_1)\ldots(x_k)(\mathsf{self}) \cong o(x_1)\ldots(x_k).$$

In other words, each object has an identifier, called self, that points to something isomorphic to itself; and, if there are any internal objects, then they are all that way too.

## Domains of Partial Objects

The way we construct such a domain is through the projective limit of domains of partial objects with projection functions. Because of extra conditions that need to be imposed, we first define intermediate domains of pre-objects.

**Definition.** The *domains of partial pre-objects* are defined inductively.

(a) For the base case:

$$P_0 \equiv [\mathsf{Ide} + \mathsf{Loc} \longrightarrow \mathsf{Loc} + V],$$

where for each $p$ in $P_0$, $p(\mathsf{self}) \equiv \perp_0$, the undefined element of the codomain.

(b) For the induction case, given $P_n$:

$$P_{n+1} \equiv [\mathsf{Ide} + \mathsf{Loc} \longrightarrow \mathsf{Loc} + V + P_n + [P_n \to P_n]].$$

Connecting the domains are the projections, whose value at a partial object is the best approximation in the domain below.

**Definition.** Let us call the elements of $\mathsf{Ide} + \mathsf{Loc}$ that map into $P_n$ or $[P_n \to P_n]$, the *object variables*. The projection functions, $\psi_n : P_{n+1} \to P_n$, are defined by:

(a) $\psi_0(p_1) \equiv p_0$, where $p_0$ agrees with $p_1$, except that for all object variables $x$,

$$p_0(x) \equiv \perp_0.$$

(b) $\psi_{n+1}(p_{n+2}) \equiv p_{n+1}$, where $p_{n+1}$ agrees with $p_{n+2}$, except that for all object variables $x$,

$$p_{n+1}(x) \equiv \psi_n(p_{n+2}(x)).$$

Essentially, what the $\psi_n$'s do is prune the leaves and replace them with undefined, if the leaves are $P_0$ objects. This is the best approximation in the next lower domain.

## Domain of Objects

The limit of the projective spectrum, $\underleftarrow{\mathrm{Lim}}\{P_n; \psi_n\}$, contains many elements that do not correspond to any objects. What we need is to restrict the eligible elements at each stage to those that are as self-referential as possible.

**Definition.** The *domains of partial objects*, $O_n$, are constructed inductively as follows:

(a) For the base case:

$$O_0 \equiv [\mathsf{Ide} + \mathsf{Loc} \longrightarrow \mathsf{Loc} + V],$$

where for each $o$ in $O_0$, $o(\mathsf{self}) = \bot$, the undefined element of the codomain.

(b) For the induction case, given $O_n$:

$$O_{n+1} \equiv \{o \in [\mathsf{Ide} + \mathsf{Loc} \longrightarrow \mathsf{Loc} + V + O_n + [O_n \rightarrow O_n]] \mid o(\mathsf{self}) = \psi_n(o)\}.$$

(c) The projections, $\psi_n$, are the restrictions of $\psi_n : P_{n+1} \rightarrow P_n$ to $O_{n+1}$'s.

**Definition.** The *domain of objects* is the projective limit of the projective spectrum $\{O_n; \psi_n\}$:

$$O_\infty \equiv \{\langle o_n \rangle_{n=0}^{\infty} \mid \text{for all n, } o_n \in O_n, o_n = \psi_n(o_{n+1}) \text{ and } o_{n+1}(\mathsf{self}) = o_n)\}.$$

**Example.** Using the notation for state structures, an object with one local variable $i$ (set to 5) looks like:

$$o \equiv \langle o_n \rangle_{n=0}^{\infty},$$

where

$$o_0 \equiv \begin{bmatrix} i \mapsto l_i \mapsto 5 \\ \mathsf{self} \mapsto \bot \end{bmatrix} \text{ and } o_{n+1} \equiv \begin{bmatrix} i \mapsto l_i \mapsto 5 \\ \mathsf{self} \mapsto o_n \end{bmatrix}.$$

**Notation.** Since all objects have the identifier self bound to its $\psi$-projection, we shall omit displaying the binding of self and picture the above object simply by

$$[\, i \mapsto l_i \mapsto 5 \,] \,.$$

## 3. Semantics of Classes

Classes are objects that return an instance of that class when they are sent the message new. Since classes themselves are objects, they are created by sending to a metaclass object, called "Class," a declaration of that class.

Declarations specify the type and value of each of the components of a class in an intensional way. We present an informal account of declarations, adopting the Algol 68 notation with a few modifications.

### Declarations

Simple declarations bind identifiers to internal structures. For example,

$$\ulcorner \mathsf{real}\ x = \mathsf{real\ loc} \urcorner$$

binds the identifier $x$ to an internal location for a real number in the current state. This is usually shortened to just $\ulcorner \mathsf{real}\ x \urcorner$.

Usually, a declaration also initializes the variable, as in $\ulcorner \mathsf{int}\ i := 5 \urcorner$. This will be considered syntactic sugar for $\ulcorner \mathsf{int}\ i = \mathsf{int\ loc};\ i := 5 \urcorner$.

Declarations also bind identifiers to more complex entities such as composite data structures, procedures, and functions. For example,

$$\ulcorner \mathsf{proc}\ p = (\mathsf{int}\ n)\ \mathsf{int}\colon\ n \times n \urcorner$$

binds $p$ to the integer squaring function.

In addition to all the standard data types of a language like Algol 68, we introduce two new types, **method** and **object**. The **method** type is to distinguish procedures in the interface from the private procedures, and the **object** is for declaration of internal objects.

**Definition.** A *declaration* is a pair $d \equiv (l, e)$, usually written $\ulcorner l = e \urcorner$, where $l$ is an identifier declaration and $e$ is an expression of the appropriate type.

**Definition.** An *object expression* is a set of declarations, written $[d_1, \ldots, d_k]$, where the $d_i$'s are declarations. The internal variable self is always declared to be pointing to itself and so need not be specified.

Intentionally, we are using the square brackets both for displaying objects and for defining them by object expressions. There can be no confusion, since a displayed object will always have a binding of identifier to some object, while object expressions contain declarations.

**Example.** The following is an object with three components, a variable $i$ that is local, the method *add*, and an object *point* that has two real numbers as components.

```
[ int $i := 5$,
  method $add$ = (int $n$) int: $i := i + 1$,
  object $point$ = [ real $x$, real $y$]
]
```

Given an object declaration, we need a function that will produce the object.

**Definition.** Let Dec be the set of declarations. The function $obj : \mathsf{Dec} \to O_\infty$ is defined inductively on the construction of the declarations. Let $\rho$ denote the standard environment.

(a) For simple declaration, $d \equiv (l, e)$, not an object declaration,

$$obj([d])\rho \equiv [x \mapsto \cdots \mapsto [\![e]\!]\rho],$$

where $x$ is the identifier in $l$, the number of internal locations in the dots depends on the type of $x$, and the value is the value of $e$ in the current environment.

(b) For multiple declarations, $d \equiv [d_1, \ldots, d_k]$, where all the identifiers are assumed distinct,

$$obj([d_1, \ldots, d_k]) \equiv obj([d_1]) \sqcup \cdots \sqcup obj([d_k]),$$

the lattice join in $O_\infty$.

(c) For object declaration, $d \equiv \ulcorner \mathsf{object}\ o = [d_1(o), \ldots, d_k(o)]\urcorner$, where $d_i(o)$ may contain the identifier $o$ in a recursive definition. Let $\vec{d}(o)$ denote $[d_1(o), \ldots, d_k(o)]$. What we want is the solution to the following equation:

$$obj(d)\rho = [o \mapsto obj(\vec{d}(o))\rho[o : obj(d)\rho(o)]\ ].$$

Like any other recursive definition, the solution is obtained as the limit of finite approximations:

$$obj(d)\rho \equiv \langle p_n \rangle_{n=0}^{\infty} n,$$

where

$$\begin{aligned} p_0 &\equiv [o \mapsto \bot] \\ p_{n+1} &\equiv [o \mapsto \psi_{\infty n}(obj(\vec{d}(o))\rho[o : p_n])\ ]. \end{aligned}$$

For the multiple declaration, since all the identifiers are distinct, the only non-trivial thing to check is whether self of the join of several objects is the join of several self's.

**Proposition.** *Let $[d_1, \ldots, d_k]$ be a multiple declaration. Then if each $obj([d_i])$ is an object, then so is $obj([d_1, \ldots, d_k]) \equiv obj([d_1]) \sqcup \cdots \sqcup obj([d_k])$.*

*Proof:* Without loss of generality, we prove the proposition for $k = 2$. To show that the join of two objects is again an object, we need to prove that the join satisfies the consistency condition and that self is well behaved.

The consistency condition is trivial, since $\psi_n$ are continuous.

Next, let $o \equiv obj([d_1])$ and $p \equiv obj([d_2])$. Then

$$o \equiv \langle o_n \rangle_{n=0}^{\infty} \quad \text{and} \quad p \equiv \langle p_n \rangle_{n=0}^{\infty},$$

where at each $n$

$$o_n = \psi_n(o_{n+1}) \quad \text{and} \quad p_n = \psi_n(p_{n+1}).$$

and

$$o_{n+1}(\mathsf{self}) = o_n \quad \text{and} \quad p_{n+1}(\mathsf{self}) = p_n.$$

We need to show that for each $n$

$$[o_n \sqcup p_n](\mathsf{self}) = o_n(\mathsf{self}) \sqcup p_n(\mathsf{self}).$$

For $n = 0$,

$$[o_0 \sqcup p_0](\mathsf{self}) = \bot = \bot \sqcup \bot = o_0(\mathsf{self}) \sqcup p_0(\mathsf{self}).$$

Assume $n$,

$$[o_{n+1} \sqcup p_{n+1}](\mathsf{self}) = o_{n+1}(\mathsf{self}) \sqcup p_{n+1}(\mathsf{self}) = o_n \sqcup p_n. \quad \Box$$

In this section we defined object expressions that define objects and a function that returns the appropriate object as its value. The object expression can be very general, allowing for recursive definitions of objects.

## Classes

Given object expressions as first class citizens, we may now construct objects that return other objects.

Intuitively, a class is an object that returns an *instance* of a class when it is given the message new. Since a class is again an instance of another class, an object that returns the class given its definition.

Using object expressions and abstraction over objects, the above concepts can easily be formalized.

Let $o$ be an object with definition $d$. Then an object $c$ that returns $o$ when given the message new can be defined

$$\text{object } c = [\ \text{method new} = \text{d}\ ],$$

and the displayed object looks like

$$c = [\text{new} \mapsto obj(d)\rho\ ].$$

## Metaclasses

Metaclasses are objects that return a class when given a declaration of an object.

If we have an object expression $d$, then we can obtain the class for $o$ as an instance of the following metaclass *class* definition,

$$\text{object } class = [\ \text{method new} = (\text{dec D})\ \text{object: } [\ \text{method new} = \text{D}\ ]\ ]\ ,$$

which looks like

$$class = [\text{new} \mapsto \lambda d.[\text{new} \mapsto obj(d)\rho]\ ],$$

where $d$ is of type dec.

Then we have

$$c = class.\text{new}(d),$$

and

$$o = c.\text{new}.$$

### Multiple Inheritance

One of the most useful technique in programming with classes is the modularity that can be achieved by abstracting command procedures and data structures into separate classes and then using the inheritance mechanism to build upon what is already there in a clearly defined manner.

As soon as the idea of inheritance has been accepted, the next question is to inherit from more than one class.

We propose to deal with multiple inheritance by defining the product of classes to be a class having all the attributes of each of the components. Then, multiple inheritance reduces to single inheritance.

**Definition.** Let $c_1$ and $c_2$ be classes, i.e. there exit declarations $d_1$ and $d_2$ such that

$$c_1 = \mathit{class}.\mathsf{new}(d_1) \quad \text{and} \quad c_2 = \mathit{class}.\mathsf{new}(d_2).$$

The *product of classes* $c_1$ and $c_2$, written $c_1 \times c_2$, is defined by

$$c_1 \times c_2 \equiv \mathit{class}.\mathsf{new}(d_1 \times d_2),$$

where $d_1 \times d_2$ is just the union of the declarations.

If the union of two object expressions results in an identifier being defined in two different ways, there are a number of ways to resolve the conflict.

1. The strictest way is to take the lattice meet of the two entities. This means that for methods, they both must be defined and be the same before it becomes meaningful.
2. The most generous is to take the lattice join of the two entities. This corresponds to parallel execution.
3. Another possibility is to let one take precedence over the other, as determined by some kind of ordering.

## 4. Summary

In this chapter we constructed a domain of objects as a projective limit of domains of partial objects. Previous work in formulating a semantics of multiple inheritance is by

Cardelli [84], where objects are defined to be sets possessing attributes, like record types of Pascal, and inheritance is then the union of these attributes.

What we have done is provide models of objects that are self-referential in that they have internal identifiers pointing to themselves. In an operational framework, self-referencing can easily be achieved by pointers back to oneself, but in denotational semantics such circularity cannot be modeled. Instead, what we have done is to have the self-referencing identifiers be bound to objects that are isomorphic to oneself. Of course, this is where the projective limit construction was needed.

By making intensional declarations first class objects, we defined a meaning function that returns an object for each object expression. Using object expressions, it becomes a simple matter to define higher order objects like classes and metaclasses, as well as define a notion of product of two object expressions in order to get multiple inheritance via the notion of product of classes.

Chapter 7

# Procedural Reflection

*Wherein a powerful mechanism that allows a program to reflect on its own environment is described, and a model is constructed using projective limit.*

Procedural reflection is a mechanism that allows a program both to refer to its own current environment and other semantic features of the implementation and to modify them in a reasonable manner.

Our source for procedural reflection is Smith [84], who implemented procedural reflection in a version of Lisp called 3-Lisp. 3-Lisp is a language whose data types and functions include computational analogs of notions from language implementations and semantics. One demonstration of 3-Lisp's power is in defining its operational semantics by a meta-circular processor written in 3-Lisp itself.

This chapter presents an overview of the language 3-Lisp, constructs a model of the 3-Lisp processor as a projective limit, and gives examples of reasoning about 3-Lisp programs with the model.

## 1. An Overview of 3-Lisp

This section is a condensation of the salient parts from Smith [84]. First, a word about terminology. Smith adopts a precise convention to prevent ambiguity.

What exists inside a computer (in an abstract way) are *structures*. The way the outside world and a computer communicate is through *notations* that can be typed in or displayed. Notations are the text of programs that are given to an interpreter which produces internal structures the text "notates." Structures have unique *designations* (which they "designate"); that is, they are representing not only external entities, like numbers and trees, but also other internal structures. This designation function is denoted $\Phi$, and the $\Phi$ of a structure is referred to as its *declarative import*. Structures are manipulated by the language processor, denoted $\Psi$, which *normalizes* or does some other operation. The $\Psi$ of a structure is called its *procedural consequence*.

Let $s_1$ and $s_2$ be structures such that $\Psi(s_1) = s_2$. Then we will say that $s_1$ is *self-referential* if $\Phi(s_1) = s_1$, that $\Psi$ *de-references* $s_1$ if $s_2 = \Phi(s_1)$, and that $\Psi$ is *designation-preserving* (at $s_1$) when $\Phi(s_1) = \Phi(s_2)$.

**Definition.** A structure is said to be in *normal form* if

(1) It is *context independent*, in the sense of having the same declarative ($\Phi$) and procedural ($\Psi$) import, independent of context of use;

(2) It is *side-effect free*, implying that the processing of the structure will have no effect on the structural field, processor state, or external world, and

(3) It is *stable*, meaning that it must normalize to itself in all contexts, so that $\Psi$ will be idempotent.

## The Elements of 3-Lisp.

There are seven structure types. Let $a_i$ range over structure notations.

(1) The *numerals* (notated as usual) and (2) the two *boolean* constants (notated $T and $F) are unique, atomic, normal-form designators of numbers and truth-values, respectively.

(3) *Rails* (notated $[a_1\ a_2\ \ldots\ a_k]$) designate sequences.

(4) *Atoms* are used as variables (i.e.., as context-dependent names); as a consequence, no atom is normal-form, and no atom will will ever be returned as the result of processing a structure.

(5) *Pairs* (also called *redexes*, and notated $(a_1.a_2)$) designate the value of the function designated by $a_1$ applied to the arguments designated by $a_2$. By taking the notational form $(a_1\ a_2\ \ldots\ a_k)$ to abbreviate $(a_1.[a_2\ \ldots\ a_k])$ instead of $(a_1.(a_2.(\cdots(a_k.\mathsf{NIL})\cdots)))$, the standard look of Lisp is preserved while maintaining category alignment. (In 3-Lisp there is no distinguished atom NIL, and () is a *notational* error, corresponding to the empty field element.)

(6) *Closures* (notated {closure: ... }) are normal-form function designators, but they are not canonical, since it is in general undecidable whether two structures designate the same function.

(7) Finally, ***handles*** are unique normal-form designators of all structures: they are notated with a leading single quote mark (thus 'a notates the handle of the atom notated a).

The functions of 3-Lisp include the standard arithmetic primitives. Identity (signified with = is computable over all the semantic domain except functions. 1st, and rest are the car/cdr analogs on both rails and sequences. The pair constructor is called pcons (letting $\Longrightarrow$ to mean "normalizes to," we have (pcons 'a 'b) $\Longrightarrow$ '(a . b)); the corresponding constructors for atoms, rails, and closures are called acons, rcons, and cons. There are eleven primitive characteristic predicates, seven for the internal structural types (atom, pair, rail, boolean, numeral, closure, and handle) and four for the external types (number, truth-value, sequence, and function).

The control structures if and cond are defined as usual; block executes a sequence sequentially. body, pattern, and environment are the three selector functions on closures. Functions are defined with structures of the form (lambda simple *args body*), where the keyword simple indicates a standard function as opposed to reflect for reflective functions. (lambda simple [x] (+ x x)) returns a closure that designates a function that doubles numbers.

3-Lisp is higher order and therefore lexically (statically) scoped like the $\lambda$-calculus. It is also meta-structural, providing an explicit ability to name internal structures. Two primitive procedures, called up and down (usually notated with the arrows $\uparrow$ and $\downarrow$) help to mediated between this metastructural hierarchy. Otherwise, there is no way to add or remove quotes — '2 will normalize to '2, never to 2. Specifically, $\uparrow$*struc* designates the normal-form designator of the designation of *struc*; i.e., $\uparrow$*struc* designates what *struc* normalizes to (therefore $\uparrow$(+ 2 3) $\Longrightarrow$ '5). Thus,

> (lambda simple [x] x) designates a function
>
> '(lambda simple [x] x) designates a pair or redex, and
>
> $\uparrow$(lambda simple [x] x) designates a closure.

Similarly, $\downarrow$*struc* designates the designation of the designation of *struc*, providing the designation of *struc* is in normal-form (therefore, $\downarrow$'2 $\Longrightarrow$ 2). $\downarrow\uparrow$*struc* is always equivalent to *struc*, in terms of both designation and result; so is $\uparrow\downarrow$*struc* when it is defined.

## The Meta-Circular Processor

While traditional Lisps have eval and apply as names for the primitive processor procedures, the 3-Lisp analogs are normalize and reduce. (normalize '(+ 2 3)) designates the normal-form structure to which (+ 2 3) normalizes, and therefore returns the handle '5. Similarly,

```
(normalize '(car '(a . b)) ⟹ ''a
(normalize (pcons '= '[2 3])) ⟹ '$F
(reduce '1st '[10 20 30]) ⟹ '10.
```

In any computational formalism able to model its own syntax and structures, it is possible to construct what are commonly known as meta-circular processors (or interpreters). They are "meta" because they operate on other formal structures, and "circular" because they are not proper definitions since they assume one knows the behavior of the processor beforehand.

The top level loop for the 3-Lisp meta-circular processor is given in the following box.

```
(define read-normalize-print
  (lambda simple [level env stream]
    (normalize (prompt&read level stream) env
      (lambda simple [result]
        (block (prompt&reply result level stream)
               (read-normalize-print level env stream))))))
```

3-Lisp Top Level Loop

It is uses explicit continuations to indicate flow of control. The top level consists of a read-normalize-print loop which calls prompt&read with continuation that first does prompt&reply and then calls itself recursively.

The function normalize tests the structure of the input and depending on its type, does accordingly. For example, if it is already in normal form, then it just sends it to the continuation. If it is a pair, it is sent to reduce for reduction.

The function normalize-rail does just that, by picking out the first element and processing it and then joining that value with the recursive result of the rest of the rail.

```
(define normalize
  (lambda simple [struc env cont]
    (cond [(normal struc) (cont struc)]
          [(atom struc) (cont (binding struc env))]
          [(rail struc) (normalize-rail struc env cont)]
          [(pair struc) (reduce (car struc) (cdr struc) env cont)])))

(define reduce
  (lambda simple [proc args env cont]
    (normalize proc env
      (lambda simple [proc!]
        (if (reflective proc!)
            (↓(de-reflect proc!) args env cont)
            (normalize args env
              (lambda simple [args!]
                (if (primitive proc!)
                    (cont ↑(↓proc! . ↓args!))
                    (normalize (body proc!)
                               (bind (pattern proc!) args! (environment proc!))
                               cont))))))))

(define normalize-rail
  (lambda simple [rail env cont]
    (if (empty rail)
        (cont (rcons))
        (normalize (1st rail) env
          (lambda simple [first!]
            (normalize-rail (rest rail) env
              (lambda simple [rest!]
                (cont (prep first! rest!))))))))))
```

3-Lisp Meta-Circular Processor

The heart of the meta-circular processor is the function reduce. It takes the procedure and arguments and first normalizes the procedure body (9). If it is a simple procedure (not reflective), it normalizes the arguments (13) and then tests whether the procedure is primitive, if so then it is executed directly (16), otherwise, the body of the procedure is normalized with the environment augmented by having the formal parameters bound to the actual arguments (18).

If the procedure is reflective (12), then it is made simple, and then executed with the current environment env and continuation cont, as well as its arguments. One way to imagine this is to think of the reflective procedure as being copied into this processor definition and having the procedure be executed by another processor.

## 2. Semantics of 3-Lisp

The semantic domain contains all the entities designated by 3-Lisp structures. These include the internal structures such as pairs, numerals and boolean constants, as well as the external values such as numbers, sequences and functions. Because 3-Lisp can model itself, we must be very careful in distinguishing a name from what a name designates.

### Domain of Structures

There are four types of external entities: the natural numbers ($N$), the truth values ($T$), sequences ($S(\cdot)$, and functions ($[\cdot \rightarrow \cdot]$). The internal structures that name these entities are: the numerals (num), the boolean constants (bool), rails ($R(\cdot)$), and closures ($C$), respectively.

The functions that mediate between semantic levels are *up* ($\uparrow$) and *down* ($\downarrow$). For example, $\downarrow$\$T $=$ the truth value true, and $\uparrow 5 =$ '5, where 5 is a numeral.

There are three additional internal structure types are: the set of atoms which are used to name internal structures, $A$, the set $P(\cdot)$ of pairs of structures, $R(\cdot)$, the rail of structures, and $H(\cdot)$, the handle of structures.

**Definition.** The lattice of *internal structures*, $Q$, is the smallest set containing the numerals, the boolean constants and closures, and closed under pairs, rails, and handles made into a flat lattice:

$$Q = \mathsf{num} + \mathsf{bool} + C + P(Q) + R(Q) + H(Q).$$

The *semantic domain of structures*, $S$, satisfies the following domain equation:

$$S = Q + N + T + P(S) + R(S) + H(S) + [S \to S].$$

## Semantics of Functions

Due to the clean semantic structure of 3-Lisp, once the semantic domain of structures has been given, the semantics of the standard functions mentioned in the previous section is straight forward. The formal semantic function merely echos in the metalanguage the informal definitions.

The only functions that need a little care are $\uparrow$ and $\downarrow$. For handles or internal structures, $\downarrow$ returns the internal structure. For numerals, Boolean constants and function closures, however, $\downarrow$ acts like a semantic function in returning the abstract entity, like an environment.

## Semantics of the Meta-Circular Processor

The crux of the difficulty in defining the semantics of the 3-Lisp meta-circular processor is in dealing with the semantic level crossing that occurs on line (12). The current environment and continuation, which are usually semantic entities, are presented to a procedure as additional arguments. Therefore, there has to be another processor with its own environment and continuation that has to deal with procedure.

In Smith [84], a non-reflective language called 2-Lisp, which is like 3-Lisp except for the reflection mechanism, is used to visualize the 3-Lisp processor. The 3-Lisp processor is thought of as the limit of a tower of 2-Lisp processors each interpreting the one below it, until the bottom one interprets a 3-Lisp procedure. Then, each time a reflective procedure is encountered, the next level processor takes over in interpreting the procedure.

What we do is formalize the above notion by taking the projective limit of an ordered sequence of processor, where the $n$th processor is able to handle reflection up to depth $n-1$.

**Definition.** A structure will be called *reflective*, if it is a pair that contains a reflective procedure as the first element.

Intuitively, the 3-Lisp processor, $p$, satisfies the following equation:

$$p(s,e,c) = \begin{cases} p(s,e,c), & \text{if } s \text{ is not reflective;} \\ p(\textsf{de-reflect}\, s, e[\textsf{env} : e, \textsf{cont} : c], c), & \text{if } s \text{ is reflective,} \end{cases}$$

where de-reflect is a function that takes a reflective function structure and returns the same function that is not reflective.

The way we solve this equation and construct a solution is by first defining a processor for non-reflective structures, then use that as the base for a recursive application of the regular processor definition.

**Notation.** Let N and NR be the definitions of the normalize and the normalize-rail procedures.

**Definition.** Let R0 be the reduce procedure with line 12 modified so that it is undefined for reflective procedures. Then, let $n_0$, $r_0$, and $q_0$ be the solution of the mutual recursion defined by the three procedure definitions N, R0, and NR in the standard environment with all the function identifiers bound to appropriate functions.

Since R0 is undefined for reflective procedures, $n_0$ is a processor for non-reflective structures $s$.

**Definition.** Let $n$,$r$, and $q$ be the denotations of N', R', and NR', where N' is N with reduce and normalize-rail abstracted out, R' and NR' are R0 and NR with normalize abstracted out.

Therefore, we have

$$\begin{aligned} n_0 &= n[r_0, q_0] \\ r_0 &= r[n_0] \\ q_0 &= q[n_0], \end{aligned}$$

where the square brackets indicate the application.

Finally, we are ready to define the sequence of reflective processors.

**Definition.** The inductive definition of the *reflective processor* is given by

$$p_0(s,e,c) \equiv \begin{cases} n_0(s,e,c), & \text{if } s \text{ is not reflective;} \\ \perp, & \text{if } s \text{ is reflective;} \end{cases}$$

$$p_{k+1}(s,e,c) \equiv \begin{cases} n_k(s,e,c), & \text{if } s \text{ is not reflective;} \\ n_k(\mathsf{de\text{-}reflect}\, s, e', c), & \text{if } s \text{ is reflective ,} \end{cases}$$

where

$$n_k \equiv n[r[p_k], q[p_k]],$$

and

$$e' \equiv e[\mathsf{env} : e, \mathsf{cont} : c].$$

Next, we define an ordering on the processors.

**Definition.** For two processors, $p$ and $p'$, define the ordering componentwise:

$$p \sqsubseteq p' \iff \text{for all } x,\ p(x) \sqsubseteq p'(x).$$

Each $p_k$ is well-defined for non-reflective procedures and for $k$ levels of reflection. By taking the limit, we have the full 3-Lisp processor.

**Definition.** Let $\{p_k\}$ be the chain of processors, then the limit,

$$p \equiv \bigsqcup_k p_k,$$

is the 3-Lisp reflective processor.

If we consider the category of processors with the approxiamtion relation as arrows, the limit is again the projective limit in the category.

## 3. Examples

Consider the following reflective function:

```
(define THREE
  (lambda reflect [[] env cont]
    (cont '3)))
```

It explicitly returns the numeral '3. Let $e$ be the standard environment, and $c$ be the initial continuation, which could be the identity function. Then

$$\begin{aligned} p(\mathsf{THREE}, e, c) &= p(\mathsf{de\text{-}reflect\ THREE}, e[\mathsf{env}:e, \mathsf{cont}:c], c) \\ &= p(\mathsf{cont\ '3}, e[\mathsf{env}:e, \mathsf{cont}:c], c) \\ &= c(\mathsf{'3}) \\ &= \mathsf{3}. \end{aligned}$$

The following reflective procedure tests whether a variable is bound in the current environment.

```
(define BOUND
  (lambda reflect [[var] env cont]
    (cont ↑(bound-in-env var env))))
```

And the evaluation of the term (let [[x 3]] (BOUND x)) proceeds as follows:

$$\begin{aligned}
p(\text{(let [[x 3]] (BOUND x))}, e, c) &= p(\text{BOUND}, e_x, c)\\
&\qquad \{\text{where } e_x \equiv e[x:3]\}\\
&= p(\text{de-reflect BOUND}, e', c)\\
&\qquad \{\text{where } e' \equiv e_x[\text{env}:e_x, \text{cont}:c]\}\\
&= p(\text{cont } \uparrow\text{(bound-in-env var env)}, e', c)\\
&= p(\text{cont}, e', c)(p(\uparrow\text{(bound-in-env var env)}, e', c))\\
&= c(\uparrow\$\text{T})\\
&= \$\text{T}.
\end{aligned}$$

We have supressed all the intermediate evaluation that takes place inside the normalize and reduce procedures. The key point to notice is the way the processor binds the current environment and continuation to the current environment, making it available to the procedure.

Using the reflection mechanism, many programming functions that modify current state or control in a dynamic way, such as dynamic debuggers, can be defined in a clean and semantically sound manner.

Finally, the way we reason about programs and processes is equational, using whatever information is contained in our definitions to deduce properties of programs.

# References

The following abbreviations will be used:

| | |
|---|---|
| ACM | Association for Computing Machinery |
| *C. ACM* | *Communications of the ACM* |
| *J. ACM* | *Journal of the Association for Computing Machinery* |
| LNCS | Lecture Notes in Computer Science |

John Backus,

[78] Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *C. ACM* **21**, No. 8 (1978), 613–641.

J. W. de Bakker, J. W. Klop and J.-J. Ch. Meyer,

[81] Correctness of Programs with Function Procedures, in: *Logics of Programs*, ed. by Dexter Kozen LNCS 131, Springer-Verlag (1981), pp. 94–112.

J. W. de Bakker and J. I. Zucker,

[83] Processes and the Denotational Semantics of Concurrency, *Foundations of Computer Science IV*, ed. by J. W. de Bakker and J. van Leewen, Mathematical Centre Tracts 159, Mathematisch Centrum: Amsterdam (1983), pp. 45–100.

H. P. Barendregt,

[81] *The Lambda Calculus: Its Syntax and Semantics*, North-Holland (1981).

H. Barringer, J. H. Cheng, and C. B. Jones,

[84] A Logic Covering Undefinedness in Program Proofs, *Acta Informatica 21* (1984), 251–269.

M. Barr and C. Wells,

[85] *Toposes, Triples and Theories*, Springer-Verlag (1985).

Michael J. Beeson,

[85] *Foundations of Constructive Mathematics*, Springer-Verlag (1985).

Errett Bishop,

[70] Mathematics as a Numerical Language, in: *Intuitionism and Proof Theory*, ed. by A. Kino, J. Myhill, and R. E. Vesley, North-Holland (1970), pp. 53–71.

Errett Bishop and Douglas Bridges,

[85] *Constructive Analysis*, Springer-Verlag (1985).

L. Cardelli,

[84] A Semantics of Multiple Inheritance, in: *Semantics of Data Types*, International Symposium, June 1984, ed. by G. Kahn, D. B. MacQueen, and G. Plotkin, LNCS 173, Springer-Verlag (1984), pp. 51–68.

Alonzo Church,

[41] *The Calculi of Lambda-Conversion*, Princeton University Press (1941).

M. Clint and C. A. R. Hoare,

[72] Program Proving: jumps and functions, *Acta Informatica 1*, 3, (1972), 214–224.

R. Constable and M. O'Donnell,

[78] *A Programming Logic*, Winthrop (Cambridge, Massachusetts 1978).

S. A. Cook,

[78] Soundness and completeness of an axiom system for program verification, *SIAM Journal on Computing* **5** (3) (September 1976), 70–90.

H. B. Curry and R. Feys,

[73] *Combinatory Logic*, Vol. 1, North-Holland (1973).

J. Dugundji,

[66] *Topology*, Allyn and Bacon (1966).

Herbert B. Enderton,

[77] *Elements of Set Theory*, Academic Press (1977).

R. W. Floyd,

[67] Assigning meanings to programs, in: *Mathematical Aspects of Computer Science*, Proceedings of Symposia in Applied Mathematics **19**, ed. by J. T. Schwartz (1967), pp. 19–32.

Adele Goldberg and David Robson,

[83] *Samlltalk-80: The Language and its Implementation*, Addison-Wesley (1983).

Robert Goldblatt,

[79] *Topoi: A Categorial Analysis of Logic*, North-Holland (1979).

G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott,

[80] *A Compendium of Continuous Lattices*, Springer-Verlag (1980).

R. Hindley, B. Lercher and J. Seldin,

[72] *Introduction to Combinatory Logic*, Cambridge University Press (1972).

C. A. R. Hoare,

[69] An axiomatic basis for computer programming, *C. ACM 12*, 10, (October 1969), 576–580 and 583.

Alan Curtis Kay,

[69] *The Reactive Engine*, Ph.D. Thesis, University of Utah (1969).

Saul A. Kripke,

[65] Semantical Analysis of Intuitionistic Logic I, in: *Formal Systems and Recursive Functions*, ed. by J. N. Crossley and M. A. E. Dummet, North-Holland (1965), pp. 92–130.

Daniel J. Lehmann,

[76] Categories for Fixpoint-Semantics, in: *Proc. 17th Annual Symposium on Foundations of Computer Science*, IEEE (1976), pp. 122–126.

C. H. Lindsey and S. G. van der Meulen,

[80] *Informal Introduction to Algol 68*, North-Holland (1980).

Saunders Mac Lane,

[71] *Categories for the Working Mathematician*, Springer-Verlag (1971).

Alain J. Martin,

[83] A General Proof Rule for Procedures in Predicate Transformer Semantics, *Acta Informatica 20*, (1983), 301–313.

Yiannis N. Moschovakis,

[86] Foundations of the Theory of Algorithms, I: The mathematical representation of algorithms, Draft of Paper (1986).

J. Nagata,

[85] *Modern General Topology*, Second Revised Edition, North-Holland (1985).

Michael J. O'Donnell,

[82] A Critique of the Foundations of Hoare Style Programming Logics, *Communications of the ACM 25*, 12, (December 1982), 927–934.

Ernst-Rüdiger Olderog,

[83] Hoare's Logic for Programs with Procedures — What has been achieved?, *Logics of Programs*, ed. by Edmund Clarke and Dexter Kozen, LNCS 164, Springer-Verlag (1983), pp. 383–395.

[84] Correctness of Programs with Pascal-like Procedures without Global Variables, *Theoretical Computer Science 30*, North-Holland (1984), 49–90.

John C. Reynolds,

[81] *The Craft of Programming*, Prentice-Hall International (1981).

[82] Idealized Algol and Its Specification Logic, in: *Tools and Notions for Program Construction*, ed. by D. Néel, Cambridge University Press (1982), pp. 121–161.

Leonid Rudin,

[81] $\lambda$-Logic, Technical Report 4521:TR:81, Computer Science Department, California Institute of Technology (May 1981).

Bertrand Russell,

[02] Letter to Frege (1902), in: *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, Harvard University Press (1967).

Dana Scott,

[70] Constructive Validity, in: *Symposium on Automatic Demonstration*, ed. by M. Laudet and D. Lacombe, Lecture Notes in Mathematics, Springer-Verlag (1970), pp. 237–275.

[71] Models for Various Type-free Calculi, in: *Logic, Methodology and Philosophy of Science IV*, ed. by P. Suppes, L. Henkin, A. Joja, and G.C. Moisil, North-Holland (1973), pp. 157-187.

[72] Continuous Lattices, in: *Toposes, Algebraic Geometry and Logic*, Lecture Notes in Mathematics, vol. 274, Springer-Verlag (1972), pp. 97–136.

[79] Identity and Existence in Intuitionistic Logic, in: *Applications of Sheaf Theory to Algebra, Analysis and Topology*, ed. by M. P. Fourman, C. J. Mulvey, and D. S. Scott, Lecture Notes in Mathematics, vol. 753, Springer-Verlag (1979), pp. 660–696.

Kurt Sieber,

[85] A partial correctness logic for procedures, in: *Logics of Programs*, ed. by Rohit Parikh, LNCS 193, Springer-Verlag (1985), pp. 320–342.

Brian Cantwell Smith,

[84] Reflection and Semantics in LISP, Report No. CSLI-84-8, Center for the Study of Language and Information, Stanford University (December 1984).

Stefan Sokolowski,

[84] Partial Correctness: The Term-Wise Approach, *Science of Computer Programming* **4**, North-Holland (1984), 141–157.

Joseph E. Stoy,

[77] *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press (1977).

Alfred Tarski,

[55] A Lattice-Theoretical Fixpoint Theorem and its Applications, *Pacific Journal of Mathematics* **5** (1955), 285 309.

R. D. Tennent,

[85] Semantical Analysis of Specification Logic, in: *Logics of Programs*, ed. by Rohit Parikh, LNCS 193, Springer-Verlag (1985), pp. 373–386.

D. A. Turner,

[82] Recursion equations as a programming language, in: *Functional Programming and its Applications*, ed. by J. Darlington, P. Henderson, and D. A. Turner, Cambridge University Press (1982), pp. 1–28.

C. Wadsworth,

[76] The Relation Between Computational and Denotational Properties for Scott's $D_\infty$-Models of the Lambda-Calculus, *SIAM Journal on Computing* **5** (3) (September 1976), 488–521.

[78] Approximate Reduction and Lambda Calculus Models, *SIAM Journal on Computing* **7** (3) (August 1978), 337–356.