Stanford Encyclopedia of Philosophy

Games, Full Abstraction and Full Completeness

First published Thu Jan 12, 2017

Computer programs are particular kinds of texts. It is therefore natural to ask what is the meaning of a program or, more generally, how can we set up a formal semantical account of a programming language.

There are many possible answers to such questions, each motivated by some particular aspect of programs. So, for instance, the fact that programs are to be executed on some kind of computing machine gives rise to operational semantics, whereas the similarities of programming languages with the formal languages of mathematical logic has motivated the denotational approach that interprets programs and their constituents by means of set-theoretical models.

Each of these accounts induces its own synonymy relation on the phrases of the programming language: in a nutshell, the full abstraction property states that the denotational and operational approaches define the same relation. This is a benchmark property for a semantical account of a programming language, and its failure for an intuitive denotational account of a simple language based on lambda-calculus has led eventually to refinements of the technical tools of denotational semantics culminating in game semantics, partly inspired by the dialogue games originally used in the semantics of intuitionistic logic by Lorenzen and his school, and later extended by Blass and others to the intepretation of Girard's linear logic. This bridge between constructive logic and programming has also suggested stronger forms of relation between semantics and proof-theory, of which the notion of full completeness is perhaps the most remarkable instance.

1. Introduction

1.1 Interpretations of programming languages

The notion of full abstraction arises from the Scott-Strachey approach to the semantical analysis of programming languages (Scott & Strachey 1971; Strachey 1966, 1967), also known as *denotational* semantics. One fundamental aim of a denotational semantics of a programming language L is to give a *compositional* interpretation $\mathcal{M} : L \to D$ of the *program phrases* of L as elements of abstract mathematical structures (*domains*) D.

We may choose another way of giving meaning to programs, based on their execution. This *operational* interpretation is only defined on the set `Prog` of programs of L, and involves the definition of a suitable set of program *values*, which are the *observables* of L. If the execution of program e terminates with value v, a situation expressed by the notation $e \Downarrow v$, then v is the operational meaning of e. This defines the operational interpretation of programs as a partial function $\mathcal{O}$ from programs to values, where $\mathcal{O}(e) = v$ when $e \Downarrow v$.

Both interpretations induce natural equivalence relations on program phrases. In one of its formulations, full abstraction states the coincidence of the denotational equivalence on a language with one induced by the operational semantics. Full abstraction has been first defined in a paper by Robin Milner (1975), which also exposes the essential conceptual ingredients of denotational semantics: compositionality, and the relations between observational and denotational equivalence of programs. For this reason, full abstraction can be taken as a vantage point into the vast landscape of programming language semantics, and is therefore quite relevant to the core problems of the philosophy of programming languages (White 2004) and of computer science (Turner 2016).

1.2 Compositionality

Compositionality (Szabó 2013) is a desirable feature of a semantical analysis of a programming language, because it allows one to calculate the meaning of a program as a function of the meanings of its constituents. Actually, in Milner's account (see especially 1975: sec. 1, 4), compositionality applies even more generally to *computing agents* assembled from smaller ones by means of appropriate composition operations. These agents may include, beside programs, hardware systems like a computer composed of a memory, composed in turn of two memory registers, and a processing unit, where all components are computing agents. This allows one to include in one framework systems composed of hardware, of software and even of both. Now, the syntactic rules that define inductively the various categories of phrases of a programming language allow us to regard L as an *algebra of program phrases*, whose signature is determined by these rules. One account of compositionality that is especially suitable to the present setting (Szabó 2013: sec. 2) identifies a compositional interpretation of programs with a homomorphism from this algebra to the domain of denotations associating with every operation of the

algebra of programs a corresponding semantical operation on denotations.

As an example, consider a simple imperative language whose programs c denote state transformations $\mathcal{M}(c) : \Sigma \to \Sigma$. Among the operations on programs of this language there is *sequential composition*, building a program $c_1 ; c_2$ from programs c_1 and c_2. The intended operational meaning of this program is that, if $c_1 ; c_2$ is executed starting from a state $\sigma \in \Sigma$, we first execute c_1 starting form state σ. If the execution terminates we obtain a state σ', from which we start the execution of c_2 reaching, if the execution terminates, a state σ''. The latter state is the state reached by the execution of $c_1 ; c_2$ from state σ. From a denotational point of view, we have an operation of composition on states as functions $\Sigma \to \Sigma$, and the compositional interpretation of our program is given by the following identity, to be read as a clause of a definition of $\mathcal{M}$ by induction on the structure of programs:

$$\mathcal{M}(c_1 ; c_2) = \mathcal{M}(c_2) \circ \mathcal{M}(c_1)$$

or, more explicitly, for any state σ:

$$\mathcal{M}(c_1 ; c_2)(\sigma) = \mathcal{M}(c_2)(\mathcal{M}(c_1)(\sigma)).$$

As most programming languages have several categories of phrases (for instance expressions, declarations, instructions) the algebras of programs will generally be multi-sorted, with one sort for each category of phrase. Denotational semantics pursues systematically the idea of associating compositionally to each program phrase a denotation of the matching sort (see Stoy 1977 for an early account).

1.3 Program equivalence and full abstraction

The existence of an interpretation of a programming language L induces in a standard way an *equivalence* of program phrases:

> **Definition 1.1** (Denotational equivalence). Given any two program phrases e, e', they are *denotationally equivalent*, written $e \simeq_{\mathcal{M}} e'$, when $\mathcal{M}(e) = \mathcal{M}(e')$.

If $\mathcal{M}$ is compositional, then $\simeq_{\mathcal{M}}$ is a congruence over the algebra of programs, whose derived operation, those obtained by composition of operations of the signature, are called *contexts*. A context $C[\ ]$ represents a program phrase with a "hole" that can be filled by program phrases e of appropriate type to yield the program phrase $C[e]$. By means of contexts we can characterize easily the compositionality of a semantic mapping:

> **Proposition 1.1.** If $\mathcal{M}$ is compositional, then for all phrases e, e' and all contexts $C[\ ]$:

> (1) $$e \simeq_{\mathcal{M}} e' \Rightarrow C[e] \simeq_{\mathcal{M}} C[e'].$$

This formulation highlights another valuable aspect of compositionality, namely the *referentially transparency* of all contexts, equivalently their *extensionality*: denotationally equivalent phrases can be substituted in any context leaving unchanged the denotation of the resulting phrase. The implication (1) states, in particular, that $\simeq_{\mathcal{M}}$ is a congruence. In order to compare denotational and operational congruence, therefore, we must carve a congruence out of the naive operational equivalence defined by setting $e \sim e'$ if and only if $\mathcal{O}(e) = \mathcal{O}(e')$. This can be done by exploiting *program* contexts $C[\ ]$, representing a program with a "hole" that can be filled by program phrases e of suitable type to yield a complete program $C[e]$.

> **Definition 1.2** (Observational equivalence) Given any two program phrases e, e', they are

observational equivalent, written $e \simeq_{\mathcal{O}} e'$, when, for all program contexts $C[\ ]$ and all program values v:

$$C[e] \Downarrow v \text{ if and only if } C[e'] \Downarrow v.$$

Observational equivalence is then a congruence over the algebra of program phrases, and in fact it is the largest congruence contained in $\sim$. From the general point of view of the account of Milner (1975), that we are following closely, the context of a computing agent represents one of its possible environments. If we adopt the principle that "the overt behavior constitutes the *whole* meaning of a computing agent" (Milner 1975: 160), then the contexts represents intuitively the observations that we can make on the behavior of the computing agent. In the case of programs, the observables are the values, so observational equivalence identifies phrases that cannot be distinguished by means of observations whose outcomes are distinct values. One consequence of Milner's methodological principle is that a computing agent becomes a

> transducer, whose input sequence consists of enquiries by, or responses from, its environment, and whose output sequence consists of enquiries of, or responses to, its environment. (Milner 1975: 160)

A behavior of a computing agent takes then the form of a *dialogue* between the agent and its environment, a metaphor that will be at the heart of the game theoretic approaches to semantics to be discussed in [Section 3](). This behavioral stance, which has its roots in the work of engineers on finite state devices has also been extended by Milner to a methodology of modeling concurrent systems, with the aim

> to describe a concurrent system fully enough to determine exactly what behaviour will be seen or experienced by an external observer. Thus the approach is thoroughly extensional; two systems are indistinguishable if we cannot tell them apart without pulling them apart. (Milner 1980: 2)

In addition, the roles of system and observer are symmetric, to such an extent that

> we would like to represent the observer as a machine, then to represent the composite observer/machine as a machine, then to understand how this machine behaves for a new observer. (Milner 1980: 19)

While observational equivalence is blind to the inner details of a computing agent but only observes the possible *interactions* with its environment in which it takes part, denotational equivalence takes as given the internal structure of a computing agent and, in a compositional way, synthesizes its description from those of its internal parts. The notion of full abstraction is precisely intended to capture the coincidence of these dual perspectives:

Definition 1.3 (Full abstraction). A denotational semantics $\mathcal{M}$ is *fully abstract* with respect to an operational semantics $\mathcal{O}$ if the induced equivalences $\simeq_{\mathcal{M}}$ and $\simeq_{\mathcal{O}}$ coincide.

As a tool for investigating program properties, full abstraction can be seen as a *completeness* property of denotational semantics: every equivalence of programs that can be proved operationally, can also be proved by denotational means. Equivalently, a denotational proof that two terms are not equivalent will be enough to show that they are not interchangeable in every program context.

Full abstraction also functions as a criterion for assessing a translation ϑ from a language L_1 into a (not necessarily different) language L_2, provided the two languages have the same sets of observables, say *Obs* (Riecke 1993). Then ϑ is *fully abstract* if observational equivalence (defined with respect to *Obs*) of

$e, e' \in L_1$ is equivalent to observational equivalence of $\vartheta(e), \vartheta(e')$. The existence of fully abstract translation between languages can be used to compare their expressive power, following a suggestion of (Mitchell 1993; Riecke 1993): L_1 is no more expressive than L_2 if there is a fully abstract translation of L_1 into L_2.

Before going on in this general introduction to full abstraction and related notions in the area of programming languages semantics, in order to show the broad relevance of these notions, it is interesting to observe that there is a very general setting in which it is possible to study the full abstraction property, suggested by recent investigations on compositionality in natural and artificial languages by Hodges (2001) and others. In this setting, full abstraction is connected to the problem of finding a compositional extension of a semantic interpretation of a subset X of a language Y to an interpretation of the whole language, via Frege's *Context Principle* (see Janssen 2001 on this), stating that the meaning of an expression in Y is the contribution it makes to the meaning of the expressions of X that contain it. In the original formulation by Frege X was the set of sentences and Y the set of all expressions, while in programming theory X is the set of programs, Y the set of all program phrases.

A weakening of the definition of full abstraction represents an essential adequacy requirement for a denotational interpretation of a language:

Definition 1.4 (Computational adequacy). A denotational semantics $\mathcal{M}$ is *computationally adequate* with respect to an operational semantics $\mathcal{O}$ if, for all programs e and all values v

$$\mathcal{O}(e) = v \text{ if and only if } \mathcal{M}(e) = \mathcal{M}(v).$$

An equivalent formulation of computational adequacy allows to highlight its relation to full abstraction:

Proposition 1.2. Assume that $\mathcal{M}$ is a compositional denotational interpretation such that $\mathcal{O}(e) = v$ implies $\mathcal{M}(e) = \mathcal{M}(v)$. The following two statements are equivalent:

1. $\mathcal{M}$ is computationally adequate with respect to $\mathcal{O}$;
2. for any two *programs* $e, e' \in \texttt{Prog}$,

$$e \simeq_{\mathcal{M}} e' \text{ if and only if } e \simeq_{\mathcal{O}} e'$$

While the definition of the full abstraction *property* is straightforward, fully abstract models for very natural examples of programming languages have proved elusive, giving rise to a full abstraction *problem*. In our discussion of full abstraction we shall mainly concentrate on the full abstraction problem for the language PCF (Programming language for Computable Functions, Plotkin 1977), a simply typed λ-calculus with arithmetic primitives and a fixed-point combinator at all types proposed in Scott 1969b. This language is important because it includes most of the programming features semantic analysis has to cope with: higher-order functions, types and recursion, with reduction rules that provide an abstract setting for experimenting with several evaluation strategies. Furthermore, PCF is also a model for other extensions of simply typed λ-calculus used for experimenting with programming features, like the Idealized Algol of Reynolds (1981). The efforts towards a solution of the full abstraction problem for PCF contributed, as a side effect, to the systematic development of a set of mathematical techniques for semantical analysis whose usefulness goes beyond their original applications. We shall describe some of them in Section 2, devoted to the semantic analysis of PCF based on partially ordered structures, the *domains* introduced by Dana Scott (1970), that we examine in Section 2.3. Technical developments in the theory of domains and also in the new research area focussed on Girard's *linear logic* (Girard 1987) have led to *game semantics* (Abramsky, Jagadeesan, & Malacaria 2000; Hyland & Ong 2000), which is now regarded as a viable alternative to standard denotational semantics based on domains. It is to this approach that we shall

dedicate [Section 3](#) trying to provide enough details to orient the reader in an extensive and still growing literature documenting the applications of games to the interpretation of a wide spectrum of programming language features.

2. Sequential higher-order computation: the full abstraction problem for PCF

The full abstraction problem has proved especially hard for a version of simply typed λ-calculus with arithmetic primitives called PCF (Programming with Computable Functions) (Plotkin 1977), a toy programming language based on the Logic for Computable Functions of Scott (1969) and Milner (1973). In this section we introduce (a version of) the language with its operational and denotational semantics, and outline how the full abstraction problem arises for this language. The problem has been one of the major concerns of the theoretical investigation of programming languages for about two decades, from its original formulation in the landmark papers (Milner 1977; Plotkin 1977) to the first solutions proposed in 1993 (Abramsky et al. 2000; Hyland & Ong 2000) using game semantics, for which see [Section 3](#).

2.1 Syntax of PCF

PCF is a language based on simply typed λ-calculus extended with arithmetic and boolean primitives, and its type system is defined accordingly:

Definition 2.1 (PCF types). The set `Types` of types of PCF is defined inductively as follows

- the *ground types* `num` (for terms representing natural numbers), `bool` (for terms representing boolean values) are types,
- if σ, τ are types, also $(\sigma \rightarrow \tau)$ is a type.

Parentheses will be omitted whenever possible, with the convention that they associate to the right, so that a type $\sigma_1 \rightarrow \cdots \sigma_n \rightarrow \tau$ is equivalent to $(\sigma_1 \rightarrow (\sigma_2 \rightarrow (\cdots (\sigma_n \rightarrow \tau) \cdots)))$

PCF terms are the terms of simply typed λ-calculus extended with the following arithmetic constants, of the indicated type:

- a constant $0 : \text{num}$, representing the natural number 0;
- a constant succ of type $\text{num} \rightarrow \text{num}$ representing the successor function over natural numbers;
- a constant pred of type $\text{num} \rightarrow \text{num}$ representing the predecessor function over natural numbers;
- constants tt and ff;
- constants of type $\text{bool} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}$ and $\text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$ for conditionals of type num and of type bool, respectively: these are both written as $\text{if} \cdot \text{then} \cdot \text{else} \cdot$, and we let context make clear what is the intended type of the result;
- a constant $\text{zero}?$ for the test for zero of type $\text{num} \rightarrow \text{bool}$;
- a unary function symbol $Y(\cdot)$ for the fixed point combinator, where $Y(e) : \sigma$ for any $e : \sigma \rightarrow \sigma$.

Terms are built inductively according to rules that allow to infer *judgements* of the form $B \vdash e : \sigma$, stating that term e is of type σ under the assumption that the variables occurring free in e are given unique types in a *basis* B of the form

$$\{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}.$$

The rule for building PCF-terms are therefore inference rules for such judgements. In particular there are

rules for typed constants, for example in any basis B there is a judgement $B \vdash \texttt{zero?} : \texttt{num} \rightarrow \texttt{bool}$, and we have rules for typed λ-abstractions

$$\frac{B, x : \sigma \vdash e : \tau}{B \vdash \lambda x : \sigma \,.\, e : \sigma \rightarrow \tau}$$

and applications

$$\frac{B \vdash e_1 : \sigma \rightarrow \tau \qquad B \vdash e_2 : \sigma}{B \vdash e_1 e_2 : \tau}$$

and a rule for the fixed-point operator:

$$\frac{B \vdash e : \sigma \rightarrow \sigma}{B \vdash \texttt{Y}(e) : \sigma}.$$

2.2 Operational semantics

A PCF *program* is a closed term of ground type. We specify how programs are to be executed by defining an evaluation relation $e \Downarrow v$ between closed terms e and *values* v, where the values are the constants and abstractions of the form $\lambda x : \sigma \,.\, e$. In particular, values of ground type $\texttt{bool}$ are $\texttt{tt}, \texttt{ff}$, and values of the ground type $\texttt{num}$ are 0 and all terms of the form

$$\texttt{n} = \underbrace{\texttt{succ}(\ldots \texttt{succ}}_{n}(0)\ldots).$$

Evaluation is defined by cases according to the structure of terms, by means of inference rules for judgements of the form $e \Downarrow v$. These rules state how the result of the evaluation of a term depends on the result of the evaluation of other terms, the only axioms having the form $v \Downarrow v$ for every value v. For example there is a rule

$$\frac{e \Downarrow v}{\texttt{succ}\ e \Downarrow \texttt{succ}\ v}$$

that states that, if the result of the evaluation of e is v, then the result of the evaluation of $\texttt{succ}\ e$ is $\texttt{succ}\ v$. Similarly we can describe the evaluation of the other constants. The evaluation of a term of the form $e_1\ e_2$ proceeds as follows: first e_1 is evaluated; if the evaluation terminates with value v', then the evaluation of $e_1\ e_2$ proceeds with the evaluation of $v'\ e_2$; if this terminates with value v, this is the value of $e_1\ e_2$, formally

$$\frac{e_1 \Downarrow v' \qquad v'\ e_2 \Downarrow v}{e_1\ e_2 \Downarrow v}$$

For a value of the form $\lambda x : \sigma \,.\, e_1$, its application to a term e_2 has the value (if any) obtained by evaluating the term $e_1[e_2/x]$ resulting by substituting e_2 to all free occurrences of x in e_1:

$$\frac{e_1[e_2/x] \Downarrow v}{(\lambda x : \sigma \,.\, e_1)e_2 \Downarrow v}.$$

These implement a *call-by-name* evaluation strategy: in an application, the term in function position must be evaluated completely before the term in argument position, which is then passed as actual parameter.

The fixed point combinator is essential to the encoding of recursive definitions. Its evaluation is described by the rule

$$\frac{e(\mathrm{Y}(e)) \Downarrow v}{\mathrm{Y}(e) \Downarrow v}$$

which is the only rule whose premiss involves the evaluation of a larger term than the one to be evaluated: this is why the definition of the evaluation relation cannot be reduced to structural induction.

We shall be especially interested in situations when the evaluation of a term e does not have a value; in these case we say that e *diverges*, and write $e \Uparrow$. It is in the presence of divergent terms that the causal structure of the evaluation process is exposed. The initial example is in fact a term that diverges in a very strong sense:

Definition 2.2 (Undefined). For any ground type γ, define $\Omega : \gamma$ as

$$\mathrm{Y}(\lambda x : \gamma . x)$$

By inspecting the evaluation rules we see that the only possible evaluation process gives rise to an infinite regress, therefore $\Omega \Uparrow$.

We can define the usual boolean operations by means of the conditional operator, as in the following examples:

(2) $\qquad\qquad$ and $= \lambda x : \mathtt{bool}, y : \mathtt{bool}.\,\mathtt{if}\ x\ \mathtt{then}\ y\ \mathtt{else}\ \mathtt{ff}.$

(3) $\qquad\qquad$ or $= \lambda x : \mathtt{bool}, y : \mathtt{bool}.\,\mathtt{if}\ x\ \mathtt{then}\ \mathtt{tt}\ \mathtt{else}\ y$

with the usual truth tables. However, we have now to take into account the possibility of divergence of the evaluation process, for example in a term like $\mathtt{or}(\Omega, \mathtt{tt})$, therefore we extend the usual truth tables by adding a new boolean value, representing absence of information, $\bot$ (read as "undefined") to $\mathtt{tt}$ and $\mathtt{ff}$, as the value of the term Ω. Here, the first argument to be evaluated is the one on the left, and if the evaluation of this diverges then the whole evaluation process diverges. Consider now an operator $\mathtt{por}$ whose interpretation is given by the table

(4)

por	tt	ff	$\bot$
tt	tt	tt	tt
ff	tt	ff	$\bot$
$\bot$	tt	$\bot$	$\bot$

In this case $\mathtt{por}(\mathtt{tt}, \Omega) = \mathtt{por}(\Omega, \mathtt{tt}) = \mathtt{tt}$: this is the *parallel-or* which plays a central role in the full abstraction problem for PCF. It will turn out that is is not definable by any PCF term, precisely because of its parallel nature. In order to carry out a semantical analysis of PCF, we need a theory of data types with *partial elements* and of functions over them that support an abstract form of recursive definition through fixed point equations: this is what is achieved in Scott's theory of domains, the original mathematical foundation for denotational semantics of programming languages as conceived by Strachey (1966, 1967).

2.3 Denotational semantics

2.3.1 Types as domains

What are the general structural properties of a space of partial data? The mathematical theory of computation elaborated by Dana Scott (1970) is an answer to this question, that takes partially ordered sets generically called *domains* as basic structures. The partial order of a domain describes a qualitative notion of "information" carried by the elements. In such a framework it is natural to reify divergence by introducing a new element $\bot$ representing absence of information. When $x \sqsubseteq y$ in this partial order,

> y is *consistent with* x and is (possibly) *more accurate than* x[…] thus $x \sqsubseteq y$ means that x and y want to approximate the same entity, but y gives *more* information about it. This means we have to allow "incomplete" entities, like x, containing only "partial" information. (Scott 1970: 171)

The resulting partially ordered sets should also have the property that sequences of approximations, in particular infinite chains $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ should converge to a *limit* containing the information cumulatively provided by the x_i. The same structure is also exploited in Kleene's proof of the First Recursion Theorem in Kleene 1952 (secs. 66, 348–50), and will allow to define a notion of continuous function in terms of preservation of limits.

Definition 2.3 (Complete partial orders). A complete partial order (cpo) is a partially ordered set $\langle D, \sqsubseteq \rangle$ with a least element $\bot$, and such that every increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ of elements of D has a least upper bound $\bigsqcup_n x_n$.

Given any set X, we write $X_\bot$ for the set $X \cup \{\bot\}$ obtained by adding a new element $\bot$. It is natural to order the elements of $X_\bot$ according to their amount of information, by setting for $x, y \in X_\bot$,

$$x \sqsubseteq y \iff (x = \bot \ \text{or} \ x = y).$$

Partially ordered structures of the form $X_\bot$ are called *flat domains*, among which we have `bool` $= \{tt, ff\}_\bot$ and `num` $= \mathbb{N}_\bot$, that will be used to interpret the ground types of PCF.

A general requirement on domains is that every element be a limit of its finite approximations, for a notion of finiteness (or *compactness*) that can be formulated entirely in terms of the partial order structure:

Definition 2.4 (Finite elements of a cpo). If D is a cpo, an element $d \in D$ is *finite* if, for every increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$

$$d \sqsubseteq \bigsqcup_n x_n \implies \exists x_i \ (d \sqsubseteq x_i).$$

For $d \in D$, the notation $\mathcal{A}(d)$ denotes the set of finite elements below d; $\mathcal{A}(D)$ is the set of finite elements of D. Finite elements are also called *compact*.

Observe that finite subsets of a set X are exactly the finite elements of the complete lattice of subsets of X. It is useful also to observe that this definition only partially matches our intuition: consider for example the finite element $\infty + 1$ in the cpo

$$0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \cdots \sqsubseteq \infty \sqsubseteq \infty + 1.$$

Definition 2.5 (Algebraic cpo). A D is *algebraic* if, for every $d \in D$, there is an increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ of finite approximations of d such that

$$d = \bigsqcup_n x_n.$$

If D is algebraic, we say that the finite elements form a *basis* of D.

One last completeness assumption on algebraic cpo's is needed in order to get a category of domains suitable for the interpretation of PCF:

Definition 2.6. Given a cpo D, if $X \subseteq D$ has an upper bound we say that X is *consistent*, and write $\uparrow X$, or $x \uparrow y$ when $X = \{x, y\}$. D is *consistently complete* if every $X \subseteq D$ such that $\uparrow X$ has a least upper bound.

The following notion of domain that has proved extremely convenient as a framework for the denotational semantics of programming languages (Scott 1982):

Definition 2.7 (Domain). A *domain* is a consistently complete algebraic cpo with a countable basis.

2.3.2 An abstract theory of computable functions of higher-types

How can we use the notion of information implicit in the ordering on the elements of domains to develop an abstract notion of computability? Clearly, a computable function should preserve *monotonically* any increase of information on its inputs: $f(x) \sqsubseteq f(y)$ whenever $x \sqsubseteq y$. In particular, *strict* functions $f : D \to E$ over flat domains, those for which $f(\bot_D) = \bot_E$, are monotonic.

Consider the domain $\{0, 1\}^\infty$ whose elements are finite and infinite sequences of bits $0, 1$, where $u \sqsubseteq v$ if either u is infinite and $u = v$, or u is finite and u is a prefix of v. What properties should be true of a computable function taking as arguments an infinite sequence of bits $\langle b_1, b_2, b_3, \ldots \rangle$? Take as an example the function *search* : $\{0, 1\}^\infty \to \mathbb{B}_\bot$ whose value is *tt* if, for $u \in \{0, 1\}^\infty$, 1 occurs in u at least once, otherwise $\bot$. Think of the sequence $\langle b_1, b_2, b_3, \ldots \rangle$ as given one element at a time: the initial segments obtained in this process are an increasing chain of finite elements of $\{0, 1\}^\infty$,

$$\langle \rangle \sqsubseteq \langle b_1 \rangle \sqsubseteq \langle b_1, b_2 \rangle \sqsubseteq \langle b_1, b_2, b_3 \rangle \sqsubseteq \ldots$$

having $\langle b_1, b_2, b_3, \ldots \rangle$ as a limit (i.e., least upper bound). By monotonicity we have a corresponding increasing chain of values

$$search(\langle \rangle) \sqsubseteq search(\langle b_1 \rangle) \sqsubseteq search(\langle b_1, b_2 \rangle) \sqsubseteq search(\langle b_1, b_2, b_3 \rangle) \sqsubseteq \ldots$$

If $search(\langle b_1, b_2, b_3, \ldots \rangle) = tt$, then there must be a finite initial segment $\langle b_1, b_2, \ldots, b_n \rangle$ for which $search(\langle b_1, b_2, \ldots, b_n \rangle) = tt$, and this will be the cumulative value of the function for the infinite sequence $\langle b_1, b_2, b_3, \ldots \rangle$. In general, a computable function $f : D \to E$ should (be monotonic and) have the property that a finite amount of information on the output $f(x)$ must be already obtainable by giving a finite amount of information on the input x. This is equivalent to the notion of continuity originally introduced by Scott in his theory of computable functions over domains:

Definition 2.8 (Continuous functions). If $\langle D, \sqsubseteq_D \rangle, \langle E, \sqsubseteq_E \rangle$ are cpo's and $f : D \to E$ is monotonic, f is *continuous* if

$$f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$$

for every increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \subseteq D$.

From the point of view of denotational semantics, a fundamental property of continuous functions

$D \to D$ is that they admit a least fixed point, whose construction can be carried out uniformly and continuously:

Theorem 2.1 (The Fixed Point Theorem for continuous functions) Let $f : D \to D$ be a continuous function and $x \in D$ be such that $x \sqsubseteq f(x)$. Then the element

$$\bigsqcup_{n \in \mathbb{N}} f^{(n)}(x)$$

is the least $y \sqsupseteq x$ such that $f(y) = y$.

Definition 2.9. The least fixed point of a continuous $f : D \to D$ is the element of D defined by

$$\mathtt{fix}(f) =_{\mathrm{def}} \bigsqcup_{n \in \mathbb{N}} f^{(n)}(\bot).$$

The continuous functions from D to E, for cpo's $\langle D, \sqsubseteq_D \rangle$ e $\langle E, \sqsubseteq_E \rangle$, form a cpo $[D \to E]$, ordered pointwise by setting, for $f, g : D \to E$:

$$f \sqsubseteq g \iff \forall d \in D.f(d) \sqsubseteq_E g(d).$$

$[D \to E]$ is a domain if D and E are, and $\mathtt{fix}(\cdot) : [D \to D] \to D$ is continuous. A further construction on cpo's which also extends to domains and is very frequently used is *cartesian product*: given cpo's D, E, their cartesian product is defined as the set $D \times E$ of pairs $\langle d, e \rangle$ where $d \in D$ and $e \in E$, ordered pointwise: $\langle d, e \rangle \sqsubseteq \langle d', e' \rangle$ if and only if $d \sqsubseteq_D d'$ and $e \sqsubseteq_E e'$. We can summarize these constructions in categorical language (Plotkin 1978, Other Internet Resources), saying that the category whose objects are domains and whose morphisms are the continuous functions is *cartesian closed*.

2.3.3 Continuous semantics for PCF

The *standard interpretation* of PCF consists of a family of cpos D^σ, for each type σ, where $D^{\mathtt{num}} = \mathbb{N}_\bot$ and $D^{\mathtt{bool}} = \mathbb{B}_\bot$, $D^{\sigma \to \tau} = [D^\sigma \to D^\tau]$ and the PCF constants have the natural interpretation as strict continuous functions of the appropriate type, for example $\mathtt{cond} : \mathbb{B}_\bot \to \mathbb{N}_\bot \to \mathbb{N}_\bot \to \mathbb{N}_\bot$ is interpreted as:

$$cond(b)(x)(y) = \begin{cases} x & \text{if } b = \mathtt{tt} \\ y & \text{if } b = \mathtt{ff} \\ \bot & \text{if } b = \bot, \end{cases}$$

Furthermore, the operator $\mathtt{Y}(\cdot)$ is interpreted as the continuous functional $\mathtt{fix}(\cdot)$ at the appropriate type. This is the interpretation considered in Scott 1969b) and Milner 1973.

The possibility that e may contain free occurrences of variables (whose types are given by a basis B) slightly complicates the interpretation of terms, making it depend on a further parameter, an *environment* ρ mapping each free variable $x : \tau$ of e to an element of D^τ (if the latter condition is satisfied, we say that ρ *respects* B). Of course, the environment is irrelevant when e is closed.

The standard interpretation of PCF terms $e : \sigma$ (from a basis B) is then an element $[\![e]\!]\rho \in D^\sigma$, for any environment ρ such that ρ respects B, built by structural induction on terms, interpreting application as function application and λ-abstractions by (continuous) functions. More generally, an interpretation is

continuous if every D^σ is a cpo and $D^{\sigma \to \tau}$ consists of continuous functions $D^\sigma \to D^\tau$.

A *model* of PCF is an interpretation that satisfies the expected identities between terms of the same type. We shall omit the details of the general characterization of models of PCF, for which the reader is referred to Ong (1995: sec. 3.2) and Berry, Curien, & Lévy (1985: sec. 4), but just to point out an example of what must be taken into account when such a generality is needed, in order to admit interpretations where the elements at function types are not, strictly speaking, functions, we have to assume a family of *application* operations

$$\cdot_{\sigma\tau} : D^{\sigma \to \tau} \times D^\sigma \to D^\tau$$

so that, if $B \vdash e_1 : \sigma \to \tau$ and $B \vdash e_2; \sigma$, $[\![e_1 e_2]\!]\rho = [\![e_1]\!]\rho \cdot_{\sigma\tau} [\![e_2]\!]\rho \in D^\tau$. A model is *order-extensional* if, for all elements $f, g \in D^{\sigma \to \tau}$, $f \sqsubseteq g$ if and only if $f \cdot x \sqsubseteq g \cdot x$ for all $x \in D^\sigma$. A model is *extensional* if, for all elements $f, g \in D^{\sigma \to \tau}$, $f = g$ if and only if $f \cdot x = g \cdot x$ for all $x \in D^\sigma$. An element $d \in D^\sigma$ of a model is *definable* is there is a closed terms $e : \sigma$ such that $d = [\![e]\!]$.

2.4 Relating operational and denotational semantics

The general setting for discussing full abstraction requires that we introduce the following notions:

Definition 2.11 (Observational preorder and equivalence) Given PCF terms e and e' of the same type σ, we write $e \precsim_{\text{obs}} e'$ (read *e is observationally less defined than e'*) if, for every program context $C[\ ]$ with a hole of type σ and any value v,

$$C[e] \Downarrow v \quad \text{implies that} \quad C[e'] \Downarrow v.$$

We say that e and e' are *observationally equivalent*, and write $e \simeq_{\text{obs}} e'$, if $e \precsim_{\text{obs}} e'$ and $e' \precsim_{\text{obs}} e$.

Observational equivalence is a congruence. Another congruence on PCF terms is given by equality of denotations in a model:

Definition 2.11 (Denotational preorder and equivalence). Given PCF terms e and e' of the same type σ relative to a basis B, we write $e \precsim_{\text{den}} e'$ if $[\![e]\!]\rho \sqsubseteq [\![e']\!]\rho$ for all environments ρ respecting B. We write $e \simeq_{\text{den}} e'$ if $e \precsim_{\text{den}} e'$ and $e' \precsim_{\text{den}} e$.

Proposition 2.1 (Computational adequacy for PCF). The following two statements hold for the standard model of PCF, and are equivalent:

1. For any two PCF terms of the same ground type e, e', $e \simeq_{\text{den}} e'$ implies $e \simeq_{\text{obs}} e'$;
2. For any closed PCF term e of ground type and any value v of that type, $[\![e]\!] = [\![v]\!]$ if and only if $e \Downarrow v$;

We can now justify our intuitive interpretation of $\bot$ in the standard model, where ground types are interpreted as flat domains:

Corollary 2.1. For any closed PCF term e of ground type, $e\Uparrow$ if and only if $[\![e]\!] = \bot$.

In [Section 1.3](#) we have already defined a very general notion of (equational) full abstraction, based on synonymy, i.e., equality of interpretation of terms. In the case PCF, whose intended models are partially ordered at all types, we can define a stronger property:

Definition 2.12 (Inequational full abstraction). A continuous model $\langle \{D^\sigma \mid \sigma \in \text{Types}\}, [\![\cdot]\!]\cdot\rangle$ of

PCF is *inequationally fully abstract* if, for closed terms $e, e', e \precsim_{obs} e'$ implies $[\![e]\!] \sqsubseteq [\![e']\!]$.

Definability is the key to full abstraction, as shown by the following important result of Milner and Plotkin:

> **Theorem 2.2.** A continuous, order-extensional model of PCF is fully abstract if and only if for every type σ, D^σ is a domain whose finite elements are definable.

We turn now to the failure of the full abstraction property for the standard model of PCF, as shown by Plotkin in his classic study (Plotkin 1977):

> **Proposition 2.2.** The standard model of PCF is not fully abstract with respect to call-by-name evaluation.

The proof is based on the observation that we can build PCF terms of type
$(\texttt{bool} \rightarrow \texttt{bool} \rightarrow \texttt{bool}) \rightarrow \texttt{num}$ that recognize the parallel-or function. Specifically, consider the "test" terms T_i defined as follows, where $i = 0, 1$:

$$\lambda f : \texttt{bool} \rightarrow \texttt{bool} \rightarrow \texttt{bool}.\texttt{if} \ (f \ \texttt{tt} \perp_{\texttt{bool}} \ \texttt{then}$$
$$\texttt{if} \ (f \perp_{\texttt{bool}} \texttt{tt}) \ \texttt{then}$$
$$\texttt{if} \ (f \ \texttt{ff} \ \texttt{ff}) \ \texttt{then}$$
$$\perp_{\texttt{num}}$$
$$\texttt{else} \ i$$
$$\texttt{else} \perp_{\texttt{num}}$$
$$\texttt{else} \perp_{\texttt{num}}$$

Then, $\mathcal{D}[\![T_0]\!] \ \texttt{por} = 0 \neq 1 = \mathcal{D}[\![T_1]\!] \ \texttt{por}$, where por is defined by table (4), so $T_0 \simeq_{den} T_1$ does not hold. However, no program context in PCF can separate T_0 and T_1 because por is not definable. This can be shown by characterizing in a combinatorial way the relations of dependence induced by the evaluation process of a program among the evaluation processes of its (sub)terms, as Plotkin does in the Activity Lemma (Plotkin 1977: Lemma 4.2). As an alternative, it is possible to build a computationally adequate models of PCF whose functions enjoy a weak sequentiality property (that we discuss below, in Section 2.5.1) and where, therefore, the function por is ruled out: a complete formal proof along these lines is given in Gunter 1992 (sec. 6.1).

One option to solve the full abstraction problem is to extend the language: a remarkable result of Plotkin (1977) shows that adding parallel-or is enough:

> **Proposition 2.3.** The standard model is fully abstract for the language PCF extended with parallel-or.

Milner (1977) has shown that there is a fully abstract model of PCF, by taking the set of closed terms at each type σ identifying observationally equivalent terms and by completing the resulting partially ordered set turning it into a cpo.

> **Corollary 2.2.** There is a unique continuous, order extensional, inequationally fully abstract model of PCF, up to isomorphism.

The full abstraction problem for PCF consists in finding a direct description of the class of domains and continuous functions that make up the fully abstract model. A solution to this problem would require a precise criterion for assessing the extent to which a proposed description of the model is satisfactory. If one accepts the "precise minimal condition that a semantic solution of the full abstraction problem should satisfy" given by Jung & Stoughton (1993), namely the possibility of describing in an effective way the

domains D^σ of a finitary version of PCF (whose only ground type is `bool`), then the story of failed attempts to give such a direct description of the fully abstract model is justified, with hindsight, by a result of Loader (2001):

> **Theorem 2.3.** Observational equivalence for finitary PCF is not decidable.

It is still possible, however, that one could find a direct description of an *intensionally* fully abstract model (Abramsky et al. 2000: 411):

> **Definition 2.13** (Intensional full abstraction). A model of PCF is *intensionally fully abstract* if every D^σ is algebraic and all its compact elements are definable.

Pursuing this line of development of the full abstraction problem leads us to game semantics, which will be the topic of the next Section. Before that, we outline the main attempts to reduce the model by means of a semantical characterization of higher-order sequential computation.

2.5 Towards a sequential semantics

The reason for the failure of full abstraction of the continuous semantics of PCF is the existence of functions whose evaluation requires parallel computation. We describe now some proposals for characterizing *sequentiality* of functions by means of properties related to the structure of the domains on which they are defined. This has been an area of intensive research toward the solution of the full abstraction problem for PCF, and some of the insights that emerged from it lead very naturally to the game models discussed in [Section 3](). In addition, the following summary of attempts at a characterization of sequentiality is also a very interesting demonstration of the expressive power of the language of partial order in the semantic analysis of programming concepts.

Intuitively, a sequential function is one whose evaluation proceeds serially: this means that it is possible to schedule the evaluation of its arguments so that the evaluation of the function terminates with the correct value; if the evaluation of one of them diverges, the whole evaluation diverges. At each stage of this process there is an argument whose value is needed to obtain more information on the output of the function. In order to account for this causal structure of computations at the semantical level, we need to enrich the domain structure so that the order on the elements reflect the happening of computational *events* and their causal order. This suggests another way of interpreting the abstract notion of information that motivated the axioms of a cpo in [Section 2.3.1](). Now,

> information has to do with (occurrences of) events: namely the information that those events occurred. For example in the case of $\mathbb{N}_\perp$, $\perp$ might mean that no event occurred and an integer n, might mean that the event occurred of the integer n being output (or, in another circumstance being input). (Plotkin 1978, Other Internet Resources)

2.5.1 Stability

One interpretation of events regards them as the production of values in the evaluation of an expression. This interpretation originates in the context of bottom-up computation of recursive programs developed by Berry (1976), where a recursive definition is translated into a graph displaying the dependence of results of an expression on results of its subexpressions. This context naturally suggests the notion of *producer* of an event x, as a set of events that must have happened in order that x may happen. Reformulating this observation in the language of partial orders, Berry (1976) defined:

> **Definition 2.14** (Stability). Let $D_1, \ldots, D_n, D$ be flat cpo's and $f : D_1 \times \ldots \times D_n \to D$ monotonic

(hence continuous). Then f is *stable* if for every $\vec{x} = \langle x_1, \ldots, x_n \rangle \in D_1 \times \ldots \times D_n$ there is a unique minimal element $m(f, x) \sqsubseteq \vec{x}$ such that $f(m(f, x)) = f(\vec{x})$.

Clearly, the parallel-or function is not stable: the value $\text{por}(\bot, tt) = tt = \text{por}(tt, \bot)$ has no minimal producer. A remarkable property of stable functions is that they allow to build a new model of PCF, where $D^{\sigma \to \tau}$ is the set of stable functions on the domains that interpret the types σ and τ, which are refinements of Scott domains called *dI-domains* (Berry 1978). From our point of view, the important outcome of these definitions is the following adequacy result (Gunter 1992: chap. 6):

> **Proposition 2.4.** The interpretation of PCF terms as elements of dI-domains, where $D^{\sigma \to \tau}$ is the dI-domain of stable functions from D^σ to D^τ with the stable order, is a computationally adequate model of PCF.

This result completes the argument showing the failure of full abstraction for the continuous model of PCF at the end of [Section 2.4](#), if the informal notion of sequentiality used there is formalized as stability. The stable model of PCF has recently been shown to be fully abstract for an extension of PCF (Paolini 2006).

2.5.2 Sequential functions

The first definitions of sequentiality, due to Vuillemin (1974) and Milner (1977) stated that an n-ary functions f over flat domains is *sequential at argument* $\langle x_1, \ldots, x_n \rangle$ if there is a *sequentiality index i* of f, depending on $\langle x_1, \ldots, x_n \rangle$, such that every increase in the output information must increase the information at argument i. For example, the function $\text{cond} : \mathbb{B}_\bot \times \mathbb{N}_\bot \times \mathbb{N}_\bot \to \mathbb{N}_\bot$ is sequential in this sense at any input tuple. In fact, its sequentiality index at $\langle \bot, m, n \rangle$ is 1; its sequentiality index at $\langle tt, m, n \rangle$ is 2, and its sequentiality index at $\langle ff, m, n \rangle$ is 3. There is however no sequentiality index for the function $\text{por} : \mathbb{B}_\bot \times \mathbb{B}_\bot \to \mathbb{B}_\bot$ at the input $\langle \bot, \bot \rangle$.

While all sequential functions (over flat domains) are stable, sequentiality is strictly stronger than stability. For example, the continuous function from $\mathbb{B}_\bot \times \mathbb{B}_\bot \times \mathbb{B}_\bot$ to $\mathbb{B}_\bot$ defined as the smallest continuous extension of the three assignments

$$\langle tt, ff, \bot \rangle \mapsto tt, \langle ff, \bot, tt \rangle \mapsto tt, \langle \bot, tt, ff \rangle \mapsto tt.$$

has no sequentiality index at the argument $\langle \bot, \bot, \bot \rangle$, but is stable because the arguments $\langle tt, ff, \bot \rangle, \langle ff, \bot, tt \rangle, \langle \bot, tt, ff \rangle$ are pairwise inconsistent.

The following result adds support to the search for a semantical characterizations of sequentiality:

> **Proposition 2.5.** Let $f : D_1 \times \cdots \times D_n \to D$ be a continuous function, where D_i, D are either $\mathbb{N}_\bot$ or $\mathbb{B}_\bot$. Then f is sequential if and only if it is definable in PCF.

2.5.3 Concrete data structures and sequential algorithms

If the domains needed for an adequate definition of sequentiality are to describe the causality relations among occurrences of computational events, then it is necessary to enrich our picture by considering events as located at *places*, generalizing the notion of argument place in the definitions of Vuillemin and Milner which depends on how a function is presented. This led to a notion of *concrete data structure* (cds) (Kahn & Plotkin 1993) and to an axiomatization of the order-theoretic properties of domains of first-order data. Kahn and Plotkin obtained a representation theorem for the domains described by their axioms, the *concrete domains*, in terms of the *states* of a process of exploration of a concrete data structure that

consists in filling, given a state x, any cell enabled by sets of events that have already happened in x, starting from *initial* cells enabled in the initial, empty state: this is similar to proving theorems in an abstract deductive system whose rules are the enablings. As a motivating example, think of a linked list of, say, natural numbers. The initial cell may be filled at any time with any value n_1. This event enables the second cell of the list, which may then (and only then) be filled with any value n_2, and so on for all later cells.

Observe that the framework of concrete data structures gives the necessary notions to reconstruct a semantical version of sequentiality. Roughly, a monotonic function f from states of M to states of M' is *sequential* (at state x) if, for any output cell c', there is an input cell c that must be filled in any transition from x to y such that the transition from $f(x)$ to $f(y)$ fills c' (if such a c' does exist) (Curien 1986: Def. 2.4.5). The cell c is the *sequentiality index* for f at x for c'.

The category whose objects are the concrete data structures and whose morphisms are the sequential functions just defined is, however, not cartesian closed, not unexpectedly. This observation (for a simple proof, see Amadio & Curien 1998 (theorem 14.1.12)) prevents the use of this category as a model of PCF. However, it is possible to define for every two concrete data structures M, M' a new one $M \rightarrow M'$ whose states represent *sequential algorithms* and which is the exponential object of M and M' in a cartesian closed category whose morphisms are sequential algorithms (Curien 1986: sec. 2.5). The generalizations of the model theory of PCF to categorical models allows us to obtain a model of PCF from this new category, even though its morphisms are not functions in the usual set-theoretic sense. It turns out that the sequential algorithm model is not extensional, because there are distinct PCF terms that denote the same continuous function yet represent distinct algorithms. As an example, consider the following two terms, that denote the same function but different algorithms:

$$\texttt{lror}(x, y) = \texttt{if } x \texttt{ then (if } y \texttt{ then tt else } x)$$
$$\texttt{else (if } y \texttt{ then tt else ff)}$$
$$\texttt{rlor}(x, y) = \texttt{if } y \texttt{ then (if } x \texttt{ then tt else } y)$$
$$\texttt{else (if } x \texttt{ then tt else ff)}.$$

By suitably introducing error values $error_1$, $error_2$ in the semantics, and enforcing an error-propagation property of the interpretations of terms (thus enlarging the observables of the language), the *functions* corresponding to the above terms can then be distinguished: clearly, for the interpreting functions *lror* and *rlor* we have

$$lror(error_1, error_2) = error_1 \qquad rlor(error_1, error_2) = error_2$$

which also points to the possibility of proving full abstraction of this (non-standard) extensional model with respect to an extension of PCF with control operators (Cartwright, Curien, & Felleisen 1994).

Before leaving this overview of the quest for an extensional characterization of higher-order sequentiality, we should mention Bucciarelli & Ehrhard (1994) who introduced a refinement of the dI-domains of Berry supporting a notion of *strongly stable function* which allows them to build an extensional model of PCF, which is not fully abstract. The reason for the failure of full abstraction in this case depends on the fact that PCF-definable functionals satisfy extensionality properties that fail when functions are ordered by the stable order. This was also the reason that motivated the introduction of *bidomains* (Berry 1978), where the stable and extensional (= pointwise) orderings of functions coexist.

2.6 Historical notes and further readings

The problem of full abstraction has been anticipated in a large amount of work on the relations between the denotational and operational interpretations of programming languages. In particular, the pioneering work on the semantics of recursive programs carried out in Stanford in the early 1970s by a group of people gathering around Zohar Manna, and including Jean Marie Cadiou, Robin Milner and Jean Vuillemin, also interacting with Gilles Kahn.

A related tradition was also quite influential on the background of the full abstraction problem, namely the characterizations of semantical notions like continuity and sequentiality inside syntactic models of the (untyped) λ-calculus based on Böhm trees (Barendregt 1984), mainly due to Lévy and Berry (see Berry et al. 1985 and Curien 1992) for accounts of the search for fully abstract models of PCF along this line).

The basic papers on full abstraction for PCF are Milner 1977; Plotkin 1977. They can be read together as giving a coherent picture of the semantic analysis of this language. An independent approach to full abstraction came from the Russian logician Vladimir Sazonov who characterized definability in PCF in terms of a certain class of sequential computational strategies (Sazonov 1975, 1976). His work, however, had no direct influence on the bulk of research on the full abstraction problem, and only recently there have been attempts to relate Sazonov's characterization to the game theoretic approaches (Sazonov 2007).

Another, completely different approach to full abstraction, exploits special kinds of *logical relations* in order to isolate quotients of the continuous model. The first use of logical relations in the context of the problem of full abstraction is Mulmuley 1987, but the resulting construction of a fully abstract model is obtained by brute force and therefore is not what the full abstraction problem searches for. Later, Sieber (1992) and O'Hearn & Riecke (1995) have employed refinements of this technique to gain a better insight into the structure of the fully abstract models, characterizing the definable elements of the standard continuous model by means of invariance under special logical relations cutting out the non-sequential functions.

Detailed accounts of the full abstraction problem for PCF can be found in Gunter 1992 (chaps 5,6), Streicher 2006, Ong 1995, Stoughton 1988 and Amadio & Curien 1998 (chaps 6, 12, 14), in approximately increasing order of technical complexity. The emphasis on the recursion-theoretic aspects of PCF and its full abstraction problem are dealt with in detail in the textbook (Longley & Normann 2015: chaps 6, 7); a shorter account can be found in Longley 2001 (sec. 4).

3. Game semantics

3.1 Full completeness

Theorem 2.2 highlights the fundamental role of definability of finite elements in the fully abstract model of PCF, an aspect that has been stressed recently in Curien 2007. As a smooth transition to the formalisms based on games, and partly following the historical development of the subject, we pause shortly to examine another aspect of definability that arises at the border between computation and the proof theory of constructive logical systems. It has been a remarkable discovery that the structure of natural deduction proofs for, say, the implicative fragment of intuitionistic propositional calculus is completely described by terms of the simply typed λ-calculus, where a provable propositional formula of the form $\sigma \rightarrow \tau$ is read as the type of the terms representing its proofs. This is the *propositions-as-types correspondence*, to be attributed to Curry, de Bruijn, Scott, Läuchli, Lawvere and Howard, which extends to much richer formal systems (for a history of this notion see Cardone & Hindley 2009: sec. 8.1.4).

The existence of this correspondence makes it possible to speak of a *semantics of proofs*, that extends to constructive formal proofs the denotational interpretations of typed λ-calculi, and in this context it also

makes sense to ask whether an element x of some D^σ in a model of a typed λ-calculus is the interpretation of some proof of formula σ. A further question asks whether *every* element of D^σ satisfying a suitably chosen property is the interpretation of a proof of formula σ. Suitable properties may be for example invariance under logical relations, suitably defined over each D^σ, like in several results of Plotkin, Statman and others summarized in Barendregt, Dekkers, & Statman 2013 (I.3, I.4). We can read the latter question as asking for a strong form of completeness for that system called *full completeness* (Abramsky & Jagadeesan 1994), whose definition can be better understood in a categorical semantics of systems of constructive logic. It is common to interpret formulas A of such systems as objects $[\![A]\!]$ of suitable categories $\mathbb{M}$, and proofs p of sequents $A \vdash B$ as morphisms $[\![p]\!] : [\![A]\!] \longrightarrow [\![B]\!]$. While ordinary completeness states that for every valid sequent $A \vdash B$ the set $\mathbb{M}([\![A]\!], [\![B]\!])$ of morphisms is not empty, in the present setting full completeness expresses the stronger requirement that every morphism $f : [\![A]\!] \longrightarrow [\![B]\!]$ in a semantical category $\mathbb{M}$ arises as the interpretation of some proof, i.e., $f = [\![p]\!]$ for some proof p of the sequent $A \vdash B$. Full completeness results have been proved for several subsystems of linear logic Girard (1987), see Abramsky (2000) for a general framework. Furthermore, it has also suggested techniques for achieving the definition of models of PCF enjoying the strong definability property required by intensional full abstraction.

3.2 Interaction

In our description of the refinements to the continuous model of PCF in order to guarantee the definability of finite elements at each type, we have progressively come closer to an interactive explanation of computation. For example, the action of a sequential algorithm $M \to M'$ (Curien 1986: sec. 3.4) exploits an external calling agent which triggers a cycle of requests and responses on input cells leading (possibly) to the emission of an output value. That interaction should be a central notion in the analysis of computation, especially in relation to full abstraction, is perhaps a natural outcome of the observational stance taken in the definition of operational equivalence. Our short account of game semantics starts precisely from an analysis of a general notion of *interaction* as a motivation to a first formalization of games which is however rich enough to provide a universe for the interpretation of a restricted set of types and terms. Later we shall add to this definition of game and strategies the features needed to express the constraints that allow strategies to characterize precisely higher-order, sequential computations, which is the aim set for denotational semantics by the full abstraction problem. The present account of the conceptual background of game semantics owes much to the work of Abramsky and Curien (Abramsky 1994, 1996, 1997; Curien 2003a).

The relevant notion of interaction has been isolated as the result of contributions that come from widely different research areas intensively investigated only in relatively recent years, notably linear logic (Girard 1987) and the theory of concurrent processes. It is in these areas that a notion of *composition as interaction* of *modules* takes shape. We give here just a simple example where the composition of modules in the form of "parallel composition + hiding" is found in nature, in order to connect it with the origin of this idea in the semantics of concurrent processes developed by Hoare (1985), and also to afford a first glimpse into a simplified game formalism.

Consider a module S with four channels labeled $a_{\mathrm{in}}, a_{\mathrm{out}}, r_{\mathrm{in}}, r_{\mathrm{out}}$. The module is intended to return on channel a_{out} the successor of the number n incoming through channel a_{in}, therefore its behavior can be specified as follows:

- S receives an input signal $?_{\mathrm{in}}$ on channel r_{in}, then
- emits a signal $?_{\mathrm{out}}$ on channel r_{out}, and
- waits for a value n on channel a_{in} and then, after receiving it,
- emits a value $n + 1$ on channel a_{out}.

(This pattern of interaction is formally identical to the *handshake protocol* which is used in hardware design to synchronize components in order to avoid hazards caused by interference of signals.) This behavior can be mapped on the channels as follows:
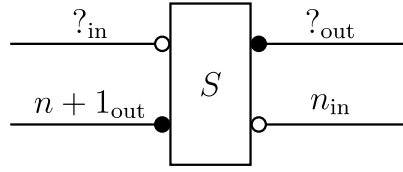

FIGURE 1: A module for the successor function.

where $\circ$ means input or, more generally, a *passive* involvement of the module in the corresponding action, whereas $\bullet$ means output, or *active* involvement in the action. We can describe the behavior of S using *traces* (Hoare 1985), i.e., finite sequences of symbols from the infinite alphabet $\alpha S = \{?_{in}, ?_{out}, n_{in}, m_{out}\}$:

$$\tau S = \{\varepsilon, ?_{in}, ?_{in}?_{out}, ?_{in}?_{out}n_{in}, ?_{in}?_{out}n_{in}n + 1_{out}, \ldots\}$$

If we consider another instance S' of S with alphabet $\alpha S' = \{?'_{in}, ?'_{out}, n'_{in}, m'_{out}\}$ we can compose S and S' by identifying (= connecting) channels r_{out}, r'_{in}, and a_{in}, a'_{out}, and the signals passing through them, as shown:

$$?_{out}, ?'_{in} \rightsquigarrow x$$
$$n + 1_{in}, n + 1'_{out} \rightsquigarrow y$$

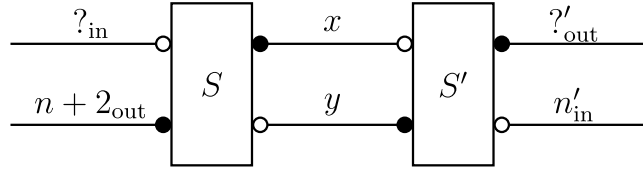This represents the parallel composition of the modules, $S\|S'$:

FIGURE 2

The behavior of the compound module is described by the set of traces

$$\{\varepsilon, ?_{in}, ?_{in}x, ?_{in}x?'_{out}, ?_{in}x?'_{out}n'_{in}, ?_{in}x?'_{out}n'_{in}y, ?_{in}x?'_{out}n'_{in}yn + 2_{out}, \ldots\}$$

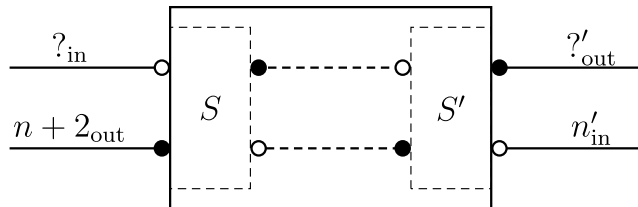The symbols x, y can now be hidden, representing the behavior of the final system

FIGURE 3

whose traces have the required form

$$\{\varepsilon, ?_{in}, ?_{in}?'_{out}, ?_{in}?'_{out}n'_{in}, ?_{in}?'_{out}n'_{in}n + 2_{out}, \ldots\}.$$

This example contains many of the ingredients on which game semantics is based. There is the idea of a System, whose behavior is triggered by an incoming request from its Environment: in a game formalism these are the roles of Proponent and Opponent in a two-person game. The behavior of each module is described as the trace of its possible interactions with other agents, and the behaviors can be composed by a peculiar change of role whereby the module who plays as System (in the above example, S emitting a request signal on channel r_{out}) is made to behave as Environment with respect to S' when this signal is received in input on channel r'_{in}. Let us see how this example can be generalized.

3.3 Games and strategies

We only give the definitions needed to understand the basic constructions on games and to see how these form a category, following Abramsky 1997 and Hyland 1997 that contain more formal details and proofs.

3.3.1 Games

Definition 3.1 A *game G* is specified by giving a set of *moves M_G*, a *labeling ℓ_G* of the moves as either moves of the *Proponent* (P) or as moves of the *Opponent* (O). Furthermore, there is a set of *positions P_G* which is a set of sequences of moves where: (1) the two players alternate, starting with O; (2) if $s \in P_G$ then every prefix s' of s is also in P_G.

As an example, consider a game associated with the data-type of Boolean values, G_{bool}. There are three possible moves,

- an O-move $?_{\text{bool}}$ and
- two P-moves tt, ff

(i.e., $\ell_{\text{bool}}(?_{\text{bool}}) = O, \ell_{\text{bool}}(tt) = \ell_{\text{bool}}(ff) = P$). The positions in this game are

$$?_{\text{bool}}, ?_{\text{bool}}\texttt{tt}, ?_{\text{bool}}\texttt{ff} :$$

think of $?_{\text{bool}}$ as a cell (as in a concrete data structure) which can be filled by one of the two values tt and ff, or as a question by the Opponent that admits as answers by the Proponent either tt or ff. Similarly we can describe a game G_{num} with an O-move $?_{\text{num}}$ and P-moves $n \in \mathbb{N}$.

3.3.2 Strategies and their composition

The players move in a game G alternately, at each move reaching a legal position in P_G. Their behavior is best thought of as describing a strategy that prescribes deterministically what is P's response to O in a position where it is its turn to move.

Definition 3.2. A *strategy σ* on a game G is a prefix-closed set of even-length positions of G such that, each time $sab, sac \in \sigma$, we have $b = c$.

For example, the strategies on G_{num} are ε and all sequences $?_{\text{num}}n$, corresponding respectively to the elements $\bot$ and n of the domain $\mathbb{N}_\bot$.

We would like to consider the behavior of the successor module described above as an element of a set $G_{\text{num}} \multimap G_{\text{num}}$ of strategies that compute functions over the natural numbers. If we consider only the sequences of interactions of S taking place either on the left or on the right side of the module of Figure 1, we see that they describe positions of G_{num}, with an inversion of polarity (active/passive) depending on

which side the interactions take place: the module is initially passive, and becomes active upon receiving a request from the environment. Such inversion, represented by the complementary labeling of the moves $\overline{\lambda_G}$, assigning to Proponent the moves of the Opponent in G and conversely, is essential to the definition of a game $G \multimap H$:

> **Definition 3.3.** Given any pair of games G, H, the game $G \multimap H$ has moves $M_{G \multimap H}$ the disjoint union $M_G + M_H$ of the games G and H, where
>
> $$\lambda_{G \multimap H}(m) = \begin{cases} \overline{\lambda_G}(m) = & \text{if } m \in M_G, \\ \lambda_H(m) = & \text{if } m \in M_H. \end{cases}$$
>
> and a position in $P_{G \multimap H}$ is any alternating sequence s of moves (of $M_{G \multimap H}$) whose restrictions $s \restriction M_G, s \restriction M_H$ to the moves in G and H, respectively, are positions of G and H.

The strategy that interprets $\texttt{succ} : \texttt{num} \to \texttt{num}$ corresponds to the behavior of the module S used above as a guiding example. The parallel composition + hiding approach used to compose two instances of the successor module can now be reinterpreted as composition of strategies, suggesting a general pattern:

G_{num}	$\multimap$	G_{num}		G_{num}	$\multimap$	G_{num}	
						$?_{\text{in}}$	O
		$?'_{\text{in}}$	O		$?_{\text{out}}$		P
$?'_{\text{out}}$			P				O
$\vdots$			$\vdots$				$\vdots$
n'_{in}			O				P
		$n+1'_{\text{out}}$	P		$n+1_{\text{in}}$		O
			O			$n+2_{\text{out}}$	P

> **Definition 3.4.** The composition $\tau \circ \sigma$ on $G \multimap K$ of strategies σ on $G \multimap H$ and τ on $H \multimap K$ consists of the sequences of moves of $M_G + M_K$ obtained by hiding the moves of M_H from the sequences s of moves in $M_G + M_H + M_K$ such that $s \restriction G, H$ is in $P_{G \multimap H}$ and $s \restriction H, K$ is in $P_{H \multimap K}$.

There is one strategy that deserves a special name, because it is the identity morphism in the category whose objects are games and whose morphisms from G to H are the strategies on $G \multimap H$. The *copy-cat strategy* id on $G \multimap G$ is defined as the set of sequences of moves s such that the restriction of s to the left instance of G coincides with its restriction to the right instance.

3.4 Special kinds of strategies

The game formalism just introduced is not detailed enough to characterize the kind of sequential computation at higher types needed to achieve definability. For this purpose, a richer structure on games is needed, making them closer to *dialogue games* between Proponent and Opponent exchanging *questions* and *answers*. This allows to formulate restrictions on plays by matching answers with the corresponding questions in an appropriate manner. The strategies for this refined game notion, that we study next essentially through examples, will yield a richer notion of morphism between games, allowing to make finer distinctions of a computational nature needed for intensionally fully abstract model of PCF, following essentially the approach of Hyland & Ong (2000) drawing also material from Abramsky & McCusker (1999) and Curien (2006).

The moves of the refined game notion will be either *questions* or *answers* played by Proponent or by the Opponent. We have then four classes of moves each represented by a kind of (round or square) bracket: Proponent's questions '('; Opponent's answers ')'; Opponent's questions '['; and Proponent's answer ']'. This labeling of the moves subsumes under the usual well-formedness criterion for bracket sequences, at one time: the alternation between Proponent and Opponent, the fact that Opponent is the first to move and that each answer of a player answers a unique question of the partner. This is not enough, however: a further *justification* structure on questions and answers is needed to discipline the nesting of (sub)dialogues in the evaluation of higher-order functions, allowing to characterize the *well-bracketed* strategies. Consider now the strategy in $(G_{\texttt{bool}}^{11} \to G_{\texttt{bool}}^{12} \to G_{\texttt{bool}}^{1}) \to G_{\texttt{bool}}$, described informally using a labeling of the copies of $G_{\texttt{bool}}$ as shown:

(1)   Opponent asks question ? in $G_{\texttt{bool}}$;
(2)   Proponent asks question $?_1$ in $G_{\texttt{bool}}^1$, justified by ?, in order to know about the output of the input value f ;
    (3.1)   if Opponent asks question $?_{11}$, Proponent answers tt in $G_{\texttt{bool}}$: the computation examines first the first argument of f ;
    (3.2)   if Opponent asks question $?_{12}$, Proponent answers ff in $G_{\texttt{bool}}$: the computation examines first the second argument of f ;

Here, the Proponent's moves at steps $(3.i)$ answer the question asked by Opponent at step (1), not the questions asked by the Opponent at steps (3.1), (3.2) that are still pending. This violates a "no dangling question mark" condition on dialogues introduced under this name by Robin Gandy in his unpublished work on higher-type computability (and well-known in the tradition of game semantics for intuitionistic logic initiated by Lorenzen (1961)). Strategies such as these interpret control operators that do not exist in the fully abstract game model of PCF, but do exist, for example, in the model based on sequential algorithms (Curien 1986: sec. 3.2.7, 3.2.8). A different phenomenon occurs in a variation of the previous example:

(1)   Opponent asks question ? in $G_{\texttt{bool}}$;
(2)   Proponent asks question $?_1$ in $G_{\texttt{bool}}^1$;
  (3.1)   if Opponent asks question $?_{11}$, Proponent answers tt in $G_{\texttt{bool}}^{11}$;
  (3.1.1)   if Opponent answers tt in $G_{\texttt{bool}}^1$, Proponent answers tt in $G_{\texttt{bool}}$;
  (3.2)   if Opponent answers tt in $G_{\texttt{bool}}^1$, Proponent answers ff in $G_{\texttt{bool}}$

Here the strategy prescribes a response to the moves by Opponent depending on the internal detail of the latter's behavior. The response prescribed to Proponent by the strategy to the initial question should not depend on what happens between the Proponent's question $?_1$ and the Opponent's answer tt. This is the property of *innocence*, that limits the amount of detail that a strategy for P can access. For this reason, failure of innocence allows strategies to model storage phenomena.

This gives us the necessary terminology to understand the statement of the intensional full abstraction theorem proved in Hyland & Ong 2000 (th. 7.1), where the types of PCF are interpreted as games and terms as innocent and well-bracketed strategies, see also Abramsky et al. 2000 (th. 3.2), Curien 2006 (th. 5.1):

> **Theorem 3.1.** For every PCF type $\sigma = \sigma_1 \to \cdots \to \sigma_n \to \kappa$ with $\kappa = \texttt{num}$ or $\kappa = \texttt{bool}$, every (compact) innocent and well-bracketed strategy corresponds to the denotation of a closed term.

This closes our quick overview of game semantics applied to the full abstraction problem for PCF, but opens a broad research area in the classification of programming disciplines according to the possible combinations of restrictions (innocence, well-bracketing) on general strategies for games as defined

above. An introductory picture (the "semantic square" by Abramsky and his students) of this landscape, that we leave to the contemplation of the reader, can be found in Abramsky & McCusker 1999.

3.5 Historical notes and further readings

Games as a semantic framework have a longstanding tradition, from ancient logic onwards. Here we list of the main sources and further readings pertaining to game semantics applied to programming languages.

The use of game semantic for dealing with the full abstraction problem for PCF originates from Abramsky et al. 2000 and Hyland & Ong 2000. Hanno Nickau (1994) proposed independently a game model similar to that of Hyland and Ong: their games are sometimes called collectively "H_2O games".

As a background for game semantics, from intuitionistic logic we have the very early Lorenzen (1961) on dialogue games, then from linear logic Lafont and Streicher (1991) and Blass (1992) and from Coquand's game theoretical analysis of classical provability (Coquand 1995). From combinatorial game theory the categorical account by Joyal (1977), "the first person to make a category of games and winning strategies" according to Abramsky & Jagadeesan (1994). A readable historical account of the first uses of games in the interpretation of constructive logical formalisms, especially linear logic, is included in Abramsky & Jagadeesan 1994. It should be observed that games for logic require winning strategies in order to capture validity, an issue that we have not dealt with at all in this entry.

Connections with concrete data structures were first noticed by Lamarche (1992) and Curien (1994), see Curien 2003b. Antonio Bucciarelli (1994) explains the connections between Kleene's unimonotone functions and concrete data structures: the use of dialogues in the former is mentioned in Hyland & Ong 2000 (sec. 1.4).

Finally, among the introductions to game semantics for PCF and other languages, we suggest Abramsky 1997; Abramsky & McCusker 1999. The latter also contains a description of the applications of game semantics to imperative languages, notably Idealized Algol. Other excellent introductions to game semantics are Hyland 1997 and Curien 2006. A broad account of the use of games in the semantics of programming languages with many pointers to Lorenzen games, and intended for a philosophical audience, is Jürjens 2002.

Bibliography

Abramsky, Samson, 1994, "Interaction Categories and Communicating Sequential Processes", in A. William Roscoe (ed.), *A Classical Mind: Essays in Honour of C.A.R. Hoare*, New York: Prentice Hall International, pp. 1–16.

–––, 1996, "Retracing Some Paths in Process Algebra", in U. Montanari & V. Sassone (eds), *CONCUR '96: Concurrency Theory, 7th International Conference*, Springer-Verlag, pp. 1–17. [Abramsky 1996 available online]

–––, 1997, "Semantics of Interaction: An Introduction to Game Semantics", in P. Dybjer & A. Pitts (eds), *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute*, Cambridge University Press, pp. 1–31. [Abramsky 1997 available online]

–––, 2000, "Axioms for Definability and Full Completeness", in G.D. Plotkin, C. Stirling, & M. Tofte (eds), *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, The MIT Press, pp. 55–76. [Abramsky 2000 available online]

Abramsky, Samson & Radha Jagadeesan, 1994, "Games and Full Completeness for Multiplicative Linear Logic", *Journal of Symbolic Logic*, 59: 543–574. [Abramsky and Jagadeesan 1994 available online]

Abramsky, Samson & Guy McCusker, 1999, "Game Semantics", in H. Schwichtenberg & U. Berger

(eds), *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, Berlin: Springer-Verlag, pp. 1–56.

Abramsky, Samson, Radha Jagadeesan, & Pasquale Malacaria, 2000, "Full Abstraction for PCF", *Information and Computation*, 163(2): 409–470. [Abramsky, Jagadeesan, & Malacaria 2000 available online

Amadio, Roberto M. & Pierre-Louis Curien, 1998, *Domains and Lambda-Calculi* (Cambridge Tracts in Theoretical Computer Science, 46), Cambridge: Cambridge University Press.

Barendregt, Henk P., 1984, *The Lambda Calculus, Its Syntax and Semantics*, Amsterdam: North-Holland Co.

Barendregt, Henk P., Wil Dekkers, & Richard Statman, 2013, *Lambda Calculus with Types* (Perspectives in Logic), Cambridge: Cambridge University Press/Association for Symbolic Logic.

Berry, Gérard, 1976, "Bottom-Up Computation of Recursive Programs", *RAIRO Informatique Théorique et Applications*, 10: 47–82. [Berry 1976 available online]

–––, 1978, "Stable Models of Typed λ-Calculi", in Giorgio Ausiello & Corrado Böhm (eds), *Automata, Languages and Programming, Fifth Colloquium* (Lecture Notes in Computer Science, 62), Berlin: Springer-Verlag, pp. 72–89. doi:10.1007/3-540-08860-1_7

Berry, Gérard, Pierre-Louis Curien, & Jean-Jacques Lévy, 1985, " Full Abstraction for Sequential Languages: the State of the Art", Maurice Nivat & John C. Reynolds (eds), *Algebraic Methods in Semantics*, Cambridge: Cambridge University Press, pp. 89–131. [Berry, Curien, & Lévy 1985 available online

Blass, Andreas, 1992, "A Game Semantics for Linear Logic", *Annals of Pure and Applied Logic*, 56(1–3): 183–220. doi:10.1016/0168-0072(92)90073-9

Bucciarelli, Antonio, 1994, "Another Approach to Sequentiality: Kleene's Unimonotone Functions", in Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, & David A. Schmidt (eds), *Mathematical Foundations of Programming Semantics, Proceedings of the 9th International Conference, New Orleans, LA, USA, April 7–10, 1993*, (Lecture Notes in Computer Science, 802), Berlin: Springer, pp. 333–358. doi:10.1007/3-540-58027-1_17

Bucciarelli, Antonio & Thomas Ehrhard, 1994, "Sequentiality in An Extensional Framework", *Information and Computation*, 110(2): 265–296. doi:10.1006/inco.1994.1033

Cardone, Felice & J. Roger Hindley, 2009, "History of Lambda-Calculus and Combinators in the 20th Century", in Dov M. Gabbay & John Woods (eds), *Handbook of the History of Logic. Volume 5. Logic from Russell to Church*, Amsterdam: Elsevier, pp. 723–817.

Cartwright, Robert, Pierre-Louis Curien, & Matthias Felleisen, 1994, "Fully Abstract Semantics for Observably Sequential Languages", *Information and Computation*, 111(2): 297–401. doi:10.1006/inco.1994.1047

Coquand, Thierry, 1995, "A Semantics of Evidence for Classical Arithmetic", *Journal of Symbolic Logic*, 60(1): 325–337. doi:10.2307/2275524

Curien, Pierre-Louis, 1986, *Categorical Combinators, Sequential Algorithms and Functional Programming*, London: Pitman; New York: Wiley.

–––, 1992, "Sequentiality and Full Abstraction", in Fourman, Johnstone, & Pitts 1992: 66–94. doi:10.1017/CBO9780511525902.005

–––, 1994, "On the Symmetry of Sequentiality", in Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, & David Schmidt (eds), *Mathematical Foundations of Programming Semantics, 9th International Conference* (Lecture Notes in Compter Science, 802, Berlin: Springer-Verlag, pp. 29–71. doi:10.1007/3-540-58027-1_2

–––, 2003a, "Symmetry and Interactivity in Programming", *Bulletin of Symbolic Logic*, 9(2): 169–180. [Curien 2003a available online]

–––, 2003b, "Playful, Streamlike Computation", in Guo-Qiang Zhang, J. Lawson, Ying Ming Liu, M.K. Luo (eds.), *Domain Theory, Logic and Computation, Proceedings of the 2nd International Symposium on Domain Theory, Sichuan, China, October 2001* (Semantic Structures in Computation,

3), Dordrecht: Springer Netherlands, pp. 1–24. doi:10.1007/978-94-017-1291-0_1 [Currien 2003b available online]

–––, 2007, "Definability and Full Abstraction", *Electronic Notes in Theoretical Computer Science*, 172(1): 301–310. doi:10.1016/j.entcs.2007.02.011

Fourman, M.P., P.T. Johnstone, & A.M. Pitts (eds), 1992, *Applications of Categories in Computer Science: Proceedings of the London Mathematical Society Symposium, Durham 1991* (London Mathematical Society Lecture Note Series, 177), Cambridge: Cambridge University Press.

Girard, Jean-Yves, 1987, "Linear logic", *Theoretical Computer Science*, 50(1): 1–102. doi:10.1016/0304-3975(87)90045-4

Gunter, Carl A., 1992, *Semantics of Programming Languages. Structures and Techniques*, Cambridge, MA: MIT Press.

Hoare, C.A.R., 1985, *Communicating Sequential Processes*, Englewood Cliffs, NJ: Prentice-Hall.

Hodges, Wilfrid, 2001, "Formal Features of Compositionality", *Journal of Logic, Language and Information*, 10(1): 7–28. doi:10.1023/A:1026502210492

Hyland, J. Martin E., 1997, "Game Semantics", in P. Dybjer & A. Pitts (eds), *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute*, Cambridge: Cambridge University Press, pp. 131–184.

Hyland, J.M.E. & C.-H.L. Ong, 2000, "On Full Abstraction for PCF: I., Models, Observables and the Full Abstraction Problem, II. Dialogue Games and Innocent Strategies, III. a Fully Abstract and Universal Game Model", *Information and Computation*, 163(2): 285–408. doi:10.1006/inco.2000.2917

Janssen, Theo M.V., 2001, "Frege, Contextuality and Compositionality", *Journal of Logic, Language and Information*, 10(1): 115–136. doi:10.1023/A:1026542332224

Joyal, André, 1977, "Remarques sur la Théorie des Jeux à Deux Personnes", *Gazette des Sciences Mathématiques du Quebec*, 1(4).

Jung, Achim & Allen Stoughton, 1993, "Studying the Fully Abstract Model of PCF Within Its Continuous Function Model", in Marc Bezem & Jjan Friso Groote (eds), *Typed Lambda Calculi and Applications, Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, the Netherlands, March 16–18, 1993,* (Lecture Notes on Computer Science, 664), Berlin: Springer-Verlag, pp. 230–244. doi:10.1007/BFb0037109

Jürjens, Jan, 2002, "Games in the Semantics of Programming Languages—An Elementary Introduction", *Synthese*, 133(1–2): 131–158. doi:10.1023/A:1020883810034

Kahn, G. & G.D. Plotkin, 1993, "Concrete Domains", *Theoretical Computer Science*, 121(1–2): 187–277. doi:10.1016/0304-3975(93)90090-G

Kleene, Stephen Cole, 1952, *Introduction to Metamathematics*, New York: Van Nostrand.

Lafont, Yves & Thomas Streicher, 1991, "Games Semantics for Linear Logic", in *Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, Los Alamitos, CA, pp. 43–50. doi:10.1109/LICS.1991.151629

Loader, Ralph, 2001, "Finitary PCF is Not Decidable", *Theoretical Computer Science*, 266(1–2): 341–364. doi:10.1016/S0304-3975(00)00194-8

Longley, John, 2001, "Notions of Computability at Higher Types, I", in René Cori, Alexander Razborov, Stevo Todorcevic, & Carol Wood (eds), *Logic Colloquium 2000* (Lecture Notes in Logic, 19), Wellesley, MA: A.K. Peters, pp. 32–142. [Longley 2001 available online]

Longley, John & Dag Normann, 2015, *Higher-Order Computability*, Heidelberg: Springer.

Lorenzen, P., 1961, "Ein Dialogisches Konstructivitätskriterium", in *Infinitistic Methods, Proceedings of the Symposium on Foundations of Mathematics (1959 : Warsaw, Poland)*, New York: Pergamon Press, pp. 193–200.

Milner, Robin, 1973, *Models for LCF*, Technical report, STAN-CS-73-332, Computer Science Department, Stanford University. [Milner 1973 available online]

–––, 1975, "Processes: a Mathematical Model of Computing Agents", in H.E. Rose & J.C. Shepherdson (eds), *Logic Colloquium '73* (Studies in the Logic and the Foundations of Mathematics, 80),

Amsterdam: North-Holland, pp. 157–174. doi:10.1016/S0049-237X(08)71948-7

–––, 1977, "Fully Abstract Models of Typed λ-Calculi", *Theoretical Computer Science*, 4: 1–22. doi:10.1016/0304-3975(77)90053-6

–––, 1980, *A Calculus of Communicating Systems* (Lecture Notes in Computer Science, 92), Berlin, New York: Springer-Verlag. doi:10.1007/3-540-10235-3

Mitchell, John C., 1993, "On Abstraction and the Expressive Power of Programming Languages", *Science of Computer Programming*, 21(2): 141–163. doi:10.1016/0167-6423(93)90004-9

Mulmuley, Ketan, 1987, *Full Abstraction and Semantic Equivalence*, Cambridge, MA: MIT Press.

Nickau, Hanno, 1994, "Hereditarily Sequential Functionals", in Anil Nerode & Yu V. Matiyasevich (eds), *Logical Foundations of Computer Science: Proceedings of the Third International Symposium, LFCS 1994 St. Petersburg, Russia, July 11–14, 1994* (Lecture Notes in Computer Science, 813), Berlin: Springer-Verlag, pp. 253–264. doi:10.1007/3-540-58140-5_25

Ong, C.-H.L., 1995, "Correspondence Between Operational and Denotational Semantics", in Samson Abramsky, Dov M. Gabbay, & T.S.E. Maibaum (eds), *Handbook of Logic in Computer Science*, vol. 4, Oxford: Oxford University Press, pp. 269–356.

O'Hearn, Peter W. & Jon G. Riecke, 1995, "Kripke Logical Relations and PCF", *Information and Computation*, 120(1): 107–116. doi:10.1006/inco.1995.1103

Paolini, Luca, 2006, "A Stable Programming Language", *Information and Computation*, 204(3): 339–375. doi:10.1016/j.ic.2005.11.002

Plotkin, G.D., 1977, "LCF Considered as a Programming Language", *Theoretical Computer Science*, 5(3): 223–257. doi:10.1016/0304-3975(77)90044-5

Reynolds, John C., 1981, "The Essence of Algol", in J.W. de Bakker & J.C. van Vliet (eds), *Algorithmic Languages: Proceedings of the International Symposium on Algorithmic Languages*, Amsterdam: North Holland, pp. 345–372.

Riecke, Jon G., 1993, "Fully Abstract Translations Between Functional Languages", *Mathematical Structures in Computer Science*, 3(4): 387–415. doi:10.1017/S0960129500000293

Sazonov, Vladimir Yu., 1975, "Sequentially and Parallelly Computable Functionals", in Corrado Böhm (ed.), *λ-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25–27, 1975* (Lecture Notes in Computer Science, 37), Berlin: Springer-Verlag, pp. 312–318.

–––, 1976, "Degrees of Parallelism in Computations", in A.W. Mazurkiewicz (ed.), *Mathematical Foundations of Computer Science 1976, Proceedings of the 5th Symposium, Gdansk, Poland, September 6–10, 1976,* (Lecture Notes in Computer Science, 45), New York: Springer, pp. 517–523.

–––, 2007, "Inductive Definition and Domain Theoretic Properties of Fully Abstract Models for PCF and PCF+", *Logical Methods in Computer Science*, 3(3): paper 7. doi:10.2168/LMCS-3(3:7)2007

–––, 1969, "A Type-Theoretical Alternative to ISWIM, CUCH, OWHY", Informally distributed. Revised and printed 1993, *Theoretical Computer Science*, 121: 411–440. doi:10.1016/0304-3975(93)90095-B

–––, 1970, "Outline of a Mathematical Theory of Computation", in *Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems*, Dept. of Electrical Engineering, Princeton University, pp. 169–176. [Scott 1970 available online]

–––, 1982, "Domains for Denotational Semantics", in M. Nielsen & E. Schmidt (eds), *Automata, Languages and Programming, Ninth International Colloquium* (Lecture Notes in Computer Science, 140), Berlin: Springer-Verlag, pp. 577–613. doi:10.1007/BFb0012801

Scott, Dana S. & Christopher Strachey, 1971, "Toward a Mathematical Semantics for Computer Languages", in Jerome Fox (ed.), *Proceedings of the Symposium on Computers and Automata Held in New York, 13–15 April 1971*, New York: Polytechnic Institute of Brooklyn Press, pp. 19–46. [Scott and Strachey 1971 available online

Sieber, Kurt, 1992, "Reasoning About Sequential Functions via Logical Relations", in Fourman, Johnstone, & Pitts 1992: 258–269. doi:10.1017/CBO9780511525902.015

Stoughton, Allen, 1988, *Fully Abstract Models of Programming Languages*, London: Pitman; New York: Wiley.

Stoy, Joseph E., 1977, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, Cambridge, MA: MIT Press.

Strachey, Christopher, 1966, "Towards a Formal Semantics", in Thomas B. Steel, Jr. (ed.), *Formal Language Description Languages for Computer Programming*, Amsterdam: North-Holland, pp. 198–220.

–––, 1967, "Fundamental Concepts in Programming Languages", International Summer School in Computer Programming, Copenhagen. Revised and reprinted 2000, *Higher-Order and Symbolic Computation*, 13(1–2): 11–49. doi:10.1023/A:1010000313106

Streicher, Thomas, 2006, *Domain-Theoretic Foundations of Functional Programming*, Singapore: World Scientific.

Szabó, Zoltán Gendler, 2013, "Compositionality", in Edward N. Zalta (ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2013.). URL = <https://plato.stanford.edu/archives/fall2013/entries/compositionality/>

Turner, Raymond, 2016, "The Philosophy of Computer Science", in Edward N. Zalta (ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2016.). URL = <https://plato.stanford.edu/archives/spr2016/entries/computer-science/>

Vuillemin, Jean, 1974, "Correct and Optimal Implementations of Recursion in a Simple Programming Language", *Journal of Computer and System Science*, 9(3): 332–354. doi:10.1016/S0022-0000(74)80048-6

White, Graham, 2004, "The Philosophy of Computer Languages", in Luciano Floridi (ed.), *The Blackwell Guide to the Philosophy of Computing and Information*, Malden, MA: Blackwell, pp. 237–247. doi:10.1002/9780470757017.ch18

Academic Tools

- How to cite this entry.
- Preview the PDF version of this entry at the Friends of the SEP Society.
- Look up this entry topic at the Indiana Philosophy Ontology Project (InPhO).
- Enhanced bibliography for this entry at PhilPapers, with links to its database.

Other Internet Resources

- Curien, Pierre-Louis, 2006, "Notes on Game Semantics", course notes
- Lamarche, François, 1992, "Sequentiality, Games and Linear Logic", from a lecture to the CLiCS Symposium Aarhus University, March 23–27, 1992.
- Plotkin, Gordon, 1978, "The Category of Complete Partial Orders: a Tool for Making Meanings", Lectures, summer school, Dipartimento di Informatica, Università di Pisa, Italy. Reprinted in *Domains* (1983)
- Translation of Joyal, André (1977) "Remarks on the Theory of Two-Player Games, translated by Robin Houston, see the posted note about this translation.

Related Entries

compositionality | computer science, philosophy of | logic: and games | logic: dialogical | logic: linear | type theory: intuitionistic

Acknowledgments

I am grateful to Ray Turner for advice and encouragement, and to Luca Paolini for comments on an early draft of this entry.