TU WIEN Informatics

Spielsemantik

Das Abstraktionsproblem für PCF

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Martin Ruiss, BSc
Matrikelnummer 01326774

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Christian Fermüller

Mitwirkung: Dipl.-Ing. Robert Freiman, BSc

Wien, 28. Jänner 2022

Martin Ruiss | Christian Fermüller

TU WIEN Informatics

Game Semantics

The Full Abstraction Problem for PCF

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Martin Ruiss, BSc
Registration Number 01326774

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Christian Fermüller

Assistance: Dipl.-Ing. Robert Freiman, BSc

Vienna, 28th January, 2022

Martin Ruiss	Christian Fermüller

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Martin Ruiss, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. Jänner 2022

Martin Ruiss

Danksagung

Ich danke meinen beiden Betreuern Christian Fermüller und Robert Freiman für die Möglichkeit das Thema nach meinen eigenen Vorstellungen zu erarbeiten. Ich danke auch für ihr ausführliches Feedback, das mir geholfen hat eine zusammenhängendere Arbeit zu schreiben und das Gesamtbild besser zu verstehen.

Kurzfassung

Spielsemantik basiert auf dem Formalismus der dialogischen Logik. Sie war ursprünglich eine Möglichkeit um die Semantik von intuitionistischer Logik und linearer Logik zu definieren. Wenig später erwies sich Spielsemantik als äußerst nützliches Werkzeug zur Interpretation verschiedener Programmiersprachen. Spielsemantik modelliert ein Spiel zwischen einem Spieler und Gegenspieler, der erstere repräsentiert das Programm selbst und der letztere den Kontext des Programms. Das Ziel dieses Modells einer Programmiersprache ist das formale Schließen über Programmiersätze, um unter anderem Compiler Optimierungen zu verbessern.

Diese Arbeit ist, wie auch die meiste Literatur, hauptsächlich auf die funktionale Programmiersprache PCF fokussiert. Es werden einige vielversprechende Ansätze vorgestellt, die nicht auf Spielsemantik basieren und die das Problem der vollständigen Abstraktion nicht zufriedenstellend lösen konnten. Schließlich wird Spielsemantik und deren Anwendung auf PCF erklärt. Außerdem wird ein Ansatz auf dem aktuellen Stand der Entwicklung vorgestellt, der es ermöglicht Programmiersprachen mit Nebenläufigkeit und Wahrscheinlichkeiten zu interpretieren.

Diese Arbeit stellt eine Einführung in das Thema der Spielsemantik und der Interpretation von PCF dar. Mathematische Konzepte die in anderen Publikationen implizit vorausgesetzt sind, werden hier detailliert erklärt.

Abstract

Game semantics, a formalism based on diaogue games used in semantics of intuitionistic logic and later linear logic, have proven to be quite useful in the interpretation of programming languages. They model a play between a player and an opponent, one regarded as the program context and the other as the program itself. The aim of creating such a model is being able to reason about equality of program phrases, in order to improve compiler optimization amongst other applications.

This thesis focuses on the interpretation of the programming language PCF. From the first models based on complete partial orders, to models introducing sequentiality, and finally game semantics, which solve the major problem of full abstraction. Furthermore, we show a current state of the art approach, that extends PCF with probability and provides a fully abstract concurrent game semantics model.

The thesis is aimed at readers with a computer science background familiarizing themselves with semantics of programming languages. It is self-contained, as mathematical backgrounds are explained on which the theory is built on.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

In this thesis we will describe semantical analysis of programming languages, in particular *operational* and *denotational semantics* of the programming language PCF.

Operational semantics mathematically specify a step-by-step execution for a programming language. This definition is syntactic and self-contained, i.e. a programming language is defined in terms of itself, which makes it very accessible and flexible. To a certain extent, it allows to reason about the programming language.

[Car17] define the notation of operational semanctics $e \Downarrow v$ as "e terminates with value v", where e is a program that terminates with v as the result, furthermore we say v is the operational meaning of e. The operational semantics of a program is defined as a partial function $\mathcal{O}$, from a set of programs Prog to values, where $\mathcal{O}(e) = v$ when $e \Downarrow v$.

However, operational semantics have their limits in reasoning about equivalence. Two program phrases e and e' might be equivalent, i.e. $e \Downarrow v$ and $e' \Downarrow v$, but this is not necessarily the case in all contexts. A context $C[\,]$ is just a larger program with a hole in it, that can be filled with a programming phrase. If we fill it with our programming phrases e and e', we get $C[e]$ and $C[e']$ which, in general, do not have to evaluate to the same value.

As reasoning about equivalence is an important feature and the basis of compiler optimization, denotational semantics were introduced, which date back to early analysis of programming languages [Str66]. Denotational semantics are defined by a semantic domain, which can be seen as a metalanguage. It is a function that is defined inductively on the syntax of a programming language, which means one needs to translate a language into that domain. Therefore, denotational semantics are more ambitious than operational semantics and they are not self-contained. But this enables to reason about equivalence

more easily. Equivalence in the syntactic domain becomes equality in the semantic domain, i.e. every term boils down to a mathematical interpretation which can be compared for equality.

The most important property that denotational semantics need to fulfill is *full abstraction*. If two terms are observationally equivalent, i.e. they evaluate to the same result in any context, then their mathematical models must be equal.

It was quickly discovered, that the standard denotational model of even a simple programming language like PCF is not fully abstract. The key to this problem is the definability of the `por`-function (parallel-or), which is shown in Table 3.1. In a fully abstract model, every semantically definable function must be expressible in the programming language. This is not the case as `por` has a semantic definition, but it is not definable in PCF. This issue will be discussed in detail in Section 3.3.

Game semantics, which is a type of denotational semantics, came out of the problem of definability of PCF. It solved the problem of definability for the parallel-or function in the semantic domain, that did not have a counterpart in operational semantics. This new approach introduced a paradigm shift from semantics that view functions in a program as mathematical functions to semantics that view functions as processes, interpreting them as sequences of interaction.

As it was stated by Dan Ghica [ghib]:

> What makes gs (game semantics) distinctive is the fact that its semantic domain abandons the sets-and-functions paradigm which dominates early denotational semantics. It is a genuinely new idea, using concrete combinatorial structures of actions called "games" and "strategies". It is perhaps easy to overlook how important this shift was, since it did away with any pretense that a programming language function is somehow like a mathematical function.

The study of game semantics lies in the intersection of the fields of logic and game theory [Hed19]. One general definition of game theory was given as follows [Mye97]:

> Game theory can be defined as the study of mathematical models of conflict and cooperation between intelligent decision-makers.

The idea of game semantics stems from turn-based games for two players, the opponent and the proponent, e.g. Chess, Checkers, or Go. More particularly, game semantics model a pattern of calls and returns.

A basic introduction to game semantics, similar to ours, was given by Dan Ghica in *Oregon Programming Languages Summer School* [ghib, ghia].

1.2 Structure of the Work

This work is structured into three main chapters.

In Chapter 2 we will give an introduction to fundamental concepts and ideas preceding game semantics. In Section 2.1 we give more detailed definitions of the general ideas that were introduced in Section 1.1, in particular semantics, compositionality, and full abstraction. In Section 2.2 we define the programming language PCF. Section 2.3 presents its operational semantics, that is defined closely intertwined with its syntax. We will then discuss some intuitive requirements for denotational models in Section 2.4, and we will present definitions based on the relation between operational and denotational semantics in Section 2.5.

In Chapter 3, we will show the first major approach of creating a denotational model of PCF with semantics that are separated from the operational evaluation. We start with the first idea of using mathematical structures and functions to model program phrases and computation, and we continue with some more adequate structures. In Section 3.3, we will explain why the model was not sufficient enough to be fully abstract.

We will then move to category theory and categorical semantics in Chapter 4, which is the mathematical foundation for further semantics. Especially the relation between the λ-calculus and some types of categories is signicant and will be presented in Section 4.2.

Finally, in Chapter 5 we discuss some of the early attempts to introduce sequentiality in interpretations of PCF. The idea of a model built from data structures and functions is replaced with a structure modelling sequential execution steps. We will show why each of these attempts did not achieve the desired outcome.

In Chapter 6 we will give an informal idea of Game Semantics in general, followed by mathematical definitions and the proof idea for the fully abstract model of PCF by Hyland and Ong [HO00], and Abramsky, Jagadeesan, and Malacaria [AJM00].

In Chapter 7 we will show some recent ideas in the field of game semantics. The main approach we will follow, does not only extend PCF with concurrency and probability, but also provides a completely new structure for games and strategies, that can be extended in many ways.

CHAPTER 2

Background

2.1 Compositionality and Full Abstraction

As mentioned in the introduction, the two main ways to assign meaning to a program discussed in this thesis are operational and denotational semantics. We have already defined the operational notation $e \Downarrow v$, where e is a program that terminates and returns the value v as the result. The partial function $\mathcal{O}$ is defined as $\mathcal{O}(e) = v$ when $e \Downarrow v$. This notation of operational semantics was given by [Car17], whose definitions we will follow in this chapter.

The main goal of denotational semantics is to provide a *compositional* interpretation $\mathcal{M} : L \to D$, where L is a set of program phrases and D is a set of abstract mathematical structures, so-called domains.

Compositionality enables us to calculate the meaning of a program phrase by the meaning of the constituents of that phrase. Therefore it is an important feature for semantical analysis of programming languages. A compositional interpretation of a programming language can be seen as a homomorphism, where each semantical operation is assigned an operation on denotations. A more detailed definition of the homomorphism and further details about compositionality are given in [Sza20].

An example can be given in a simple imperative language [Car17], where each program $\mathtt{c}$ expresses a state transformation $\mathcal{M}(c) : \Sigma \to \Sigma$. One of the operations of that language is *sequential composition*. From two phrases $\mathtt{c_1}$ and $\mathtt{c_2}$, we build a phrase $\mathtt{c_1}; \mathtt{c_2}$. The composition is executed as follows: We start from a state σ and execute $\mathtt{c_1}$, if $\mathtt{c_1}$ terminates, we get to state σ' and execute $\mathtt{c_2}$. If $\mathtt{c_1}$ terminates, we get to the final state σ''. Alternatively, it is also possible to execute $\mathtt{c_1}; \mathtt{c_2}$ directly, reaching state σ''. In the semantic domain the operation translates to composition of functions over the domain $\Sigma \to \Sigma$. The equivalence over the semantic and syntacic domain of this operation is given

as follows:

$$\mathcal{M}(\mathsf{c}_1; \mathsf{c}_2) = \mathcal{M}(\mathsf{c}_2) \circ \mathcal{M}(\mathsf{c}_1).$$

To include the state σ into the definition, we can write:

$$\mathcal{M}(\mathsf{c}_1; \mathsf{c}_2)(\sigma) = \mathcal{M}(\mathsf{c}_2)(\mathcal{M}(\mathsf{c}_1)(\sigma)).$$

If an interpretation of a programming language L exists, we can define equivalence:

Definition 2.1.1 (Denotational equivalence [Car17])**.** Given any two program phrases e, e', they are *denotationally equivalent* when $\mathcal{M}(e) = \mathcal{M}(e')$, written as $e \simeq_{\mathcal{M}} e'$.

We mentioned the general idea of *contexts* in the introduction. More formally, it can be expressed by a parameter X:

Definition 2.1.2 (Context [Ong95])**.** To create a program phrase $C[M]$, we start from a phrase $C[X]$, where X stands for a parameter of type σ. By substituting every occurrence of X with the phrase M of the same type we get $C[M]$. Often X is left out, resulting in the notation $C[\ ]$.

If $\mathcal{M}$ is compositional, then the equivalence $\simeq_{\mathcal{M}}$ holds for contexts. We will not need this in our thesis, but the interested reader can find further information in [Sza20]. The congruence can be described more formally as follows:

Proposition 2.1.1 ([Car17])**.** If $\mathcal{M}$ is compositional, then for all phrases e, e' and all contexts $C[\ ]$:

$$e \simeq_{\mathcal{M}} e' \Rightarrow C[e] \simeq_{\mathcal{M}} C[e']$$

Furthermore, the congruence over all contexts is more expressive than the naive form of *operational equivalence*. We write $e \sim e'$ if and only if $\mathcal{O}(e) = \mathcal{O}(e')$. This definition is usually not powerful enough to reason about equivalence. To compare operational and denotational congruence, we can define a more precise notion of equivalence, using the concept of contexts $C[\ ]$ once again.

Definition 2.1.3 (Observational equivalence [Car17])**.** Given two program phrases e, e', they are observationally equivalent, denoted by $e \simeq_{\mathcal{O}} e'$, when the following holds for all program contexts $C[\ ]$ and all program values v:

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

where $C[e] \Downarrow v$ denotes that e in the context $C[\ ]$ evaluates to v. The notation $e \Downarrow v$ is described more precisely in Section 2.3.

A more detailed description of operational equivalence from the viewpoint of intelligent agents is given by [Mil75] and is also summarized in [Car17].

While observational equivalence does not take the inner details of computation into account, the denotational model is compositionally build from its smaller parts. Consequently, denotational equivalence strongly depends on inner details. The purpose of *full abstraction* is to reflect those dual perspecives.

Definition 2.1.4 (Full abstraction [Car17])**.** 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.

Full abstraction is similar to the *completeness* property, in the sense that if phrases are observationally equivalent, then they also must be denotationally equivalent. Conversely, if phrases are not denotationally equivalent, then they are not interchangeable in every program context.

Computational adequacy is a weaker form of full abstraction, but it is a useful intermediate step towards full abstraction.

Definition 2.1.5 (Computational adequacy [Car17])**.** 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

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

The relation to full abstraction can be shown easier by an equivalent formulation of computational adequacy [Car17]:

Proposition 2.1.2 ([Car17])**.** 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 $e, e' \in \mathsf{Prog}$,

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

2.2 Syntax of PCF

While full abstraction is quite easy to define, it is not as easy to find fully abstract models of programming languages. This thesis and many other sources [Car17, Mil77, AJM00, HO00, Paq20] focus on the language PCF (Programming language for Computable Functions [Plo77]), which is based on λ-calculus. λ-calculus was introduced by Alonzo Church [Chu41], a good tutorial is given by [Roj15]. It was further extended by Logic for

Computable Functions [Sco69] and finally PCF was introduced [Plo77]. Essentially PCF is λ-calculus with ground types, a fixed-point operator, and further simple operators such as the conditional. The significance of this language originates from its rather simple syntax, which still includes features that need to be dealt with in semantical analysis of any programming language: higher-order functions, types, recursion, and reduction rules for experimenting with several evaluation strategies. In this section we will introduce the syntax of PCF, following the introduction of [Car17], but using a Haskell-style syntax, which was introduced in [Dyb04].

As already mentioned, PCF is based on simply typed λ-calculus. The type system is defined with boolean and arithmetic primitives:

Definition 2.2.1 (PCF types [Car17])**.** The set Types of types of PCF is defined inductively by the context-free grammar

$$\text{<type>} ::= \texttt{Bool} \mid \texttt{Nat} \mid (\text{<type>} \to \text{<type>}).$$

The types `Bool` and `Nat` are called *ground* types.

We will exclude parentheses when possible, assuming that function types associate to the right, i.e. $\texttt{a}_1 \to \ldots \to \texttt{a}_n \to \texttt{t}$ is equivalent to $(\texttt{a}_1 \to (\texttt{a}_2 \to (\ \ldots \to (\texttt{a}_n \to \texttt{t})\ldots)$.

PCF terms are defined by the following context-free grammar, which also includes all terms of simply-typed λ-calculus:

```
<term> ::= <num> | <bool> |
           if <bool> <bool> <bool> | if <bool> <num> <num> |
           (\<var> -> <term>) | isZero <num> | fix <term>

<num> ::= 0 :: Nat | succ <num> | pred <num>

<bool> ::= True :: Bool | False :: Bool

<var> ::= x | y | z | ...
```

This grammar also includes the Haskell-style λ-Notation `(\x -> e)`.

Furthermore, we have the following constants that extend simply typed λ-calculus:

- a constant `0` of type `Nat`
- constants `True` and `False` of type `Bool`
- constant `pred` and `succ` functions of type (`Nat` $\to$ `Nat`), defining the predecessor and successor respectively

- contant if-functions of type (`Bool` → `Bool` → `Bool` → `Bool`) and (`Bool` → `Nat` → `Nat` → `Nat`)
- a constant function `isZero` to verify if a number is zero; it is of type (`Nat` → `Bool`)
- the fixed-point combinator is denoted by `fix`, which returns type `t` for any input type (`t` → `t`), i.e. it has the type (`t` → `t`) → `t`. The fixed point combinator is also often denoted by the symbol $Y(\cdot)$.

We will use λ-expressions with multiple input parameters `\x1 x2 ... xn -> e` as defined in Haskell, which is just syntactic sugar for `\x1 -> ( \x2 -> (\... (\xn -> e) ... )`.

Terms are built inductively according to *judgements*, denoted by

$$B \vdash \texttt{e} :: \sigma.$$

Any free variables $\texttt{x}_1, ..., \texttt{x}_k$ in the term `e` of type σ are assigned a unique type in the *basis* B, where

$$B = \{\texttt{x}_1 :: \sigma_1, ..., \texttt{x}_k :: \sigma_k\}.$$

To maintain congruent types, PCF-terms are built according to inference rules for the aforementioned type judgements. First, any basis B contains constants (where $n \in \mathbb{N}$):

$$B \vdash \texttt{True} :: \texttt{Bool} \qquad B \vdash \texttt{False} :: \texttt{Bool} \qquad B \vdash n :: \texttt{Nat}$$

As well as rules for typed constants, e.g. the judgement for `isZero` in any basis B: $B \vdash$ `isZero` :: `Nat -> Bool`. Furthermore, there are type judgements for building more complex terms:

$$\frac{}{\texttt{x}_1 :: A_1, ..., \texttt{x}_m :: A_m \vdash \texttt{x}_i :: A_i} \ (1) \qquad \frac{B \vdash \texttt{e}_1 :: \sigma \to \tau \qquad B \vdash \texttt{e}_2 :: \sigma}{B \vdash \texttt{e}_1\ \texttt{e}_2 :: \tau} \ (2)$$

$$\frac{B, \texttt{x} :: \sigma \vdash \texttt{e} :: \tau}{B \vdash \texttt{(\x -> e)} :: (\sigma \to \tau)} \ (3) \qquad \frac{B \vdash \texttt{e} :: \sigma \to \sigma}{B \vdash \texttt{fix e} :: \sigma} \ (4)$$

Rule (1) indicates that a variable $\texttt{x}_i$ of type A_i in the basis has the same type in the PCF-term, where $1 \leq i \leq m$. If we have a λ-term $\texttt{e}_1$ of type (σ `->` τ) and another term $\texttt{e}_2$ of type σ under the same basis B, then the term $\texttt{e}_1\ \texttt{e}_2$ is of type τ. This is shown in rule (2). Rule (3) shows that we can use a variable `x` from the basis as a parameter for a lambda term, removing `x` from the basis. The free variable `x` in `e` becomes a bound variable. Rule (4) allows us to infer a fixed point of type σ from a term `e` of type (σ `->` σ).

Examples of how to apply type judgements in practice have been shown in [Coc16], e.g. for the fixed-point combinator:

$$\dfrac{\dfrac{\dfrac{}{\texttt{x} :: \sigma \to \sigma \vdash \texttt{x} :: \sigma \to \sigma}\,(1)}{\texttt{x} :: \sigma \to \sigma \vdash \texttt{fix x}}\,(4)}{\vdash \texttt{\textbackslash x -> fix x} :: (\sigma \to \sigma) \to \sigma}\,(3)$$

2.3 Operational Semantics

In this section, we will once again follow mainly the explanations of [Car17]. Similar to the general definition of operational semantics, we specify the evaluation relation $e \Downarrow v$ for PCF programs, where e are closed[1] terms and v are values having the structure `\x` :: σ -> e. We will continue to use this notation to indicate that the parameter `x` of the lambda-term is of type σ. Intuitively, we already introduced that $e \Downarrow v$ denotes that the evaluation of term e terminates and returns the value v. We will now define this $\Downarrow$-relation more formally with a set of logical inference rules.

We start with the values of the ground types `Bool` and `Nat`. Values of `Bool` are `True` and `False`. And values of `Nat` are all natural numbers $\mathbb{N}$, that can essentially be built by a constant `0` and by the successor function `succ` as follows:

$$\texttt{n} = \underbrace{\texttt{succ succ} \ldots \texttt{succ}}_{n}\,\texttt{0}$$

As previously defined, we can omit brackets, and interpret the functions as right associative.

The evaluation of terms is defined by inference rules. According to the structure of a term e and value v of our known form $e \Downarrow v$, we can deduce some other form $e' \Downarrow v'$. Axioms have the form $v \Downarrow v$ for every value v. The first inference rule we introduce is denoted as follows:

$$\dfrac{e \Downarrow v}{\texttt{succ}\ e \Downarrow \texttt{succ}\ v}\ \text{Successor 1}$$

This means, that if e evaluates to v, then we can deduce, that `succ` e evaluates to `succ` v.

We continue with an inference rule for the application operation of a term $e_1\ e_2$. If the first term e_1 terminates and returns a value v' and we apply e_2 to v' and get a result value, i.e. $v'\ e_2$ evaluates to v, then we can deduce that $e_1\ e_2$ must evaluate to v. It is formally specified as follows:

$$\dfrac{e_1 \Downarrow v' \qquad v'\ e_2 \Downarrow v}{e_1\ e_2 \Downarrow v}\ \text{Application 1}$$

We need a second inference rule for applications. When we apply a value e_2 to a term `\x` :: σ -> e_1, we obtain a term $e_1[e_2/x]$, where every free occurrence of x in e_1 is replaced by e_2:

[1] A closed term e is a term that does not contain free variables.

$$\frac{e_1[e_2/x] \Downarrow v}{(\texttt{\textbackslash x} :: \sigma \texttt{ -> } e_1)e_2 \Downarrow v}\ \text{Application 2}$$

We need to consider again, that a value is not returned in the general case, but only if the computation terminates, but the value is only called when it is needed in the outer function $(\texttt{\textbackslash x} :: \sigma \texttt{ -> } e_1)$. This is a *call-by-name* evaluation strategy. The argument of a function is inserted into the function and left to be evaluated when it appears inside the function. This is in contrast to *call-by-value*, where a parameter is first evaluated and the resulting value is passed to the outer function [Plo75].

To encode recursion, we need the fixed point combinator `fix`. It is the only rule whose premise contains a larger term than the resulting term in the consequence. The evaluation is described as follows:

$$\frac{\texttt{e fix e} \Downarrow v}{\texttt{fix e} \Downarrow v}\ \text{Fix}$$

Additional definitions for further PCF operations were given by [HO00]:

$$\frac{s \Downarrow \texttt{True} \qquad u \Downarrow v}{\texttt{if } s\ u\ u' \Downarrow v}\ \text{If 1} \qquad \frac{e \Downarrow \texttt{succ } v}{\texttt{pred } e \Downarrow v}\ \text{Successor 2} \qquad \frac{e \Downarrow 0}{\texttt{isZero } e \Downarrow \texttt{True}}\ \text{Zero 1}$$

$$\frac{s \Downarrow \texttt{False} \qquad u' \Downarrow v}{\texttt{if } s\ u\ u' \Downarrow v}\ \text{If 2} \qquad \frac{e \Downarrow 0}{\texttt{pred } e \Downarrow 0}\ \text{Predecessor} \qquad \frac{e \Downarrow \texttt{succ } v}{\texttt{isZero } e \Downarrow \texttt{False}}\ \text{Zero 2}$$

To get a better understanding of the different kinds of semantics, we will introduce a practical example of a program adding two natural numbers, and we will show the semantics of that program in each relevant section. Starting with the implementation of the program, we followed the examples given in [Coc16], but adjusting them to our syntax. The definition of the addition is shown in Listing 2.1.

Listing 2.1: Addition of two numbers in PCF.

```
\f n m → if (isZero n) m (f (pred n) (succ m))
```

To calculate an addition, the fixed-point operator `fix` is used. First we show that the rule

$$\frac{\texttt{(e fix e) u}_1\ \texttt{u}_2 \Downarrow \texttt{v}}{\texttt{(fix e) u}_1\ \texttt{u}_2 \Downarrow \texttt{v}}\ \text{Fix'}$$

is derivable:

$$\dfrac{\dfrac{\dfrac{\texttt{e fix e} \Downarrow \texttt{e fix e}}{\texttt{fix e} \Downarrow \texttt{e fix e}}\ \text{Fix 1} \qquad \texttt{(e fix e) u}_1 \Downarrow \texttt{(e fix e) u}_1}{\texttt{(fix e) u}_1 \Downarrow \texttt{(e fix e) u}_1}\ \text{Application 1} \qquad \texttt{(e fix e) u}_1\ \texttt{u}_2 \Downarrow \texttt{v}}{\texttt{(fix e) u}_1\ \texttt{u}_2 \Downarrow \texttt{v}}\ \text{Application 1}$$

Now let Φ := `\f n m -> if isZero(n) m (f (pred n) (succ m))` and `add` := `fix` Φ. Note that Φ is of type `(Nat -> (Nat -> Nat)) -> (Nat -> (Nat -> Nat))` and `add` is of type `(Nat) -> (Nat -> Nat)`. We show how to compute the addition of 1 and 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{1} \Downarrow \texttt{succ 0}}{\texttt{isZero 1} \Downarrow \texttt{False}}\,\text{Zero 2} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{1} \Downarrow \texttt{succ 0}}{\texttt{pred 1} \Downarrow \texttt{0}}\,\text{Successor 2} \qquad \texttt{succ 2} \Downarrow \texttt{3}}{\texttt{if (isZero (pred 1)) (succ 2) ((fix }\Phi\texttt{) (pred pred 1) (succ succ 2))} \Downarrow \texttt{3}}\,\text{If 1}}{\texttt{(}\backslash\texttt{n m -> if (isZero n) m ((fix }\Phi\texttt{) (pred n) (succ m))) (pred 1) (succ 2)}}\,\text{Application 2 x2}}{\texttt{(}\Phi\texttt{ fix }\Phi\texttt{) (pred 1) (succ 2)} \Downarrow \texttt{3}}\,\text{Application 2}}{\texttt{(fix }\Phi\texttt{) (pred 1) (succ 2)} \Downarrow \texttt{3}}\,\text{Fix'}}{\texttt{if (isZero 1) 2 ((fix }\Phi\texttt{) (pred 1) (succ 2))} \Downarrow \texttt{3}}\,\text{If 2}}{\backslash\texttt{n m -> if (isZero n) m ((fix }\Phi\texttt{) (pred n) (succ m))} \Downarrow \texttt{3}}\,\text{Application 2 x2}}{\texttt{(}\Phi\texttt{ fix }\Phi\texttt{) 1 2} \Downarrow \texttt{3}}\,\text{Application 2}}{\texttt{add 1 2} \Downarrow \texttt{3}}\,\text{Fix'}$$

2.3.1 Divergence

In other situations, a term e might not evaluate to a value. In that instance, we say that e diverges and we write $e \Uparrow$. The first example of a divergent term is an infinite recursion:

Definition 2.3.1 (Undefined [Car17]). For any ground type γ, define $\Omega :: \gamma$ as

$$\texttt{fix (}\backslash\texttt{x} :: \gamma \texttt{ -> x)}$$

If we have a look at the available evaluation rules (from bottom up), we see that they can only be applied indefinitely. In the first step, the only available rule in the first case is "Fix", in the second step we have a choice between "Application 1" and "Application 2". When we use "Application 1" we end up with the initial term and do not progress:

$$\dfrac{\dfrac{\dfrac{\dots}{\texttt{fix (}\backslash\texttt{x} :: \gamma\texttt{ -> x)} \Downarrow v}}{\texttt{(}\backslash\texttt{x} :: \gamma\texttt{ -> x) fix (}\backslash\texttt{x} :: \gamma\texttt{ -> x)} \Downarrow v}\,\text{Application 1}}{\texttt{fix (}\backslash\texttt{x} :: \gamma\texttt{ -> x)} \Downarrow v}\,\text{Fix}$$

When using "Application 2" in the second step, we must chose some term v', but given the left branch, the term `(\x ::` γ `-> x)` must evaluate to v'. Since we do not know a further evaluation step, we have no other choice than v' being the same term `(\x ::` γ `-> x)`. As a result, we cannot progress in the right branch:

$$\dfrac{\dfrac{\texttt{(}\backslash\texttt{x} :: \gamma\texttt{ -> x)} \Downarrow v' \qquad \dfrac{\dots}{v'\texttt{ fix (}\backslash\texttt{x} :: \gamma\texttt{ -> x)} \Downarrow v}}{\texttt{(}\backslash\texttt{x} :: \gamma\texttt{ -> x) fix (}\backslash\texttt{x} :: \gamma\texttt{ -> x)} \Downarrow v}\,\text{Application 2}}{\texttt{fix (}\backslash\texttt{x} :: \gamma\texttt{ -> x)} \Downarrow v}\,\text{Fix}$$

We cannot reach any other terms than `fix (\x :: γ -> x)` and `(\x :: γ -> x) fix (\x :: γ -> x)`, resulting in infinite evaluation steps. Therefore, Ω does not change and we write Ω ⇑.

From the previously defined rules we can easily introduce further boolean operations like `and` and `or` by the conditional operator:

1. `and x y = \x :: Bool, y :: Bool -> if x y False`
2. `or x y = \x :: Bool, y :: Bool -> if x True y`

But these definitions do not take the possibility of divergence into account. For a term like `and True` Ω, we need to introduce another boolean value. We denote the value "undefined" as ⊥, which is the result value of Ω. Normally, the first argument of the operator `and` would be evaluated first, and if it does not terminate, then outer term would also not terminate. However, there is also a possibility of evaluating operations in parallel. We consider the operator **pand**, whose evaluation is given in Table 2.1.

Table 2.1: The truth table for the operator **pand**.

pand	True	False	⊥
True	True	False	⊥
False	False	False	False
⊥	⊥	False	⊥

The result for *parrallel-and* **pand** `False` Ω = **pand** Ω `False` = `False` indicates that it will terminate and return a result when either one of both arguments terminates. The *parallel-or*-operator can be defined analogously. This operator is quite relevant, because it cannot be modelled by PCF, as it is a sequential language, but it is non-trivial to find a semantics that properly deals with this case. To analyze PCF semantically, data types containing the undefined element were introduced. Computation steps are modelled functions between data types. This was first introduced in the theory of domains [Sco69].

2.4 Requirements for Denotational Semantics

How does a denotational model for PCF look like? It should be a mathematical structure, that is abstract enough to allow exact reasoning within the model, but expressive enough such that all semantical features are represented properly. First, we need some kind of objects to model types. Each type is assigned an object, this includes the ground types `Bool` and `Nat` as well as higher types of the form σ -> τ. Secondly, we need some operation or function that models terms and the operation of applying a parameter to a term. Finally, we need to find an interpretation for the remaining constant PCF operations such as successor-function, the if-function, and the fixed-point combinator. In

particular, the fixed-point combinator requires some non-trivial work to be represented reasonably. Some further requirements for models of PCF are discussed in Subsection 3.2.2 and Section 4.3.

We have already described compositionality, as this is one of the most important requirements. The meaning of a composed program phrase is inductively defined by the meaning of its subphrases. Particularly in PCF, this means that if σ is a type and τ is a type, then σ -> τ must be a type as well.

In the first attempt of creating a fully abstract model in Chapter 3, the objects modelling types are represented by domains, which are more precisely lattices and later complete partial orders. The operations are represented by continuous functions between these domains.

Later, category theory is used as the basis for a model. There are several advantages of using cartesian closed categories (ccc) for modelling PCF. Types in PCF can be represented by objects and terms can be represented by arrows between these objects. A result by [Lam86] supports this corresponcence: Typed lambda calculus, and cartesian closed categories are essentially equivalent. This is explained in more detail in [Awo10]. PCF is an extension to typed lambda calulus, i.e. any program written in typed lambda calculus is a program of PCF. The key advantage of cccs is that they are closed under function space construction [Ong95]. Intuitively, the exponential of a ccc guarantees that if σ is an object and τ is an object, then σ -> τ is an object. There is a clear correspondence to compositionality. The difficulty of category theory lies in describing those objects concretely, as it is quite abstract by itself. Also, proving that a model is a ccc alone is not enough, we still need to take the other constructs into consideration, such as the fixed-point operator. Although category theory was not considered for the model based on domains, it was proven later that it in fact is a ccc.

Intuitively, the difficulty is to find an appropriate model that is expressive enough to model all terms of the language, but still does not contain "more" than that. In particular, the latter causes some models to fail. The model of Scott Domains fails because there are parallel operations that can be represented within the model, but cannot be defined in PCF, because it is a sequential language. PCF is sequential in the sense that its interpreter may be a "sequential" program. That does not mean that a parallel evaluation is not possible, but that a natural sequential evaluation exists [BCL85]. As a reaction, the search for a fully abstract model continued with sequential semantics, but it turns out, it is not as simple to define sequentiality for higher types as it may seem. We will discuss this in further detail in Chapter 5.

Finally, game semantics introduced some new features, that allowed to model PCF, while still maintaining the frame of category theory. The most important change was the distinction between a program and an environment, also described by player and opponent.

2.5 Relating Operational and Denotational Semantics

We introduced Observational Equivalence already in Definition 2.1.3 for the general case. For full abstraction of PCF, we introduce Observational preorder and equivalence:

Definition 2.5.1 (Observational preorder and equivalence [Car17])**.** Given PCF terms e and e' of the same type σ, we write $e \precsim_{obs} e'$ (meaning e is *observationally less defined than* e') if, for every program context $C[\,]$ with a hole of type σ (see Definition 2.1.2) and any value v,

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

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

The denotational side of it is defined as follows:

Definition 2.5.2 (Denotational preorder and equivalence [Car17])**.** Given PCF terms e and e' of the same type σ relative to a basis B, we write

$$e \precsim_{den} e' \text{ if } [\![e]\!]\rho \sqsubseteq [\![e']\!]\rho$$

for all environments ρ respecting B. We write $e \simeq_{den} e'$ if $e \precsim_{den} e'$ and $e \precsim_{den} e'$

To sum it up informally [Ong95], observational equivalence states that two equivalent program phrases can be swapped without affecting the outcome of the outer context. Observational equivalence is a congruence. In contrast, in denotational semantics, the meaning of a program is what the semantic function denotes. Denotationally equivalent programs are equal if they have the same denotation in the model, which enables to formally reason about equality of program phrases.

A denotational semantics is *adequate* for a programming language if denotational equality implies observational equivalence. If the implication holds in the other direction as well, the denotational semantics is fully abstract for a programming language. There is also a weaker criterion in relation to adequacy: a denotational model is *weakly adequate* if for any program e and any value v, $[\![e]\!] = [\![v]\!]$ if and only if $e \Downarrow v$.

Furthermore, adequacy and weak adequacy are equivalent for the standard model:

Proposition 2.5.1 (Computational adequacy for PCF [Car17])**.** The following statements are equivalent:

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

We will see later how ground types are modelled using domains that contain the undefined value $\bot$. The intuitive interpretation of $\bot$ can be justified by the following corollary.

Corollary 2.5.1 ([Car17])**.** For any closed PCF term e of ground type, $e \Uparrow$ if and only if $[\![e]\!] = \bot$.

However, there is an issue with PCF and full abstraction as we have defined it based on observational equivalence in Section 2.1. We would like to anticipate one major result, that had a great impact on the development of denotational semantics:

Theorem 2.5.1 ([Loa01])**.** Observational equivalence for finitary PCF is not decidable.

Finitary PCF is simply PCF, but containing `Bool` as the only ground type. Clearly, if observational equivalence is not decidable for finitary PCF, it is also not decidable for regular PCF and the definition of full abstraction based on observational equivalence cannot be applied. However, it is possible to define a model that is *intensionally fully abstract*. Its formal definition needs some further preliminary explanations shown in Chapter 3.

CHAPTER 3

Scott Domains

3.1 Domains as Lattices

The original approach of denotational semantics is based on the mathematical theory of computation by Scott [Sco70]. A procedure in the semantic domain is interpreted as a function from the elements of the data type of the input variables to elements of the data type of the output. Using functions and domains has the benefit that the semantical evaluation is independent from regular computation and operational semantics. We will refer to the type of semantics using as lattices as the fundamental structures as $\mathcal{S}$. In this section, we will give some of the fundamental definitions by Scott [Sco70].

First, we define a data type, which is simply a set D of all elements of that type. To refine this simple representation, the elements need to be considered as structured, as elements have certain relations to each other. Suppose two elements $x, y \in D$ are of the same data type, we might say that y is a better version of what x is trying to approximate. The relation is defined as

$$x \sqsubseteq y,$$

which states that y is more accurate than x.

The mathematical background for this type of semantics are lattices and we will see later, that a data type is a lattice. A lattice L is a partially ordered set, where any two elements have a greatest lower bound and a least upper bound. L is complete, when each of its subsets $X \subseteq L$ has a least upper bound and a greatest lower bound. Lattice theory concerns itself with properties of a generally undefined binary relation, that is read as "is contained in" or "is part of" [Bir40].

The ordering based on accuracy is illustrated in [Sco70] with a numerical data type R based on real numbers. Closed intervals $[\underline{x}, \bar{x}] \in R$ of real numbers $\underline{x}$ and $\bar{x}$ form a lattice, together with the elements $\bot$ and $\top$. They are below and above all others respectively,

i.e. $\bot \sqsubseteq [\underline{x}, \bar{x}]$ and $[\underline{x}, \bar{x}] \sqsubseteq \top$ for any interval $[\underline{x}, \bar{x}]$. The ordering of the other elements is defined as

$$[\underline{x}, \bar{x}] \sqsubseteq [\underline{y}, \bar{y}] \text{ iff } \underline{x} \leq \underline{y} \leq \bar{y} \leq \bar{x}.$$

The "perfect" reals are one-point intervals $[\underline{x}, \bar{x}]$ with $\underline{x} = \bar{x}$ and "approximate" reals have $\underline{x} < \bar{x}$

To characterize this approach of denotational semantics there are five axioms.

Axiom 1 ([Sco70])**.** A data type is a partially ordered set.

A partially ordered set is a set with a binary relation "$\sqsubseteq$", that satisfies reflexivity, antisymmetry, and transitivity [Bir40]. Clearly, the data type R is reflexive. It is transitive, because when we have $[\underline{x}, \bar{x}] \sqsubseteq [\underline{y}, \bar{y}]$ and $[\underline{y}, \bar{y}] \sqsubseteq [\underline{z}, \bar{z}]$, then the numbers must be orderered as follows:

$$\underline{x} \leq \underline{y} \leq \underline{z} \leq \bar{z} \leq \bar{y} \leq \bar{x}.$$

It follows that $[\underline{x}, \bar{x}] \sqsubseteq [\underline{z}, \bar{z}]$. R is also antisymmetric, because when $[\underline{x}, \bar{x}] \sqsubseteq [\underline{y}, \bar{y}]$ and $[\underline{y}, \bar{y}] \sqsubseteq [\underline{x}, \bar{x}]$, by definition, $[\underline{x}, \bar{x}] = [\underline{y}, \bar{y}]$.

We continue with the second axiom:

Axiom 2 ([Sco70])**.** Mappings between data types are **monotonic**.

The data types D and D' with the relations $\sqsubseteq$ and $\sqsubseteq'$ respectively form partially ordered sets. The mapping f between data types D and D', $f : D \to D'$ with two elements $x, y \in D$, is monotonic if $x \sqsubseteq y$ implies $f(x) \sqsubseteq' f(y)$. In other words, the more precisely the input for an operation is given, the more accurate the output has to be. Monotonicity in the Scott sense coincides with the standard definition of monotonic funcitons. Especially, all monotonic functions $f : R \to R$ are also monotonic in the Scott sense.

The third axiom specifies the behavior of approximations for applications.

Axiom 3 ([Sco70])**.** A data type is a **complete lattice** under its partial ordering.

We already previously defined lattices. In short, a lattice is a partially ordered set, where any two elements have a greatest lower bound and a lowest upper bound. More precisely, for any two elements x and y there must be a highest element $m = x \sqcap y$ such that $m \sqsubseteq x$ and $m \sqsubseteq y$. For any other element z fulfilling $z \sqsubseteq x$ and $z \sqsubseteq y$, it must hold that $z \sqsubseteq m$. Analogously, there must be a lowest element $M = x \sqcup y$, such that $x \sqsubseteq M$ and $y \sqsubseteq M$, and for any other element z fullfilling $x \sqsubseteq z$ and $y \sqsubseteq z$, it must hold that $M \sqsubseteq z$.

The requirement for bounds is given, because for infinite sequences it is reasonable to assume that the elements might converge to a limit. Assuming, we have a sequence $x_0 \sqsubseteq x_1 \sqsubseteq ... \sqsubseteq x_n \sqsubseteq x_{n+1} \sqsubseteq ...$, we denote the limit by

$$y = \bigsqcup_{n=0}^{\infty} x_n,$$

which also denotes the least upper bound in the sense of the partial ordering $\sqsubseteq$. If the successive terms are containing more and more information, then the limit represents a kind of "union" of all separate contributions.

To guarantee upper and lower bounds for any subset of elements, we assume that the entirety of a set D has an upper bound $\top$, the largest element, and a lower bound $\bot$, the smallest element.

The intuitive meaning of the equation $x \sqcup y = \top$ is that x and y are *inconsistent*. It needs to be distinguished from the term *incomparable*, which is simply $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$. Furthermore, we can read $x \sqcap y = \bot$ as *unconnected*, meaning that there is no "overlap" of information between these elements.

In our data type R, we clearly have a least upper bound and greatest lower bound for any set of elements, as $\top$ and $\bot$ are contained in the type.

The next axiom is again concerned with mappings.

Axiom 4 ([Sco70])**.** Mappings between data types are **continuous**.

To be able to explain this axiom in detail, we need to introduce some notions of topology first. First, we give two general definitions:

Definition 3.1.1 (Topology [Wil12])**.** A topology on a set X, in general, is a collection τ of subsets of X, called the *open sets*, satisfying the following conditions:

1. Any union of elements of τ belongs to τ,
2. any finite intersection of elements of τ belongs to τ,
3. $\emptyset$ and X belong to τ.

Definition 3.1.2 (Continuity [Wil12])**.** Let X and Y be topological spaces and let f be a function $f : X \to Y$. f is *continuous* on $x_0 \in X$ iff for each open set V containing $f(x_0)$ in Y, there is an open set U in X that contains x_0. We say that f is continuous on X iff f is continous on each $x_0 \in X$.

Furthermore, a non-empty subset $X \subseteq D$ is *directed* if every finite subset of X has at least one upper bound in the sense of $\sqsubseteq$ in X. Note that in this approach all subsets are already directed. This is always the case when axiom 3 holds. A function $f : D \to D'$ is *(Scott-)continuous* iff for all directed subsets $X \subseteq D$ we have:

$$f(\bigsqcup X) = \bigsqcup \{f(x) : x \in X\}.$$

How is the theory of topology related to the theory of domains? First of all, any type that satisfies axioms 1 and 3 is already a topological space. Secondly, the *Scott-continuous*

functions betweeen two data types are precisely the continuous functions w.r.t. the Scott-topology [Moo13].

In the case of the data type D, one can check that the following defines a topology on D.

Definition 3.1.3 (Open set for data types [Sco70])**.** A subset $U \subseteq D$ of a data type D is open

1. whenever $x \in U$ and $x \sqsubseteq y$, then $y \in U$; and
2. whenever $X \subseteq D$ is directed and $\bigsqcup X \in U$, then $X \cap U \neq \emptyset$.

For our data type R, a mapping $f : R \to T$, and a non-empty subset $S \subseteq R$, the most precise element $s \in S$ maps to the upper bound of all mapped elements in the domain $\{f(x) : x \in S\}$. This property is provided by the fact that all mappings of R are monotonic.

Finally, the last axiom restricts only data types that can be approximated by finite information, as only those are representable by machines.

Axiom 5 ([Sco70])**.** A data type has an **effectively given** basis.

To be able to explain this statement, we need some preliminary concepts. First, we need some general topological concepts: the interior and the dense subset.

Definition 3.1.4 (Interior [Wil12])**.** If X is a topological space and $E \subset X$, the *interior* of E is

$$\mathrm{Int}(E) = \bigcup\{G \subset E \mid G \text{ is open}\},$$

i.e. it is the union of all open sets of X that are a subset of E.

Definition 3.1.5 (Closed Set [Wil12])**.** If X is a topological space and $E \subset X$, we say E is *closed* iff $X \backslash E$ is open.

Definition 3.1.6 (Closure [Wil12])**.** If X is a topological space and $E \subset X$, the *closure* of E in X is the set

$$Cl(E) = \bigcap\{K \subset X \mid K \text{ is closed and } E \subset K\}.$$

Definition 3.1.7 (Dense Subset [Wil12])**.** A set D is *dense* in a topological space X iff $Cl(D) = X$.

We now apply these general notions to data types. Let $\mathcal{U}(x) = \{x' \in D \mid x \sqsubseteq x'\}$ be the *upper section determined by* x. For $x, y \in D$, $x \prec y$ means that y belongs to the topological interior of $\mathcal{U}(x)$. And we write $x \preccurlyeq y$ to denote that the greatest lower bound of the topological interior of $\mathcal{U}(x)$ is $\sqsubseteq y$. From those notations we have the relationship that $x \prec y$ is weaker than $x \preccurlyeq y$ and $x \preccurlyeq y$ is weaker than $x \sqsubseteq y$.

Furthermore, if every element of a dense subset of a data type is a limit, then we call such a subset a *basis*. This is a familiar notion in topology in general:

Definition 3.1.8 (Topological Basis [Wei])**.** A *basis* for a topology on X is a collection B of subsets of X, with the properties:

1. For each $x \in X$, there is at least one basis element of B containing x.
2. If x belongs to the intersection of two basis elements B_1 and B_2, then there is a basis element B_3 that contains x, such that $B_3 \subset B_1 \cap B_2$.

For example, the set of all open intervals in the real numbers is a basis for the Euclidean topology. In the case of data types, we have a notion that coincides with the general topological notion of a basis. A subset $E \subseteq 2^D$, where 2^D is the powerset of D, is a basis iff it the following conditions hold:

1. whenever $e, e' \in E$, then $e \sqcup e' \in E$; and
2. for all $x \in D$ we have $x = \bigsqcup\{e \in E : e \preccurlyeq x\}$.

"Effectively given" in this context means that the basis E must be known, in particular it must be known how to decide which of the relationships $e \prec e'$, $e \preccurlyeq e'$, and $e \sqsubseteq e'$ are true or false. This requires that the basis E must be at most countably infinite, and that we have an effective enumeration $E = \{e_0, e_1, e_2, ...\}$.

The most important consequence of the assumption of an effectively given basis is that it allows us to define what computable elements are.

Definition 3.1.9 (Computability [Sco70])**.** Let $x \in D$ and E be a basis of D. Then x is computable relative to a basis iff there is an effectively given subsequence

$$\{e'_0, e'_1, e'_2, ...\} \subseteq E$$

such that $e'_n \sqsubseteq e'_{n+1}$ for each n, and

$$x = \bigsqcup_{n=0}^{\infty} e'_n$$

In other words, we must have an effective enumeration towards the element x, that converges a limit x. This notion is essential to identify computable elements of a data type D, as it might contain *uncountably* many elements, but only countably many computable elements.

As a final remark to the structure of Scott domains, it can be seen that axiom 3 implies axiom 1 and axiom 4 implies axiom 2, but the axioms are given in the order of how the ideas naturally occur. To sum up data types in one sentence, they are complete lattices with effectively given bases where all mappings are continuous.

3.2 Domains as Complete Partial Orders

The original idea of formalizing types has been refined in [Sco82]. Replacing the concept of lattices with complete partial orders allows two elements of a data type not having a common least upper bound. In general, the model became more sophisticated and discarded overlapping axioms from the original model. We will only present the most important ideas, which have been summarized in [Car17].

Definition 3.2.1 (Complete partial orders [Car17])**.** A complete partial order (cpo) is a partially ordered set $\langle D, \sqsubseteq \rangle$ with a least element $\bot$, such that every increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq ...$ of elements of D has a least upper bound $\bigsqcup_n x_n$.

For PCF ground types we have the complete partial orders $\texttt{Bool} = \{\texttt{True}, \texttt{False}\}_\bot$ and $\texttt{Nat} = \mathbb{N}_\bot$ as *flat domains*. The ordering of the elements is displayed in Figure 3.1 and 3.2.

A flat domain for any set X, $X_\bot$ is the set by adding the new element $\bot$ to X, $X \cup \{\bot\}$. Elements of $X_\bot$ can be simply ordered according to their information, by setting based on the amount of information:

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

for $x, y \in X_\bot$.

An often needed property on domains is that every element is the limit of its finite approximations. This is made formal in the following definition of finiteness (or compactness). This formulation also coincides with compactness in the topological sense.

Definition 3.2.2 (Finite/compact elements of a cpo [Car17])**.** If D is a cpo, an element $d \in D$ is finite, if for every increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq ...$ the following holds:

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

Finite elements are also called compact.

For the definition of an *algebraic cpo*, we use the Definition 3.1.9 of computability.

Definition 3.2.3 ([Car17])**.** A cpo D is *algebraic* if every $d \in D$ is computable. Furthermore, the finite elements of D form a basis.

We need one more definition to characterize a domain, which is *consistency*.

Definition 3.2.4 ([Car17])**.** Given a cpo D, a subset $X \subseteq D$ that has an upper bound is called *consistent*, denoted by $\uparrow X$. D is *consistently complete* if every subset $X \subseteq D$ has a least upper bound $\uparrow X$. Furthermore, we write $x \uparrow y$ if X is the set $\{x, y\}$.

Finally, we present the definition of a domain, which is the framework we will continue to use to present denotational semantics for PCF.

Definition 3.2.5 (Domain [Car17])**.** A *domain* is a consistently complete algebraic cpo with a countable basis.

As we have previously defined computability, it would be equivalent to say that a domain is a consistently complete computable cpo.

3.2.1 Fixed points

To fully define denotational semantics, we need a fundamental property of continuous functions $D \to D$, the least fixed point. It allows us to calculate recursion, mainly with a finite number of steps. It can be constructed uniformly and continuously:

Theorem 3.2.1 (Fixed Point Theorem of continous functions [Car17])**.** Let $f : D \to D$ be a continuous function and $x \in D$ be such that $x \sqsubseteq f(x)$. The unique element

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

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

Definition 3.2.6 (Least fixed point of a continuous function [Car17])**.** The least fixed point of a continuous function $f : D \to D$ is the element of D defined by

$$\text{lfix}(f) := \bigsqcup_{n \in \mathbb{N}} f^{(n)}(\bot)$$

For example, let us look at the continuous function in pseudo-code:

```
\f -> \x -> if isZero(x) then 0 else x + f(x - 1)
```

We get a least fixed-point dependent on the value of `x`:

- for $f^{(0)}$ we get $\bot$,
- for $f^{(1)}$ we get 0 for `x`$= 0$ and $\bot$ otherwise
- for $f^{(2)}$ we get 1 for `x`$= 1$ and all the previous results,
- in general, $n =$ `x`$+1$ steps are needed.

In summary, the least fixed point lfix is independent from the parameter given to the function, as n ranges over $\mathbb{N}$.

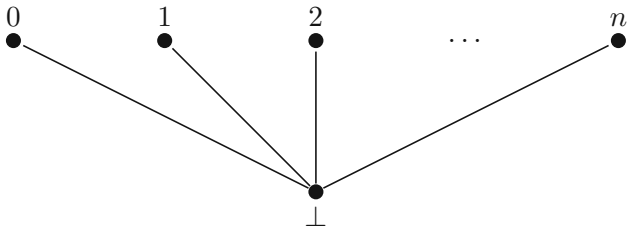


Figure 3.1: The flat cpo of natural numbers [Ong95].

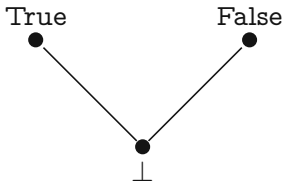


Figure 3.2: The flat cpo of boolean values [Ong95].

3.2.2 Continuous Semantics for PCF

The *standard interpretation* $\mathcal{D}$ of PCF consists of cpos for ground types $D^{\mathtt{Nat}} = \mathbb{N}_\perp$ and $D^{\mathtt{Bool}} = \{\mathtt{True}, \mathtt{False}\}_\perp$, and cpos for higher-order types $D^{\sigma\to\tau} = D^\sigma \to D^\tau$. The cpos for ground types are visualized in Figures 3.1 and 3.2. The constant functions are modelled by continuous functions between cpos according to their type, e.g. `if :: Bool -> Nat -> Nat -> Nat` and `succ :: Nat -> Nat` are interpreted as:

$$\mathtt{if}\ b\ x\ y\ = \begin{cases} x, \text{ if } b = \mathtt{True} \\ y, \text{ if } b = \mathtt{False} \\ \perp, \text{ if } b = \perp \end{cases} \qquad \mathtt{succ}\ n = \begin{cases} \perp, \text{ if } n = \perp \\ n+1, \text{ if } n \in \mathbb{N} \end{cases}$$

The other constants are interpreted similarly, except for the function `fix`, which is modelled by the continuous function lfix$(\cdot)$.

If a PCF term e contains free variables, the types of those variables are given by a basis B. Furthermore, the environment ρ maps free variables $x :: \tau$ in e to an element of D^τ. It is said that "ρ *respects* B", when every free variable is mapped to an element of appropriate type. Of course, we do not need an environment when the term e is closed.

In the interpretation $\mathcal{D}$, each term $e :: \sigma$ with a basis B is assigned to an element $[\![e]\!]\rho \in D^\sigma$, where ρ must be an environment that respects B. The element $[\![e]\!]\rho$ is built by structural induction on terms based on application operations.

In general, an interpretation is represented by a *value domain*, consisting of

- a collection $\{D^\sigma\}$ of cpos, one for each type σ,

- an application operation $D^{\sigma\to\tau} \times D^{\sigma} \to D^{\tau}$ for each pair of types σ and τ. An element $e \in D^{\tau}$ can be applied to $d \in D^{\sigma\to\tau}$, written as $d \cdot e$ or de.
- Constants must be interpreted as expected, i.e. as described in the beginning of this subsection.

If every D^{σ} of an interpretation is a cpo and every $D^{\sigma\to\tau}$ consists of continuous functions $D^{\sigma} \to D^{\tau}$, then the interpretation itself is *continuous*. We say that a model is extensional, when each function type $(\sigma \to \tau)$ has a domain $D^{(\sigma\to\tau)}$ that is a collection of functions from D^{σ} to D^{τ}.

Definition 3.2.7 (Extensional [Car17])**.** A model is *extensional*, if for all elements $f, g \in D^{\sigma\to\tau}$ the following holds:

$$f = g \qquad \text{if and only if} \qquad f \cdot x = g \cdot x$$

for all $x \in D^{\sigma}$.

Definition 3.2.8 (Order-extensional [Car17])**.** A model is *order-extensional*, if for all elements $f, g \in D^{\sigma\to\tau}$ the following holds:

$$f \sqsubseteq g \qquad \text{if and only if} \qquad f \cdot x \sqsubseteq g \cdot x$$

for all $x \in D^{\sigma}$.

There are some criteria, that the standard model as well as all other continuous models need to fulfill. In short, specific identities between terms are expected. We will now describe those identities and further criteria in more detail.

Definition 3.2.9 (Environment [Ong95])**.** The set *Env* is a set of enviroments. Environments are type respecting functions from the set of variables to $\bigcup_{\sigma} D^{\sigma}$, i.e. the union of all domains over all types. The environment is ranged over by the variable ρ, defined as

$$\rho[x^{\sigma} \mapsto d](y) = \begin{cases} d, \text{ if } y = x^{\sigma} \\ \rho(x) \text{ otherwise.} \end{cases}$$

Definition 3.2.10 (Model of PCF [Ong95])**.** A *model* of PCF is a continuous semantic function

$$[\![\ ]\!] : \text{PCF} \to (\mathit{Env} \to \bigcup_{\sigma} D^{\sigma}),$$

that satisfies three sets of criteria:

- compositional semantics
- structural constraints
- operational soundness

The criteria of **compositional semantics** ensures that the following equalities hold. We have $d \in D^\sigma$, terms M and N, variables x^σ, constants c^σ:

$$\begin{aligned}
[\![x^\sigma]\!]\rho &= \rho(x^\sigma) \\
[\![\Omega^\sigma]\!]\rho &= \bot^\sigma \\
[\![c^\sigma]\!]\rho &= c^\sigma \\
[\![M^{\sigma\to\tau} N^\sigma]\!]\rho &= [\![M^{\sigma\to\tau}]\!]\rho \cdot [\![N^\sigma]\!]\rho \\
[\![\texttt{\textbackslash x} :: \sigma \texttt{ -> } M]\!]\rho \cdot d &= [\![M]\!](\rho[x^\sigma \mapsto d]) \\
[\![\texttt{fix}^\sigma M]\!]\rho &= \mathrm{lfix}^\sigma([\![M]\!]\rho)
\end{aligned}$$

Ω^σ is the undefined operation for a type σ, as it has been described in Definition 2.3.1.

Structural constraints ensure that the evaluation function is a well-behaved function of the environment. It should be able to interact with operations of term-substitutions and contextual-substitution. For any terms M and N, and any environment ρ and ρ' the following should hold:

(Env) if $\rho(x^\sigma) = \rho'(x^\sigma)$ for all $x^\sigma \in FV(M)$, then $[\![M]\!]\rho = [\![M]\!]\rho'$

(Subst) $[\![M[N/x^\sigma]]\!]\rho = [\![M]\!](\rho[x^\sigma \mapsto [\![N]\!]\rho])$

(Cont) if $[\![M]\!]\rho = [\![N]\!]\rho$ for all ρ, then for any type-compatible context $C[X]$, we have $[\![C[M]]\!]\rho = [\![C[N]]\!]\rho$ for all ρ

The first constraint **(Env)** states that the valuation of a term should be invariant over environments which agree on all variables occurrring free in a term ($FV(M)$ denoting free variables of M). The second constraint **(Subst)**, together with the compositional semantics constraints, enable β-equivalence, i.e. for any PCF-terms $M :: \tau$ and $N :: \sigma$, and for any environment ρ, the following holds:

$$[\![(\lambda x^\sigma.M)N]\!]\rho = [\![M[N/x^\sigma]]\!]\rho$$

Every occurrence of x^σ in the environment of $[\![M]\!]$ is replaced with $[\![N]\!]$ under the environment ρ by application, while equality is preserved. Finally, the third constraint **(Cont)**, asserts that the valuation of PCF-terms is a context-free operation.

Operational soundness states that whenever a program converges to a value, its denotational semantics coincides with it: If $M \Downarrow N$, then $[\![M]\!]\rho = [\![N]\!]\rho$ for all ρ.

A similar introduction to continuous models has also been shown in [BCL85].

The standard interpretation $\mathcal{D}$ is an order-extensional continuous model of PCF [Plo77, Ong95]. Furthermore, any collection of cpos that make up an order-extensional continuous model are Scott domains, which are closed under function space construction. Together with continous functions they form a cartesian closed category. We will not go into further detail and leave the definitions to be looked up in [Ong95], but we will take up the concept of cartesian closed categories in later chapters.

Example

Suppose we would like to examine the denotational semantics of our program defined in Listing 2.1 for the addition of $1 + 2$. Recall that the standard interpretation $\mathcal{D}$ on types is given as the following domains:

- $\texttt{Nat}^{\mathcal{D}} = D^{\texttt{Nat}} = \mathbb{N}_{\perp}$
- $\texttt{Bool}^{\mathcal{D}} = D^{\texttt{Bool}} = \{\perp, \texttt{True}, \texttt{False}\}$,
- $(\sigma \to \tau)^{\mathcal{D}} = D^{\sigma} \to D^{\tau} = D^{\sigma \to \tau}$

The order on compound types is given by $x \sqsubseteq^{\mathcal{D}}_{\sigma\to\tau} y$ iff for every $d \in \sigma^{\mathcal{D}}$ we have that $x(d) \sqsubseteq^{\mathcal{D}}_{\tau} y(d)$. In particular, the least element $\perp^{\mathcal{D}}_{\sigma\to\tau}$ in $(\sigma \to \tau)^{\mathcal{D}}$ is $\{(d, \perp^{\mathcal{D}}_{\tau}) : d \in \sigma^{\mathcal{D}}\}$.

We assume that the relations for natural numbers $<, >, \leq, \geq$ are only between elements of $\mathbb{N}$, and $\perp$ is ruled out by previous conditions. The element $\perp$ is defined as expected and previously defined in 2.3.1.

By definition $[\![\texttt{add}]\!]_{\mathcal{D}} = [\![\texttt{fix } \Phi]\!]_{\mathcal{D}} = \text{lfix}([\![\Phi]\!]_{\mathcal{D}}) = \bigsqcup_i [\![\Phi]\!]^{(i)}_{\mathcal{D}}(\perp)$, where $\perp$ is the least element in $D^{(\texttt{Nat}\to(\texttt{Nat}\to\texttt{Nat}))\to(\texttt{Nat}\to(\texttt{Nat}\to\texttt{Nat}))}$.

$$\begin{aligned}[\![\Phi]\!]_{\mathcal{D}}(f)(n)(m) &= [\![\texttt{\textbackslash f n m -> if (isZero n) m (f (pred n) (succ m))}]\!]_{\mathcal{D}}(f)(n)(m)\\ &= [\![\texttt{if}]\!]_{\mathcal{D}}([\![\texttt{isZero}]\!]_{\mathcal{D}}(n))(m)(f([\![\texttt{pred}]\!]_{\mathcal{D}}(n))([\![\texttt{succ}]\!]_{\mathcal{D}}(m)))\end{aligned}$$

To compute the recursion, remember, that for $f : D \to D$, we have $f^{(0)}(d) := \perp_D$ and $f^{(i+1)}(d) := f(f(i)(d))$. For simplicity, let us set $\phi_i(n, m) := [\![\Phi]\!]^{(i)}_{\mathcal{D}}(\perp)(n)(m)$ and $n \oplus m := \perp$, if n or m are $\perp$ and else $n \oplus m = n + m$.

$$\phi_0(n, m) = \perp$$

$$\phi_1(n, m) = \begin{cases} \perp, & \text{if } n = \perp \\ m, & \text{if } n = 0 \\ \phi_0(n-1, m+1), & \text{if } n > 0 \end{cases} = \begin{cases} \perp, & \text{if } n > 0 \\ n \oplus m, & \text{if } n = 0 \end{cases}$$

$$\phi_2(n, m) = \begin{cases} \perp, & \text{if } n = \perp \\ m, & \text{if } n = 0 \\ \phi_1(n-1, m+1), & \text{if } n > 0 \end{cases} = \begin{cases} \perp, & \text{if } n = \perp \text{ or } n > 1 \\ m, & \text{if } n = 0 \\ m \oplus 1, & \text{if } n = 1 \end{cases} = \begin{cases} n \oplus m, & \text{if } n \leq 1 \\ \perp, & \text{if } n > 1 \end{cases}$$

$$\vdots$$

$$\phi_i(n, m) = \begin{cases} n \oplus m, & \text{if } n \leq i-1 \\ \perp, & \text{if } n > i \end{cases}$$

We conclude that

$$[\![\texttt{add}]\!]_{\mathcal{D}}(n)(m) = (\bigsqcup_i [\![\Phi]\!]_{\mathcal{D}}^{(i)}(\bot))(n)(m) = n \oplus m.$$

A good example with the calculation of faculty and further detailed explanations on domain theory and fixed-point semantics can be found in [Slo95].

3.3 Full abstraction results

The general notion of full abstraction shown in Definition 2.1.4 is based on the equality of denotational representations of terms, which is also called synonymy. Informally, a model of a programming language is fully abstract, if observational equivalence follows from denotational equivalence and vice versa.

The following proposition was proven quite early by Plotkin:

Proposition 3.3.1 ([Plo77])**.** The standard model $\mathcal{D}$ of PCF is not fully abstract.

We will give the proof by [Plo77], which has been nicely summarized in [Car17]. Another way to prove this was given by [Gun92]. The observation that there are some terms in the denotational model, that cannot be represented syntactically, is the foundation of this proof. This becomes clear considering the "test" terms T_i defined as follows (where $i = 0, 1$):

```
\f :: (Bool -> Bool -> Bool). if (f True ⊥Bool) then
                                  if (f ⊥Bool True) then
                                      if (f False False) then
                                          ⊥Nat
                                      else i
                                  else ⊥Nat
                              else ⊥Nat
```

The function por is defined in Table 3.1. When f is interpreted as por, i.e. $[\![T_0]\!]$por, then we get different results for T_0 and T_1. This can be seen when taking a more detailed look at T_i under the evaluation of f as por:

```
if (por True ⊥Bool) then
        if (por ⊥Bool True) then
                if (por False False) then
                        ⊥Nat
                else i
        else ⊥Nat
else ⊥Nat
```

It follows that $[\![T_0]\!]\texttt{por} = 0 \neq 1 = [\![T_1]\!]\texttt{por}$, and $T_0 \simeq_{den} T_1$ does not hold. On the other hand, no program context in PCF can separate T_0 and T_1. The function por needs to execute two parameters in parallel, because by the truth table, it must terminate if one of the two parameters terminates. However, PCF is a sequential programming language. We will see in Chapter 5 that sequentiality is not as easy to define as it may seem. Intuitively, PCF has a sequential behavior, i.e. its interpreter may be a sequential program [BCL85]. This just means that natural sequential evaluation exists, while a parallel evaluation still may be possible.

Let us assume that we want to calcluate por $\perp$ True. If we execute the first parameter before we execute the second, then the overall computation would not terminate, although the defined outcome is True. In summary, there are parallel functions that can be modeled with domains, but cannot be implemented in PCF. This was first observed by [Sco69]. The Activity Lemma [Plo77] proves that por cannot be defined, by characterizing the dependences of the evaluation process of terms in a combinatorial way.

Table 3.1: The truth table for the operator por.

por	True	False	$\perp$
True	True	True	True
False	True	False	$\perp$
$\perp$	True	$\perp$	$\perp$

[Ong95] has presented two approaches trying to cope with these problems. The first one upgrades the language, and the second one cuts the model down.

The expansive approach, which extends PCF with parallel-or, seems to add very simple operational behavior, but the addition results in a language that is not sequential any more. The new language has enough expressive power such that all elements of the standard model $\mathcal{D}$ are definable in the extended language. And by the Definability Theorem [Plo77], the standard model $\mathcal{D}$ is fully abstract for this new language. The proofs of full abstraction are given in [Ong95].

The restrictive approach [Mul86a, Mul86b] gives a construction based on the proof of existence of a unique fully abstract model identified by [Mil77]. It states that it is possible to create a fully abstract model by using only closed terms and creating a partially ordered set out of them.

Theorem 3.3.1 ([Mil77])**.** There is a unique continuous, order extensional, inequationally fully abstract model of PCF, up to isomorphism.

Milner [Mil77] showed that a unique fully abstract model can be constructed by taking off the top element of the lattice interpretation, that we described at the beginning of this Chapter.

However, both of these models did not give a direct and fully satisfactory description of a fully abstract interpretation of PCF [Car17]. A precise criterion is needed to decide if a description of a model is satisfactory. An example of such a "precise and minimal condition of full abstraction for a semantic solution" was given by [JS93] for a finitary version of PCF, but a direct description of such a model was not given. This was later justified by the following result:

Theorem 3.3.2 ([Loa01])**.** Observational equivalence for finitary PCF is not decidable.

However, there is still a possibility to give a direct description in terms of *intensional full abstraction*.

Definition 3.3.1 (Intensional full abstraction)**.** A model of PCF is *intensionally fully abstract* if every D^σ is algebraic and all its compact elements are definable [Abr14].

Following this direction leads to a sequential semantical characterization and later to game semantics. But before we move on to that topic, we will give an introduction to categorical semantics.

CHAPTER 4

Categorical Semantics

Categorical semantics are an abstraction from standard denotational semantics, and were introduced by [Cro94]. While domains define a mathematical denotation for each element, categorical semantics do not provide a direct model, but an abstract basis with useful properties for creating a model. The former corresponds to a extensional definition, whose objects are enumerated, i.e. the objects of a definition are specified. On the other hand, the latter corresponds to an intensional definition, whose objects are specified by sufficient and necessary conditions.

As we will see in later sections, categories form the basis for sequential and game semantics. Many sources [HO00, Ong95, AM99, Ber78] use this framework to prove certain properties of their model. In particular, finding a Closed Cartesian Category (CCC) is of interest, because it provides properties needed for compositionality.

We will start with an introduction to category theory. We will give the basic ideas of the detailed introduction by [Awo10].

4.1 Category Theory

The following statement gives a good first idea of category theory [Awo10]:

> What is category theory? As a first approximation, one could say that category theory is the mathematical study of (abstract) *algebras of functions*. Just as group theory is the abstraction of the idea of a system of permutations of a set or symmetries of a geometric object, category theory arises from the idea of a system of functions among some objects.

We consider collections of functions and products of those, e.g. $g \circ f$ is the product of functions f and g (depicted in Figure 4.1). The central elements of categories are objects $A, B, C, ...$ and arrows arrows $f : A \to B, g : B \to C, ...$ from one object to another.

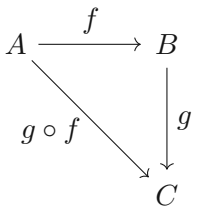


Figure 4.1: The product of functions f and g. [Awo10]

Category theory is quite a universal mathematical language, just as set theory and it reveals certain connections between different fields. For example, the *adjoint functor* in category theory and the existential quantifier in logic are similar concepts.

To fix the idea, let us discuss some notions known from basic set theory. As usual, if f is a function from a set A to a set B, it is denoted as

$$f : A \to B.$$

Explicitly, the regular function definition means that each element of A has a mapping in f that maps to an element in B, i.e.

$$\text{range}(f) \subseteq B.$$

When also a function $g : B \to C$ exists, then the composite function $g \circ f : A \to C$ is given by

$$(g \circ f)(a) = g(f(a)) \qquad a \in A.$$

Function composition "$\circ$" is associative, i.e. if there is another function $h : C \to D$ then we get the following equality (shown in Figure 4.2):

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

Finally, in category theory each set A is completely specified by its identity function

$$1_A : A \to A$$

given by

$$1_A(a) = a.$$

Keeping this intuition in mind, we give the formal definition of a category:

Definition 4.1.1 ([Awo10])**.** A *category* **C** consists of the following data:

- Objects: $A, B, C, ...$

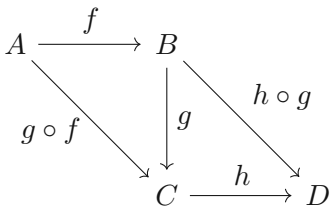


Figure 4.2: Associativity of compositions of functions. [Awo10]

- Arrows (or Morphisms): $f, g, h, ...$
- For each arrow f there are given objects: $dom(f), cod(f)$, called the *domain* and *codomain* of f. For an arrow $f : A \to B$, we have $dom(f) = A$ and $cod(f) = B$.
- Given arrows $f : A \to B$ and $g : B \to C$, with $cod(f) = dom(g)$, there is an arrow $g \circ f : A \to C$ called the *composite* of f and g.
- For each object A, there is an arrow $1_A : A \to A$ called the *identity arrow* of A.
- For simplicity, we write $C \in \mathbf{C}$, when an object C is in the category $\mathbf{C}$, and $f \in \mathbf{C}$, when an arrow is in the category $\mathbf{C}$.
- The collection of objects of $\mathbf{C}$ is sometimes denoted by C_0 and the collection of arrows is denoted by C_1.

Furthermore, the following laws need to be fulfilled:

- Associativity:
$$h \circ (g \circ f) = (h \circ g) \circ f$$
for all $f : A \to B, g : B \to C, h : C \to D$.
- Unit:
$$f \circ 1_A = f = 1_B \circ f$$
for all $f : A \to B$.

Any structure fulfilling the definition with its laws is a category. The objects are not only sets and the arrows are not restricted to functions.

For example, when all sets are objects and all functions between those sets are arrows, they form a category with the regular function composition and identity functions.

A concept quite useful in category theory is an isomorphim. It is the counterpart to the set-theoretic notion of bijection.

Definition 4.1.2 ([Awo10])**.** In any category C, an arrow $f : A \to B$ is called an *isomorphism* if there is an arrow $g : B \to A$ in C such that

$$g \circ f = 1_A \text{ and } f \circ g = 1_B.$$

If there is an isomorphism we write $g = f^{-1}$ or $A \cong B$. The definition of isomorphism is an abstract, category-theoretic definition, because only category theoretic notions are used.

We already mentioned, that arrows are also called morphisms, in fact, they are an abstraction of homomorphisms, which are used in many fields, e.g. in group theory. For this thesis, Hom-sets will be of interest, they are only possible in certain kinds of categories:

Definition 4.1.3 (Locally small categories and hom-sets [Awo10])**.** A category $\mathbf{C}$ is called *locally small* if for all objects A, B in $\mathbf{C}$, the collection

$$Hom(A, B) = \{f \in C_1 \mid f : A \to B\}$$

is a set. It is also called a *hom-set*.

The concepts of *subcategories* are sometimes useful, when talking about semantics:

Definition 4.1.4 (Subcategory and Full Subcategory [Awo10])**.** A category $\mathbf{U}$ is a subcategory of the category $\mathbf{C}$ iff all its arrows are also arrows of $\mathbf{C}$. $\mathbf{U}$ is a full subcategory iff for any pair of objects A and B, $\mathbf{U}$ contains all the arrows between A and B in $\mathbf{C}$.

We will later need to map one category to another one. This is achieved by a functor, the "homomorphism of categories":

Definition 4.1.5 (Functor [Awo10])**.** A *functor*

$$F : \mathbf{C} \to \mathbf{D}$$

maps each object of a category $\mathbf{C}$ to an object of a category $\mathbf{D}$ and each arrow of $\mathbf{C}$ to an arrow of $\mathbf{D}$. Furthermore, the following rules need to be fulfilled:

- $F(f : A \to B) = F(f) : F(A) \to F(B)$
- $F(g \circ f) = F(g) \circ F(f)$
- $F(1_A) = 1_{F(A)}$.

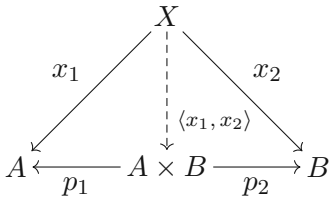


Figure 4.3: The commutative diagram of a product $A \times B$. [Awo10]

4.1.1 Constructions on objects

We will now begin to introduce some constructions on objects of a category that we will need for later definitions. One of the most important constructions in a category is the product.

Definition 4.1.6 ([Awo10])**.** In a category **C**, a binary product of elements A and B is an object $A \times B$, for which the arrows $p_1 : A \times B \to A$ and $p_2 : A \times B \to B$ must exist. For any object X in the same category, with arrows $x_1 : X \to A$ and $x_2 : X \to B$, there must be a unique arrow $u = \langle x_1, x_2 \rangle$, such that $x_1 = p_1 \circ u$ and $x_2 = p_2 \circ u$. This is shown in the commutative diagram in Figure 4.3.

For our example of sets, this corresponds to the cartesian product. For sets A and B, we have the cartesian product $A \times B$ and the function $f_1 : A \times B \to A$ is just a projection to the first parameter of the function. The function for $f_2 : A \times B \to B$ works analogously. For any set X, with functions $x_1 : X \to A$ and $x_2 : X \to B$, we can define a function $x_{12} : X \to A \times B$ as follows

$$x_{12}(z) = \langle x_1(z), x_2(z) \rangle.$$

It is clear that the following equalities must hold:

$$x_1(z) = f_1(x_{12}(z)) = f_1 \circ x_{12}(z),$$
$$x_2(z) = f_2(x_{12}(z)) = f_2 \circ x_{12}(z).$$

Additionally, we say that a category has (all) binary products, when for any pair of objects A, B a product $A \times B$ exists. Ternary and n-ary products can be defined as well. I can be shown that the binary product is associative up to isomporphism [Awo10], i.e.

$$(A \times B) \times C \cong A \times (B \times C).$$

If a category contains all finite n-ary products, we say that the category has all products.

A product of arrows can be defined between two objects that are already products. For arrows from object $A \times A'$ to $B \times B'$, the arrow is defined as

$$f \times f' = A \times A' \to B \times B',$$

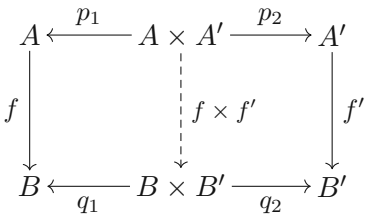


Figure 4.4: The commutative diagram for $f \times f'$. [Awo10]

where $f \times f' = \langle f \circ p_1, f' \circ p_2 \rangle$. The commutative diagram is shown in Figure 4.4.

Another construction of objects, that is important for the definition of a Cartesian Closed Category (CCC) is the exponential. First, we will define a similar structure on sets.

Intuitively, the exponential states the correspondence between a function $f : (A \times B) \to C$ and a function $\tilde{f}$ from A to a set of functions from B to C and the evalation of the returned function $g : B \to C$ with the second parameter $b \in B$. This looks very similar to currying, we will show later how these concepts go together.

More formally, given a function $f(x, y) : A \times B \to C$, where $x \in A$ and $y \in B$, we can fix an element $a \in A$, and get a function $f_a(y) : B \to C$, where $f_a(y) = f(a, y)$. We define C^B as the set of all functions from B to C, and in particular $f_a(y) \in C^B$. The function $\tilde{f} : A \to C^B$ takes a parameter $a \in A$ and returns a function $f_a(y)$ where that parameter is fixed, i.e.

$$\tilde{f}(a)(b) = f_a(b) = f(a, b).$$

Clearly, the functions $f : A \times B \to C$ and $\tilde{f} : A \to C^B$ correspond to each other bijectively. To complete this idea, the evaluation function $eval : C^B \times B \to C$ is defined, that maps an element (g, b) to $g(b)$, where g is a function $g : B \to C$ and $b \in B$, i.e. $eval(g, b) = g(b)$.

Exponentials in categories are now defined in a similar manner:

Definition 4.1.7 ([Awo10])**.** Assuming we have a category $\mathbf{C}$, having binary products. An *exponential* of objects B and C of $\mathbf{C}$ consists of an object C^B and an arrow

$$\epsilon : C^B \times B \to C$$

such that for any object A and arrow $f : A \times B \to C$ there is a unique arrow

$$\tilde{f} : A \to C^B,$$

such that

$$\epsilon \circ (\tilde{f} \times 1_B) = f.$$

The exponential object can also be denoted by $B \Rightarrow C$.

The definition is visualized in Figure 4.5, and we have the following terminology:

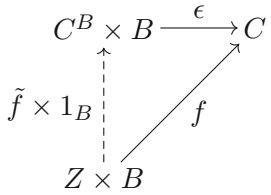


Figure 4.5: The diagram of the exponential and the necessary conditions. [Awo10]

- $\epsilon : C^B \times B \to C$ is called *evaluation*,
- $\tilde{f} : Z \to C^B$ is called the (exponential) *transpose* of f.

4.1.2 Cartesian Closed Category

Finally, we can give the formal definition of Cartesian Closed Categories:

Proposition 4.1.1 ([Awo10])**.** A category **C** is a CCC iff it has the following structure:

- A distinguished terminal object 1, i.e. there is a unique object 1 and for each object C there is a unique arrow $f : C \to 1$. If there are arrows $f : C \to 1$ and $g : C \to 1$ of the same object C, then $f = g$ must hold.
- For each pair of objects A, B, there is a product $A \times B$.
- For each pair of objects A, B, there is an exponential object B^A with the corresponding arrow $\epsilon : B^A \times A \to B$.

An example for CCCs is our example of sets with functions as morphisms. The unique terminal object is the empty set, the product is defined as the cartesian product between two sets. Also the exponential can be easily defined on any pair of sets A and B. The evaluation arrow ε is given by applying an element of A to the function $A \to B$.

4.2 Cartesian Closed Categories and the λ-calculus

A link between simply typed λ-calculus and CCC has been shown by [Lam86]. Indeed, simply typed λ-calculus and CCC are essentially equivalent. We will follow the explanations given in [Hu18].

First we will define the syntax of typed λ-calculus.

Definition 4.2.1 ($\lambda^{\to\times}$-signature [Hu18])**.** A $\lambda^{\to\times}$-signature σ is given by the following data:

- We fix a set TV of ground types. The collection of *simple types* over TV is called $\mathsf{ST(TV)}$, and is defined by a BNF:

$$\mathsf{ST(TV)}\text{: } A ::= G \mid \mathsf{unit} \mid A \to A \mid A \times A \qquad \text{where } G \in \mathsf{TV}.$$

- A collection of *function symbols*, each with a given arity. If f is an n-ary function symbol, such that $n > 0$, it is associated to a mapping $f : A_1 \times ... \times A_n \to B$. A 0-ary function symbol c is a constant and associated to a type $c :: B$.

Definition 4.2.2 (λ-terms [Hu18])**.** The collection of *raw terms* Λ generated by σ, assuming a countably infinite set V of variables, is defined as follows:

$$\Lambda : \qquad t ::= x \mid f(t, ..., t) \mid \lambda x : A.t \mid tt \mid \langle\rangle \mid \langle t, t\rangle \mid \mathsf{fst}(t) \mid \mathsf{snd}(t),$$

where $x \in V$, A is in $\mathsf{ST(TV)}$, and f is an n-ary function symbol. The last two terms are used to evaluate the pairs, i.e. $\mathsf{fst}(\langle u, v\rangle) = u$ and $\mathsf{snd}(\langle u, v\rangle) = v$. The set of raw terms are regarded as syntactically valid expression, where the typing could be incorrect. Function types $A \to A$ are the core concept of typed lambda calculus. Product types $A \times A$ can be represented as function types due to currying, but they are included, because of the dual concept of products in CCCs.

The following model of $\lambda^{\to\times}$ in category theory was introduced by [Cro94].

Definition 4.2.3 (Model of $\lambda^{\to\times}$ [Hu18])**.** Given a $\lambda^{\to\times}$-signature σ, we define a model $\mathbb{M}$ in a Cartesian Closed Category $\mathcal{C}$ to be a structure generated by the following data:

- for every ground type $G \in \mathsf{TV}$ of σ, an associated object $[\![G]\!]$;
- for every constant $c : A$, an associated morphism $[\![c]\!] : 1 \to [\![A]\!]$;
- for every n-ary function symbol, where $n > 0$, $f : A_1 \times ... \times A_n \to B$, an associated morphism $[\![f]\!] : [\![A_1]\!] \times ... \times [\![A_n]\!] \to [\![B]\!]$;

for which the following rules are given:

- the unit type unit is associated to the terminal object 1;
- for every other simple type $A \in \mathsf{ST(TV)}$, associations are given inductively by

$$[\![A \to B]\!] := [\![A]\!] \Rightarrow [\![B]\!] = [\![B]\!]^{[\![A]\!]},$$
$$[\![A \times B]\!] := [\![A]\!] \times [\![B]\!].$$

- *Typing contexts* are also modelled by objects:

$$[\![\Gamma]\!] = \begin{cases} 1, & \text{if } \Gamma = [\,] \\ [\![A_1]\!] \times ... \times [\![A_n]\!], & \text{if } \Gamma = [x_1 : A_1, ..., x_n : A_n]. \end{cases}$$

We have previously introduced the typing context as basis or environment for free variables. In this setting, the typing context is defined as a finite list of pairs of variable and type

$$\Gamma := [x_1 : A_1, ..., x_n : A_n],$$

where $x_1, ..., x_n \in V$ and $A_1, ..., A_n \in \mathsf{ST}(\mathsf{TV})$. $\Gamma \vdash a : A$ indicates that each free variable $x_1, ..., x_n$ occurring in a term a is assigned a type $x_1 : A_1, ..., x_n : A_n$.

Finally, λ-terms are interpreted by typing rules. We will show how a variable (x) and application (tt) are modelled and leave the other rules to be looked up in [Hu18]:

$$\frac{}{[\![\Gamma + [x : A] + \Gamma' \vdash x : A]\!] := \pi_i : [\![\Gamma]\!] \times [\![A]\!] \times [\![\Gamma']\!] \to [\![A]\!]}$$

If a variable occurs free, then the value should be given by the basis $\Gamma_{All} = \Gamma + [x : A] + \Gamma'$. $x : A$ is the i-th element of Γ_{All} and $+$ denotes the disjoint union. The term is modelled by an arrow from the product of $[\![\Gamma]\!] \times [\![A]\!] \times [\![\Gamma']\!] \to [\![A]\!]$.

$$\frac{[\![\Gamma \vdash u : A \to B]\!] = [\![u]\!] : [\![\Gamma]\!] \to [\![B]\!]^{[\![A]\!]} \qquad [\![\Gamma \vdash v : A]\!] = [\![v]\!] : [\![\Gamma]\!] \to [\![A]\!]}{[\![\Gamma \vdash uv : B]\!] := [\![\Gamma]\!] \xrightarrow{\langle [\![u]\!], [\![v]\!] \rangle} [\![B]\!]^{[\![A]\!]} \times [\![A]\!] \xrightarrow{\epsilon} [\![B]\!]}$$

This rules shows an application of a term v of type A to a term u of type $A \to B$, that results in a term uv of type B. Semantically, if under the context $[\![\Gamma]\!]$, $[\![u]\!]$ is of type $[\![B]\!]^{[\![A]\!]}$ and $[\![v]\!]$ is of type $[\![A]\!]$, then the application of v to u, i.e. $[\![uv]\!]$, is of type $[\![B]\!]$. We can think of $[\![B]\!]^{[\![A]\!]}$ as a function from A to B.

We have an arrow $[\![u]\!] : [\![\Gamma]\!] \to [\![B]\!]^{[\![A]\!]}$ for the term u and an arrow $[\![v]\!] : [\![\Gamma]\!] \to [\![A]\!]$. As the CCC contains all finite products, there must be also an arrow $\langle [\![u]\!], [\![v]\!] \rangle : [\![\Gamma]\!] \to [\![B]\!]^{[\![A]\!]} \times [\![A]\!]$. Furthermore, a CCC contains exponentials, and therefore it must also contain the evaluation arrow ϵ from the object $[\![B]\!]^{[\![A]\!]} \times [\![A]\!]$ to the object $[\![B]\!]$. Indeed, we reach an object $[\![B]\!]$ from existsing objects $[\![B]\!]^{[\![A]\!]}$ and $[\![A]\!]$, as we can also create a term $uv : B$ from terms $u : A \to B$ and $v : A$.

4.3 Relation to PCF

The approach of categorical semantics for PCF has been described in detail by [HO00], we will give some basic ideas of that publication. Categorical semantics allow to model certain aspects of of PCF that could not quite be described in terms of the standard denotational semantics.

A programming language can be described in terms of a type theory T. The syntax can be formalized in a category $\mathbb{T}$ and the model is given by functors $\mathcal{M} : \mathbb{T} \to \mathbb{C}$ from a *classifying category* $\mathbb{T}$ to $\mathbb{C}$, where $\mathbb{C}$ is the resulting model.

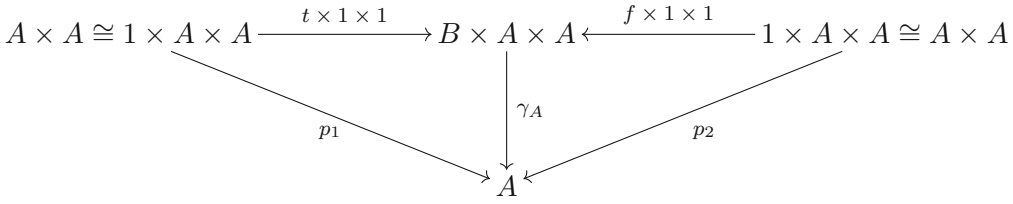


Figure 4.6: The commutative diagram of the conditional.

In a very general notion, *c-fix categories* are introduced, that provide the properties of a model of PCF.

Before we give the definition, we need explain some preliminary conditions. For any object X of a category $\mathbf{C}$, there is an isomorphism from X to the object $X \times 1$. By the definition of a category, we have an identity arrow $1_X : X \to X$ and by the defintion of a CCC, we also have a unique arrow f to the terminal object 1. Consequently, there also must be a unique arrow $\langle 1_X, f \rangle : X \to X \times 1$. The arrow in the other direction is given by $p_1 : X \times 1 \to X$. It can be easily checked that this operation is commutative, and as a result we write

$$X \cong X \times 1 \cong 1 \times X.$$

We have already defined the product between two arrows. For an arrow $f : A \to B$, and a category C, we can easily construct an arrow $f \times 1_C : A \times C \to B \times C$ by inserting 1_C for the general arrow f' in the commutative diagram in Figure 4.4.

For any object C, there is a product $C \times C$ and by definition, we have an arrow from C to $C \times C$, which we will denote by Δ.

Definition 4.3.1 ([HO00])**.** A *c-fix category* is a Cartesian Closed Category $\mathbb{C}$ equipped with the additional structure:

- *The conditional.* An object B, two maps $t : 1 \to B, f : 1 \to B$, and a family of maps
$$\gamma_A : B \times A \times A \to A$$
for each object A of $\mathbb{C}$ with the property shown in the commutative diagram in Figure 4.6.
- *Fixed points.* A family of maps for each object A of $\mathbb{C}$
$$\mathbf{Y}_A : A^A \to A$$
with the poperty shown in Figure 4.7.

Additionally, numerals are needed to model ground type of `Num`. This is done by an object N of the category $\mathbb{C}$ with the following maps

$$1 \xrightarrow{0} N \qquad\qquad N \xrightarrow{s} N,$$

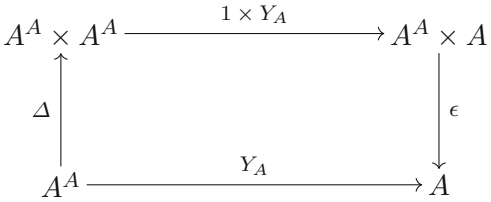


Figure 4.7: The diagram of the fixed-point operator at type A. It shows the equality of $f(Y_A(f)) = Y_A(f)$.

where the map 0 stands for the numeral 0, $s \circ 0$ stands for the successor of 0, i.e. $s(0)$), $s \circ s \circ 0$ stands for $s(s(0))$, etc. A very detailed explanation is given in [HO00].

Finally, a categorical model of PCF can be characterized in a concrete and in an abstract way:

Definition 4.3.2 (Concrete version [HO00])**.** A categorical model for PCF is a c-fix category equipped with an object of numerals.

Definition 4.3.3 (Abstract version [HO00])**.** A categorical model for PCF is a functor $\mathcal{M} : \mathbb{T} \to \mathbb{C}$ of c-fix categories.

CHAPTER 5

Sequential Semantics

As we have seen in the previous chapter PCF is an example of a sequential programming language, i.e. it only computes sequential functions. A part of the difficulty of this chapter is the formal definition of sequentiality. Although the idea of sequentiality is intuitively clear, it is hard to formulate in precise, mathematical terms. The definition of sequentiality has similar difficulties as the intuitive definition of *effectively computable* functions, and the Church-Turing thesis [GW05]:

> Whenever there is an effective method (algorithm) for obtaining the values of a mathematical function, the function can be computed by a Turing Machine.

We will see several models, that define sequentiality in a slightly different way. This section mainly gives an overview of major attempts of solving the full abstraction problem of PCF through sequential functions shown in [Ong95].

The sticking point of full abstraction of PCF, on which various attempts failed, is the tradeoff between sequentiality and extensionality. On one hand, when functions are interpreted more extensionally, i.e. by their input and output, they make good computational sense and have good closure properties for higher types (e.g. Scott-continuous functions of $\mathcal{S}$). However, they often include some functions which can only be computed by parallel algorithms. On the other hand, mathematical structures that are intuitively sequential rely on intensional interpretations, i.e. the intensional behavior of a function. They often do not fullfill closure properties and the conditions for a cartesian closed category of functions.

5.1 Sequential Functions

First-order sequential functions take tuples of ground values, e.g. natural numbers, as arguments and return ground values as a result. The description of such functions is

quite natural, the intuition is that the argument of the function has to be visited in a specific order. If a function is strict, i.e. $\perp$ as an argument must cause the function to return $\perp$, and it is not the constant $\perp$-function, then there must be an i-th argument that is needed first for the computation of the result. If the argument is undefined, then undefined is returned, otherwise the next argument, say j, is used in the computation process. A continuous function from cpos $D_1 \times ... \times D_{n+1}$ to E is defined as sequential by [Mil77] if it is either a constant, or there is an integer i with $1 \leq i \leq n+1$ such that f is strict in the i-th argument, and the function obtained by fixing the i-th argument is sequential. The product $D_1 \times D_2 \times ... \times D_{n+1} \to E$ is simply defined by the curried version $(D_1 \to (D_2 \to ...(D_{n+1} \to E)...)$.

These definitions of sequentiality have some drawbacks. First, they apply only to functions with two or more arguments, functions with one argument are reduced to strictness. Secondly, sequentiality is only captured for flat cpos and it breaks down for higher types. However, sequential functions give a good indication of requirements for description of sequentiality:

- *Local partiality:* This condition describes local strictness, i.e. what happens if one argument is undefined.
- *State:* Sequentiality as a computational process describes an intensional property. To be able to concretely analyze sequentiality, we need a notion of a state of computation.
- *Place:* A notion of place indicates different sites of sub-computations, where many things are done, but one thing at a time. In Milner's definition of first-order sequentiality, the component of a cartesian product is used to express the notion of place as a first approximation.

Concrete Data Structures (CDS) [KP93] are just what the name indicates: a concrete (or intensional) mathematical structure well-suited for denoting the structure of computation. This concept brings together local partiality, state, and place to form an abstract notion of sequentiality that extends to higher types.

The definition may look rather complicated at first, but it is helpful to consider an example of a CDS. Formal (partial) terms can be regarded as labelled trees over a first-order signature Σ, representing partial functions from the set $\mathbb{N}*$ of finite sequences of natural numbers to Σ. For example, the term $f(a, g(b, \Omega), \Omega)$ is shown in Figure 5.1.

Our example term $f(a, g(b, \Omega), \Omega)$ is also represented by the set $\{(\epsilon, f), (0, a), (1, g), (1 \cdot 0, b)\}$. In general, given a first-order signature Σ, a partial Σ-term t is a set of pairs of the form (u, f) where u is just a finite sequence of natural numbers, and $f \in \Sigma$ must satisfy the following conditions:

- *Consistency:* whenever $(u, f), (u, g) \in t$, then $f = g$;

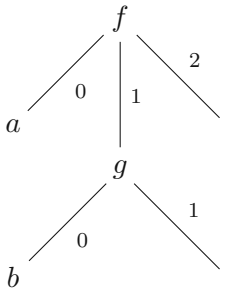


Figure 5.1: The term $f(a, g(b, \Omega), \Omega)$ represented as labelled tree [Ong95].

- *Safety:* if $(u, f) \in t$ and $u = v \cdot i$, then $(v, g) \in t$ for some g with arity of at least i

The first component u in the pair (u, f) is read as the *occurrence* of the symbol f in the term t. The membership predicate $(u, f) \in t$ denotes that the symbol f occurs in the term t at occurrence u. The consistency condition ensures that the set of pairs defines a graph of a functional relation. The safety condition requires that the tree construction respects the arity of the function symbols.

The introduced concepts now allow us to formally define a CDS:

Definition 5.1.1 (Concrete Data Structure (CDS) [Ong95])**.** A CDS M is a structure $\langle C, V, E, \vdash \rangle$ where

- C is the set of *cells*,
- V is the set of *values*,
- the collection E of events is a subset of $C \times V$ (an event is writtten as (c, v)),
- $\vdash$ is an *enabling relation* between finite collections of events on the one hand and cells on the other. For any finite subset X of E (denoted by $X \subseteq_{fin} E$), we read $X \vdash c$ as "X enables the cell c".

A CDS is more concrete than the previous setting of cpos, because it allows a more local and lower-level presentation of computations. Computation proceeds in a discrete manner, at each step some cells may be filled and some cells may not be filled. The collection of cells that have been filled at a point in time give a snapshot of computation at that point. As a natural consequence, we would like to describe a state of computation. Formally, a state x of M is a subset of E satisfying the previously-defined conditions of consistency and safety. Applied to the definition of CDSs, we get the following notation [Ong95]:

- *Consistency:* whenever $(c, v) \in x$ and $(c, v') \in x$, then $v = v'$

- *Safety:* whenever $(c, v) \in x$, then there is a sequence of events $e_0, ..., e_n$ such that $e_n = (c, v), e_i = (c_i, v_i) \in x$, and $\{e_j : j < i\}$ contains an enabling of c_i for all $i \leq n$.

When an event takes place, i.e. a cell is filled, e.g. at state x, then some cells might become enabled, so that they may be filled in the next step. This effect is captured by the enabling relation $\vdash$.

Furthermore, we define *deterministic concrete data structures* (DCDS), for which we need some further notations. For a state x, we define [Ong95]:

- $c \in \mathbf{F}(x) \quad \exists v \in V.(c, v) \in x$ "c is filled in x"
- $c \in \mathbf{E}(x) \quad \exists X \subseteq_{fin} x.X \vdash c$ "c is enabled in x"
- $c \in \mathbf{A}(x) \quad c \in \mathbf{E}(x) - \mathbf{F}(x)$ "c is accessible from x"

A cell c which has been enabled, but not yet filled in a state x is called *accessible* from x. Generally, there might be more than one cell that is accessible from a state x. The notion of accessibility of a cell is an important idea, that allows us to define a new notion of sequential functions between CDSs. We consider CDSs satisfying the following conditions:

- *Stability:* for any state x, each cell that is enabled in x has a unique enabling,
- *Well-foundedness:* the binary relation $c \ll d$ over cells is defined by

$$c \ll d \iff \text{there is some } x \text{ such that } c \in \mathbf{F}(x) \text{ and } x \vdash d.$$

A well-founded and stable CDS is called a *deterministic concrete data structure*. Furthermore, a DCDS is said to be in *filiform* if every enabling is affected by at most one event. Looking back at the definition of c being enabled in x, the set X must contain only one element, i.e. $\exists z.(z \in x \text{ and } \{z\} \vdash c)$. A filiform DCDS can be seen as a tree, where the nodes are events and the edges are enablings.

Finally, we will formally define sequential functions and present some results that were already hinted at the beginning of this subsection.

Definition 5.1.2 (Kahn-Plotkin sequential functions [Ong95])**.** Let f be a function from the set of states of the CDS $M = \langle C, V, E, \vdash \rangle$ to the set of states of the CDS $M' = \langle C', V', E', \vdash' \rangle$. For any state x of M, and any cell c' accessible from $f(x)$, the function f is said to be *sequential* for x at c' if exactly one of the following conditions is valid:

- for any state $z \supseteq x, c'$ remains accessible from $f(z)$,

- there is a cell c accessible from x such that for any $y \supseteq x$, if c' is filled in $f(y)$ then c must also be filled in y.

A cell c, as defined above, is called a *sequentiality index* of f for the state x at the cell c'. A function f is said to be *sequential* if it is sequential for x at any cell c accessible from $f(x)$, for any state x.

The structure of CDS and Kahn-Plotkin sequential functions helped a lot to advance the understanding of higher-type sequentiality, and would be a great model for higher-type sequential computation. Unfortunately, it is only possible to pass ground types as parameters to a function and not compound types σ -> τ. But passing functions as parameters is an important feature of typed λ-calculus and PCF. The exponential in cartesian closed categories corresponds to the closure of function construction, i.e. if σ and τ are types, then $\sigma \to \tau$ is a type as well. The following statement is a sensible consequence.

Theorem 5.1.1 ([Ong95])**.** The category of DCDS and sequential functions is not cartesian closed.

The proof by [Cur12] relies on the following result: The category of DCDS and sequential functions is equivalent to the category of distributive concrete domains and sequential functions. The proof by contradiction shows that an assumed exponentiation from a domain D to E is not a concrete domain.

We will present two further attemps that were made in finding a cartesian closed category of higher-type sequential functions. One effort was made by [Ber78], which relaxed the requirements of sequentiality, stayed within the framework of functions, and introduced the notion of *stability*. The second approach uses ideas of CDS and sequential functions, giving up on extensionality, which was introduced by [BCL85]. The idea was to refine sequential functions to *sequential algorithms* over CDSs.

5.2 dI-domains and stable functions

Stable functions, introduced by [Ber78, Ber79], are located between continuous functions by Scott [Sco70] and Kahn-Plotkin sequential functions [KP93] in terms of degree of sequentiality. Stability can be seen as an approximation of higher-type sequentiality. Intuitively, a function is stable, if a definite amount of information is needed from the argument to obtain an approximation of the result. In domain-theoretic terms, if D and E are cpos, then a continuous function $f : D \to E$ is *stable* if for every $e \in E$ and $d \in D$ such that $e \sqsubseteq f(d)$, the set

$$\{x \in D : x \sqsubseteq d, e \sqsubseteq f(x)\}$$

has a least element. We denote it as $m(e, f, d)$. An example of a continuous function that is not stable is `por`. Let e be `True` and d be $\langle$`True, True`$\rangle$. It can be seen that the

set $\{x \sqsubseteq d : e \sqsubseteq \texttt{por}\ x\}$ has two minimal elements: $\langle \bot, \texttt{True}\rangle$ and $\langle \texttt{True}, \bot\rangle$; but it has no least element.

With respect to the standard extensional order of cpos, stable functions do not form a cartesian closed category. A proof for this is given in [Ong95], which shows that exponentiation fails for certain function applications. To resolve this issue, the ordering between stable functions is modified by introducing *dI-domains*. A dI-domain D is a Scott domain, i.e. a consistently complete algebraic cpo) satisfying the axioms (d) and (I) [Ong95]:

Axiom (d) For any $d, e, f \in D$, whenever $e \uparrow f$, then $d \sqcap (e \sqcup f) = (d \sqcap e) \sqcup (d \sqcap f)$. The notation $e \uparrow f$ was defined in 3.2.4.

Axiom (I) Every compact element dominates finitely many elements, i.e. for any compact element d of D, the set $\{x \in D : x \sqsubseteq d\}$ is finite.

Using dI-domains together with stable functions leads to the following result:

Theorem 5.2.1 ([Ber78])**.** The category of dI-domains and stable functions is cartesian closed.

Finally, the most important question is, if there is a fully abstract model of PCF. Considering the function `gustave`[1] :: `(Bool -> Bool -> Bool -> Bool)` which is defined as the function evaluating to `True` given the following parameters and being `False` otherwise:

- `gustave`$\langle\Omega$, `True`, `False`$\rangle$ = `True`
- `gustave`$\langle$`False`, Ω, `True`$\rangle$ = `True`
- `gustave`$\langle$`True`, `False`, $\Omega\rangle$ = `True`

The three tuples are pairwise incompatible. From each one, the other can be obtained by cyclically permuting the components. Using the above function, let us assume e to be `True`, and d to be $\langle\Omega, \texttt{True}, \texttt{False}\rangle$. The set $\{x \sqsubseteq d : e \sqsubseteq \texttt{gustave}\ x\}$ has one element: $\langle\Omega, \texttt{True}, \texttt{False}\rangle$. The function `gustave` is stable but not Kahn-Plotkin sequential, i.e. it can be represented in the model, but it cannot be represented syntactically in PCF, because PCF is a sequential language. We came across a similar issue with the model $\mathcal{D}$ in Section 3.3. Consequently, the stable function space model is not fully abstract for PCF.

There have been attempts to extend PCF such that the stable function space model is fully abstract but they were not quite satisfying [JM96]. Stability has also been further studied by [Gir86].

[1] Gustave is Gérard Berry's nickname. [Pao06]

5.3 Sequential algorithms

While Kahn-Plotkin sequential functions are in principle applicable to higher types, the category of deterministic concrete data structures with sequential functions is not cartesian closed. A cartesian closed category of DCDSs was proposed by [BC82]. Given two DCDSs M and M', they define a new DCDS $M \Rightarrow M'$ whose states are sequential algorithms from M to M'. A sequential algorithm is characterized by a function that associates input-output behaviour with a computation strategy. This enables a definition of categorical composition of sequential algorithms.

Let $M = \langle C, V, E, \vdash \rangle$ and $M' = \langle C', V', E', \vdash' \rangle$ be two DCDSs. The structure $M \Rightarrow M' = \langle C'', V'', E'', \vdash'' \rangle$ is defined as follows:

- C'' is the set of pairs, written xc', where x is a finite state of M, and c' is a cell of M';
- V'' is the disjoint union of C and V'; the two kinds of elments are represented as $\mathsf{valof}\ c$ and $\mathsf{output}\ v'$ respectively;
- E'' is defined by the following bi-implications:

$$\begin{array}{lcll} (xc', \mathsf{valof}\ c) \in E'' & \iff & c \in A(x) & \text{“valof events”}, \\ (xc', \mathsf{output}\ v') \in E'' & \iff & (c', v') \in E' & \text{“output events”}; \end{array}$$

- the enabling $\vdash''$ is of two types defined as follows
  - type valof: $(xc', \mathsf{valof}\ c) \vdash'' yc'$ if and only if $x \prec_c y$;
  - type output: $(x_1c'_1, \mathsf{output}\ v'_1), ..., (x_nc'_n, \mathsf{output}\ v'_n) \vdash'' xc'$ if and only if

$$x = \bigcup_{1 \le i \le n} x_i \text{ and } (c'_1, v'_1), ..., (c'_n, v'_n) \vdash' c'.$$

For the notation $x \prec_c y$, we need to recall the *filiform DCDS*. The states of this DCDS can be seen as a tree, where the nodes are events and the edges are enablings. $x \prec_c y$ means that $y = x \cup \{(c, v)\}$ for some value v, i.e. the state y equals to state x extended with one cell and value pair. We say y *covers* x.

A state of $M \Rightarrow M'$ is called *sequential algorithm* from M to M'. Each sequential algorithm is associated with an input-output function. For any state x of M and any state of a of $M \Rightarrow M'$,

$$a \cdot x \stackrel{\text{def}}{=} \{(c', v') : \exists y \subseteq x.(yc', \mathsf{output}\ v') \in a\}.$$

This definition of $a \cdot x$ is the result of applying a to x. Formally, events are a collection of pairs which have two shapes: $(xc', \mathsf{valof}\ c)$ or $(xc', \mathsf{output}\ v')$, so either a valof event

or an output event. Considering the effect of applying a sequential algorithm a to the argument x, valof does not play a role in the definition of $a \cdot x$. This is because the definition is intended to capture the extensional behaviour of the algorithm a, as opposed to the intensional details of the output events which get suppressed.

As an example, we show the formal descriptions of the two addition algorithms "left-add" and "right-add" (ω denotes the set of natural numbers):

$$\begin{aligned}
\mathsf{l\text{-}add} &= \{(\{\}\Gamma, \mathsf{valof}\ \Gamma \cdot 1)\} \\
&\quad \cup \{(\{(\Gamma \cdot 1, i)\}\Gamma, \mathsf{valof}\ \Gamma \cdot 2) : i \in \omega\} \\
&\quad \cup \{(\{(\Gamma \cdot 1, i), (\Gamma \cdot 2, j)\}\Gamma, \mathsf{output}\ i + j) : i, j \in \omega\}; \\
\mathsf{r\text{-}add} &= \{(\{\}\Gamma, \mathsf{valof}\ \Gamma \cdot 2)\} \\
&\quad \cup \{(\{(\Gamma \cdot 2, j)\}\Gamma, \mathsf{valof}\ \Gamma \cdot 1) : j \in \omega\} \\
&\quad \cup \{(\{(\Gamma \cdot 2, j), (\Gamma \cdot 1, i)\}\Gamma, \mathsf{output}\ i + j) : i, j \in \omega\}.
\end{aligned}$$

We will explain the "left-add" algorithm, the "right-add" algorithm is analogous to that. At the start of the computation at the event $(\{\}\Gamma, \mathsf{valof}\ \Gamma \cdot 1)$, the algorithm asks for the value of the left argument which corresponds to the cell $\Gamma \cdot 1$. Suppose the left argument supplies a value, say i, then the event $(\{(\Gamma \cdot 1, i)\}\Gamma, \mathsf{valof}\ \Gamma \cdot 2)$ indicates that the algorithm asks for the value of the right argument $\Gamma \cdot 2$. We assume that the right argument returns a value j and the output event $(\{(\Gamma \cdot 1, i), (\Gamma \cdot 2, j)\}\Gamma, \mathsf{output}\ i + j)$ concludes the computation by returning the value $i + j$.

The following result was proven by Berry and Curien:

Theorem 5.3.1 ([BC82])**.** The category of DCDSs (deterministic concrete data structures) and sequential algorithms is cartesian closed. It has a full subcategory, the cartesian closed category of filiform DCDSs.

An issue of DCDSs and sequential algorithms is that they do not yield an extensional model of PCF. For example. "left-add" and "right-add" are two distinct sequential algorithms, although they have the same extensional behaviour, which is standard addition. It was shown by [Cur12], that there is a way to construct a continuous and order-extensional model of PCF, by using the idea of bi-structures [Ber78]. However, the model associated with these bi-structures was not fully abstract for PCF. Again, there are compact elements in the model that cannot be defined in PCF. There were three counterexamples given in [Cur86], we will show the first of those examples:

The function $\mathtt{A}^1$ of type `(Bool -> (Bool -> Bool)) -> Bool` is characterized as

$$\begin{aligned}
&(\mathtt{A}^1)\ \mathtt{a}_1^1 = \mathtt{True} \\
&(\mathtt{A}^1)\ \mathtt{a}_2^1 = \mathtt{False}
\end{aligned}$$

where

Model	**CCC**	**Fully Abstract for PCF**
Sequential Functions and DCDS	No	No (not even CCC)
Stable Functions and dI-domains	Yes	No (counterexample: `gustave` function)
Sequential Algorithms and DCDS	Yes	No (elaborate counterexamples given in [Cur86])

Table 5.1: Summary of presented sequential models.

(a_1^1) `True` $\perp$ = `True`
(a_1^1) `False True` = `True`
(a_1^1) `False False` = `False`

(a_2^1) $\perp$ `True` = `True`
(a_2^1) `False False` = `False`
(a_2^1) `True False` = `True`.

The representation of this function in the model with bi-structures as well as the induction proof of non-definability in PCF is given in [Cur86]. We will omit it here.

However, the model of sequential algorithms and DCDS is fully abstract for the language **CDS**. This was investigated in detail in [BCL85]. Other developments in sequential semantics were made by [CF92, CCF94].

We have seen three major attempts to define a model of PCF through sequential semantics. None of them were successful in providing a fully abstract model of PCF. They are summarized in Table 5.1. Only later, a new model was introduced that used the ideas of sequentiality, but also assigned each step of execution to a player or opponent. This crucial change led to a full abstraction.

CHAPTER 6

Game Semantics

Game semantics were first introduced by [Hyl97] and [A+97]. As mentioned in the introduction, in game semantics computation is represented as a game between two participants, the Player (or Proponent) P and the Opponent O. In general, the Player P represents the system and O represents the environment. In this application of game semantics, the system is a term and the environment is the context in which the term is used. This is the main difference to previous sequential semantics we have shown. In the games considered, O makes the first move and then the two players move in alternating turns. In each step, there is a set of available moves that are allowed depending on the rules of the game. A strategy is a set of predetermined responses of P for the moves of O. Before we present formal details of games, we will give an informal description following the introduction by [AM99].

6.1 Informal Introduction

When modelling a programming language, the first thing is to model the ground types. In standard denotanional semantics, a ground type is represented as a flat domain with a least element $\perp$. In game semantics, each value of a ground type is modelled by a simple interaction. Looking at numbers for example, the environment starts the initial move q as a question "What is the number?", and P responds by playing a natural number. The game N of natural numbers is depicted in Figure 6.1. The strategy for a concrete number, e.g. 3, looks like this: "When O plays q, I will play 3."

This can be visualized in the following diagram, where the moves are shown from top to bottom. O starts by playing q and after that P responds with 3.

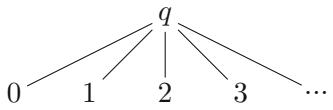


Figure 6.1: The game N of natural numbers [AM99].

N	
q	O
3	P

For functions, the principle is the same, i.e. O starts to ask for value of the function, but now O also needs to provide an input for the function. The player P consumes the input and produces an output. The game $N \Rightarrow N$ consists of copies of N, one for input and one for output. In the input copy, the situation is reversed: P demands a value with the move q, in this case the O/P role of moves is reversed. For a simple predecessor function the strategy is formulated as follows: "When O asks for output, I will ask for input; when O provides input n, I will give output $n-1$.". This strategy looks as follows:

N	$\Rightarrow$	N	
		q	O
q			P
3			O
		2	P

Both plays of N, in each column of N, are indeed valid. For example, O cannot start in the third move by directly providing a number instead of first asking for a value (in move 1). This idea of strategies and games allows to model all first-order functions, i.e. functions that require ground type arguments and return ground types. The strategy for addition is shown below.

N	$\Rightarrow$	N	$\Rightarrow$	N	
				q	O
q					P
3					O
		q			P
		2			O
				5	P

There is another way to look at this computation [MT16], which resembles more an actual dialogue:

O What is the result?
P What is the first parameter of the function?
O It is 3.
P What is the second parameter of the function?
O It is 2.
P The result is 5.

In the same way, we can form a game $A \Rightarrow B$ from games A and B. We take a copy of A and a copy of B, we place them in a strategy diagram and reverse the O/P roles of the moves in A.

Moving further from this idea, we get to higher-order functions. Let's have a look at the λ-function $\lambda f.f\ 0\ 1$:

$(N$	$\Rightarrow$	N	$\Rightarrow$	$N)$	$\Rightarrow$	N	
						q	O
				q			P
q							O
0							P
		q					O
		1					P
				n			O
						n	P

In this example O demands the value of the rightmost N, to which P replies with a question for value of the function f. Then O poses a question for the first parameter, which is given by P. The same happens with the second parameter. In the end, O responds with $n = f\ 0\ 1$ for the game $(N \Rightarrow N \Rightarrow N)$, which is then returned by P as the overall output. In the dialogue view it would look like this:

O What is the result?
P What is the function f?
O What is the first parameter of the function?
P It is 0.
O What is the second parameter of the function?
P It is 1.
O The output of f is n.
P The result is n.

However, the strategy for the same function could also look different. While it is not possible for P to change its moves, it has to answer 0 to the first input and 1 to the second, O could behave differently. For example, it could ask for the arguments to f in the other order or could neglect to ask for arguments at all. In general, higher-order functions could use their arguments more than once. This is displayed in the example of $\lambda f.\ (f\ 0) + (f\ 1)$ below, where f is used twice.

$(N \Rightarrow N) \Rightarrow N$		
	q	O
q		P
q		O
0		P
n		O
q		P
q		O
1		P
m		O
	$n+m$	P

It is also possible for runs to be interleaved, this is shown in the strategy of $\lambda f.\ f(f(3))$ the diagram below. In this example, P asks for the output of f. When f (played by O) asks for input, P asks again for output from f. Now, when O asks for input, P can provide 3. O returns some output n which represents the value $f(3)$. P copies n as the input for the first call of f. The output m from O then represents $f(f\ 3)$ and P copies that as the overall output.

$(N \Rightarrow N) \Rightarrow N$		
	q	O
q		P
q		O
q		P
q		O
3		P
n		O
n		P
m		O
	m	P

6.1.1 Ambiguity of higer-order functions

As seen before, a play of a game could use the same chunk several times, i.e. a game $A \Rightarrow B$ may consist of several plays of A. Representing these interleaved sequences in the usual way, there is a possibility to create ambiguous plays and higher-order functions cannot be modelled correctly. Consider the type $((N \Rightarrow N) \Rightarrow N) \Rightarrow N$ with the following λ-terms:

- $M_1 = \lambda f.\ f(\lambda x.\ f(\lambda y.\ y))$
- $M_2 = \lambda f.\ f(\lambda x.\ f(\lambda y.\ x))$

In the usual style of game semantics, as presented in this chapter, the strategy is displayed in the diagram below.

$((N$	$\Rightarrow$	$N)$	$\Rightarrow$	$N)$	$\Rightarrow$	N	
						q	O
				q			P
		q					O
				q			P
		q					O
q							P
			$\vdots$				O

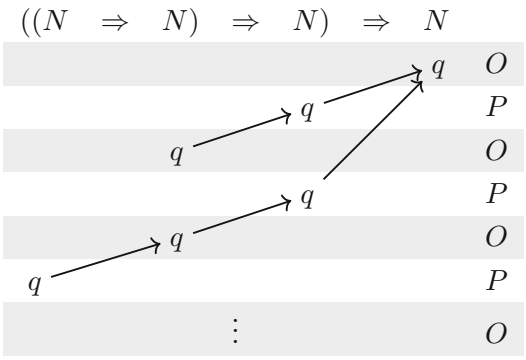
The O and P-moves are described as follows:

1. O asks for the output of the function. The result is calculated applying f to an argument, so P asks for the output of f.
2. O asks about the first input of f, which is some function $g = \lambda x.\ f(...)$ of type $N \Rightarrow N$. P could now ask about the value of x, but it knows that the output of the function g comes from f, so P asks for output from f.
3. O asks again about input for f, this time it is a function $h = \lambda y.\ ....$ To this, P replies with a question for input to a function of type $N \Rightarrow N$. But we do not know if P asks about the input for g or h. If it asks about g, we get the λ-term M_1, and if it asks about h, we get the λ-term M_2. This means that the strategies, as we have presented them so far, are ambiguous in some cases.

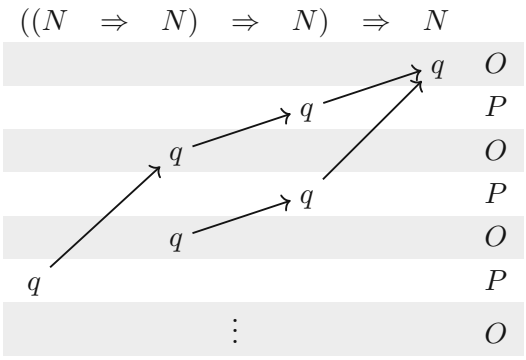
The solution proposed by [AM99] is to attach (justification) pointers to the diagrams, such that we know which sub-game belongs to which move. For M_1 and M_2, this is displayed in Tables 6.10a and 6.10b respectively.

Table 6.10: The strategies of M_1 and M_2.

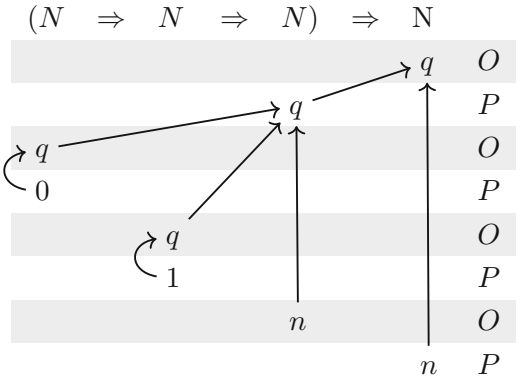
(a) M_1



(b) M_2



Every time there is a request for input of a function, there is a justification pointer to the move which asked for the output of that occurrence of the function. Also, there are pointers attached to answers, where the arrow points back to the question being answered. The example of $\lambda f.f\ 0\ 1$ is shown below.



6.1.2 Products

New types can be composed of subtypes by taking products. The type $A \times B$ consists of pairs of elements, one for A and one for B. The game $A \times B$ is a copy of A and a copy of B. When starting a play, O can decide whether to play in A or in B. From there, either a play of A or B is carried out. Consequently, a strategy for $A \times B$ determines a strategy for A and a strategy for B. The two plays of the strategy of the pair $\langle 3, 5 \rangle$ are:

N	$\times$	N	
q			O
3			P
		q	O
		5	P

In any given play, only one side of the product is investigated. This means that if a function of type $N \times N \Rightarrow N$ needs to use both components of the input, it must ask for the input twice. The strategy for addition in this style is shown below.

N	$\times$	N	$\Rightarrow$	N	
				q	O
q					P
3					O
		q			P
		2			O
				5	P

The strategy of $N \Rightarrow N \Rightarrow N$ was turned into $N \times N \Rightarrow N$, this is known as uncurrying. The opposite, turning a strategy $N \times N \Rightarrow N$ into the form $N \Rightarrow N \Rightarrow N$, is known as currying.

6.1.3 Composition of strategies

As we have stated previously, compositionality is a central part of interpretations of a programming language. In the way that several terms can be combined to a larger term, a strategy can be composed of several smaller strategies. In standard denotational semantics, this composition is done by function application, in game semantics the notion of compositionality is achieved by interaction of strategies.

As an example, we compose the strategy for addition with the type $N \times N \Rightarrow N$ with the strategy $\langle 3, 5\rangle : I \Rightarrow N \times N$. We use the game I to indicate, that our example might be part of a larger program, and there might be further plays to the left. Of course, it is also possible to compose strategies, where the left strategy does not have any more strategies to the left, e.g. by composing $N \times N$ with `add`, we would get a strategy for N. I is the empty game, which does not contain any moves. The plays of $I \Rightarrow A$ are the same as those of A, and the strategies for $I \Rightarrow A$ and A are also the same. An important note is that there is no way to play a move in the game $A \Rightarrow I$.

The messages from one stragegy are sent to the other in both ways. When a move is played in the subgame in $N \times N$ of `add`, we send it as an O-move to $\langle 3, 5\rangle$. Conversely, when a move is played in $N \times N$ of $I \Rightarrow N \times N$, we feed this move as an O-move back to `add`. The composed strategy is shown in Table 6.14a. By hiding of the middle game, we gain a new strategy, which is displayed in Table 6.14b. We will leave out the O-P column from now on.

In the composition of strategies, the strategy on the left hand side may be called repeatedly. The property of referential transparency assures in functional programming languages, that no matter how often a function is called, it behaves in the same way each time. In the game semantics space this property is covered by the condition of *innocence*, which is necessary for strategies modelling functional programming, and will be explained in detail soon.

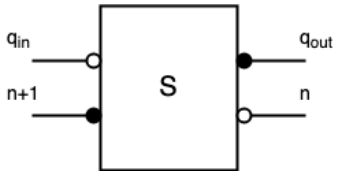


Figure 6.2: A module for the successor function [Car17].

6.1.4 Composition as modules

Another great illustration of composition was given by [Car17] in terms of modules. This notion of *composition as interaction of modules* was first investigated in linear logic [Gir87].

We give the example from [Car17], in which a module S is composed with a copy of itself. It has four channels labelled q_{in}, q_{out}, n, $n+1$. S is seen as the player P and the behaviour is defined as follows:

- S receives an input signal q_{in} from O, then
- emits a signal q_{out}, and
- waits for a value n from O and then, after receiving it,
- emits a value $n+1$.

The depicted behavior is exactly the same as the *handshake protocol*, that is designed to synchronize components and avoid interference of signals. The module is visualized in Figure 6.2, where the empty circle (○) means input (from O) and the filled circle (●) means output (from P).

We show the parallel composition of two modules S and S', $S||S'$, where S is the module defined above and S' is another instance of S. Note that the module S' is played by O,

Table 6.14: The composition of the strategies for $I \Rightarrow N \times N$ and $N \times N \Rightarrow N$.

(a) Without hiding

I	$\xRightarrow{\langle 3,5 \rangle}$	N	$\times$	N	$\xRightarrow{\texttt{add}}$	N
						q
		q				
		3				
				q		
				5		
						8

(b) Hiding the middle game

I	$\Rightarrow$	N
		q
		8

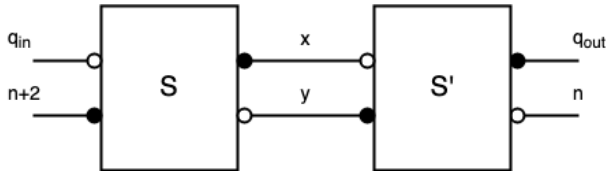


Figure 6.3: The composition of S and S', $S||S'$ [Car17].

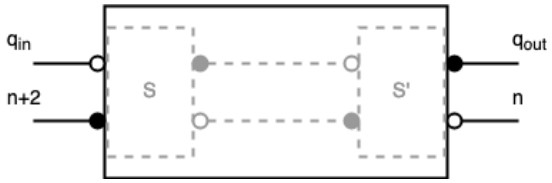


Figure 6.4: The final module calculating the successor of the successor [Car17].

as opposed to the usual play by P. As we have mentioned previously, in a game $A \Rightarrow B$ the roles of O and P are reversed in A. By connecting the the channels q_{out} and n of S with q_{in} and $n+1$ of S' we create a new module. The composition is shown in Figure 6.3, where the channels have been renamed appropriately:

- $n+1$ of S is renamed to $n+2$,
- q_{out} of S and q_{in} of S' are renamed to x,
- n of S and $n+1$ of S' are renamed to y;

By hiding the symbols x, y, we can build a single module, that is build from multiple modules, it is shown in Figure 6.4. The composed strategy, without information hiding, in the usual diagram style is shown in Table 6.15a. The composed strategy, where the inner information is hidden, is shown in Table 6.15b.

6.1.5 Copycat strategies

For any game A, there is a strategy $A \Rightarrow A$ that responds to an O-move in one copy of A by playing the same move in another copy of A, where it is played as a P-move. The identity function in $N \Rightarrow N$ is such an example:

N	$\Rightarrow$	N
		q
q		
n		
		n

The copycat stategy can be composed with other strategies, e.g. in Table 6.17 the strategy of $\lambda f.\ f(3)$ is composed with the copycat strategy. After hiding the middle part, the play of $\lambda f.\ f(3)$ in $(N \Rightarrow N) \Rightarrow N$ remains. We will need the copycat strategies later to define games as categories.

Table 6.15: The composition of the strategies of S and S', where $S||S'$ is the composed strategy hiding inner information.

(a) S and S'

$(N \overset{S}{\Rightarrow} N)$	$\overset{S'}{\Rightarrow} N$
	q
q	
	q
n	
	$n+1$
	$n+2$

(b) $S||S'$

$(N \overset{S\|\|S'}{\Longrightarrow} N)$
q
q
n
$n+2$

Table 6.17: The strategy $\lambda f.\ f(3)$ composed with the copycat strategy.

$(N \Rightarrow N)$	$\overset{\text{copycat}}{\Longrightarrow}$	$(N \Rightarrow N)$	$\overset{\lambda f.\ f(3)}{\Longrightarrow}$	N
				q
		q		
q				
		q		
		3		
3				
n				
		n		
				n

6.2 Categories of Games

We will now give formal definitions of the ideas presented in the previous section. They are based on the definitions introduced by [HO00] and [AJM00]. Both of these definitions were brought together in [AM99], which we will follow in this section. Similar ideas were also introduced by [Nic94].

Games are deeply rooted in category theory, and we will describe two basic categories. One is $\mathcal{G}$, which is (almost) a model of a fragment of intuitionistic linear logic. The other one is $\mathcal{C}$, that is a cartesian closed category built from $\mathcal{G}$ by using the Girard translation [Gir87] from $A \Rightarrow B$ to $!A \multimap B$. The morphisms of both categories are strategies. By adding constraints of well-bracketedness and innocence, we get several subcategories of $\mathcal{C}$ and $\mathcal{G}$.

6.2.1 Fundamental Definitions

Definition 6.2.1 (Arena [AM99])**.** An *arena* A is specified by a structure $\langle M_A, \lambda_A, \vdash_A \rangle$ consisting of

- a set of moves M_A;
- a labelling function $\lambda_A : M_A \to \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$, that indicates if a move is made by the opponent O or the player P, and if a move is a question Q or an answer A. The set $\{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$ is written as $\{\mathsf{OQ}, \mathsf{OA}, \mathsf{PQ}, \mathsf{PA}\}$. λ_A^{OP} indicates λ_A followed by left projection, i.e. $\lambda_A^{\mathsf{OP}}(m) = \mathsf{O}$ if $\lambda_A(m) = \mathsf{OQ}$ or $\lambda_A(m) = \mathsf{OA}$. We define λ_A^{QA} analogously. Finally, $\overline{\lambda_A}$ is λ_A with the O/P part reversed, such that for example
$$\overline{\lambda_A}(m) = \mathsf{OQ} \iff \lambda_A(m) = \mathsf{PQ}.$$
If $\lambda^{\mathsf{OP}}(m) = \mathsf{O}$, we call m an O-move and a P-move otherwise.
- $\vdash_A$ is a relation between $M_A \cup \{*\}$ and M_A, called *enabling*. The star $*$ is a dummy symbol not contained in M_A and the enabling satisfies the following conditions:

  (e1) $* \vdash_A m$ implies that
  - $\lambda_A(m) = \mathsf{OQ}$ and
  - $n \vdash_A m$ if and only if $n = *$;

  (e2) if $m \vdash_A n$ and $\lambda_A^{\mathsf{QA}}(n) = \mathsf{A}$ then $\lambda_A^{\mathsf{QA}}(m) = \mathsf{Q}$;

  (e3) if $m \vdash_A n$ and $m \neq *$ then $\lambda_A^{\mathsf{OP}}(m) \neq \lambda_A^{\mathsf{OP}}(n)$.

The idea behind the enabling relation is that a move can only be played, when it has been enabled before. The enabler $*$ is the initial enabler, it indicates which moves have been enabled on the outer set. A move m with enabling $* \vdash_A m$ is called *initial*. Condition (e1) guarantees that there are no other moves enabling the initial move and the initial move is an opponent question. Condition (e2) says that answers are enabled by questions, and condition (e3) makes sure that protagonists enable each other's moves and not their own.

The arena of natural numbers [ghib] is $N = \langle M_N, \lambda_N, \vdash_N \rangle$. The set of moves M_N contains all natural numbers and an initial question, i.e. $M_N = \mathbb{N} \cup \{q\}$. The labelling function is defined as
$$\lambda_N(n) = \begin{cases} \mathsf{OQ} & \text{for } n = q \\ \mathsf{PA} & \text{for } n \in \mathbb{N}. \end{cases}$$

And the enabling relation is
$$* \vdash_N q, \quad q \vdash_N 0, \quad q \vdash_N 1, \ldots$$

We will now have a look on certain kinds of sequences and operations on those sequences. The concatenation of two sequences s and t is denoted by $s \cdot t$ and the empty sequence is denoted by ε. Given two sequences x and y, we write $x \sqsubseteq y$ if x is a prefix of y, i.e. x has a number of n moves and the first n moves of y match with x.

Furthermore, we already introduced pointers in the informal introduction in Section 6.1. A (justification) pointer is denoted by an arrow from one move n to a prior move m that initiated the move n. It is a pair (m, n) of moves in a sequence s, such that $m < n$ in s.

Definition 6.2.2 (Justified Sequence [AM99])**.** A sequence of moves s in an arena A with associated (justification) pointers is *justified*, if for each non-initial move m there is a pointer to an earlier move n such that $n \vdash_A m$, where $n, m \in s$. It is said that n *justifies* m. The first move in any justified sequence must be initial, and by the rule (e1) of the definition of the $\vdash_A$-relation, justified sequences must start with an opponent question after $*$.

The *proponent view* $\lceil s \rceil$ and *opponent view* $\lfloor s \rfloor$ of a justified sequence s, is defined by induction on $|s|$ as follows (where s and t are sequences of moves and m and n are individual moves):

$$
\begin{aligned}
\lceil \varepsilon \rceil &= \varepsilon. && \\
\lceil s \cdot m \rceil &= \lceil s \rceil \cdot m, && \text{if } m \text{ is a P-move.} \\
\lceil s \cdot m \rceil &= m, && \text{if } * \vdash m. \\
\lceil s \cdot \; m \overset{\curvearrowleft}{\cdot} t \cdot \; n \rceil &= \lceil s \rceil \cdot \; m \overset{\curvearrowleft}{\cdot} n \;, && \text{if } n \text{ is an O-move.} \\
\lfloor \varepsilon \rfloor &= \varepsilon. && \\
\lfloor s \cdot m \rfloor &= \lfloor s \rfloor \cdot m, && \text{if } m \text{ is an O-move.} \\
\lfloor s \cdot \; m \overset{\curvearrowleft}{\cdot} t \cdot \; n \rfloor &= \lfloor s \rfloor \cdot \; m \overset{\curvearrowleft}{\cdot} n \;, && \text{if } n \text{ is a P-move.}
\end{aligned}
$$

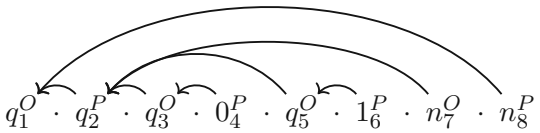
The view of a justified sequence does not need to be justified itself, the appearance of a move m does not guarantee the appearance of its justifier. This can be solved by adding a *visibility condition.*

Definition 6.2.3 (Legal Sequence [AM99])**.** A justified sequence s is legal, or a legal position, if it satisfies the following *alternation* and *visibility conditions:*

- Players alternate: if $s = s_1 mns_2$ then $\lambda^{OP}(m) \neq \lambda^{OP}(n)$.
- If $tm \sqsubseteq s$ where m is a P-move, then the justifier of m occurs in $\lceil t \rceil$.
- If $tm \sqsubseteq s$ where m is a non-initial O-move, then the justifier m occurs in $\lfloor t \rfloor$.

The set of legal positions of A is denoted by L_A.

We use the example of $\lambda f.f\ 0\ 1$ of Subsection 6.1.1 to show how to determine legal positions. We numbered each move $x_1, x_2, ...$ to distinguish them and marked each move with x^P for a proponent move and x^O for an opponent move, the sequence of moves s with justification pointers is given as follows:



The players clearly alternate, i.e. no player makes a move twice in a row, which satisfies the first condition. For the second and third conditions, we need to check each prefix of our sequence s. First, we check each prefix $tm \sqsubseteq s$ where m is a P-move:

- $tm = q_1 \cdot q_2$
  - $m = q_2$
  - $\lceil t \rceil = q_1$
  - The justifier of m is q_1 and q_1 is in $\lceil t \rceil$.
- $tm = q_1 \cdot q_2 \cdot q_3 \cdot 0_4$
  - $m = 0_4$
  - $\lceil t \rceil = q_1 \cdot q_2 \cdot q_3$
  - The justifier of m is q_3 and q_3 is in $\lceil t \rceil$.
- $tm = q_1 \cdot q_2 \cdot q_3 \cdot 0_4 \cdot q_5 \cdot 1_6$
  - $m = 1_6$
  - $\lceil t \rceil = q_1 \cdot q_2 \cdot q_5$
  - The justifier of m is q_4 and q_4 is in $\lceil t \rceil$.
- $tm = q_1 \cdot q_2 \cdot q_3 \cdot 0_4 \cdot q_5 \cdot 1_6 \cdot n_7 \cdot n_8$
  - $m = n_8$
  - $\lceil t \rceil = q_1 \cdot q_2 \cdot n_7$
  - The justifier of m is q_1 and q_1 is in $\lceil t \rceil$.

Second, we check each prefix $tm \sqsubseteq s$ where m is a non-initial O-move:

- $tm = q_1 \cdot q_2 \cdot q_3$
  - $m = q_3$
  - $\lfloor t \rfloor = q_1 \cdot q_2$

  - The justifier of m is q_2 and q_2 is in $\lfloor t \rfloor$.

- $tm = q_1 \cdot q_2 \cdot q_3 \cdot 0_4 \cdot q_5$
  - $m = q_5$
  - $\lfloor t \rfloor = q_1 \cdot q_2 \cdot q_3 \cdot 0_4$
  - The justifier of m is q_2 and q_2 is in $\lfloor t \rfloor$.
- $tm = q_1 \cdot q_2 \cdot q_3 \cdot 0_4 \cdot q_5 \cdot 1_6 \cdot n_7$
  - $m = n_7$
  - $\lfloor t \rfloor = q_1 \cdot q_2 \cdot q_3 \cdot 0_4 \cdot q_5 \cdot 1_6$
  - The justifier of m is q_2 and q_2 is in $\lfloor t \rfloor$.

6.2.2 Games and Strategies

Definition 6.2.4 (Hereditarily justified [AM99])**.** Let s be a legal position in an arena A and let m be a move in s. The move m is *hereditarily justified* by an occurrence of a move n in s if the chain of justifications leads back from m to n. In more detail, m is justified by an intermediate move m_1, which is then justified by a move m_2. This is continued until we reach some move m_k, that is justified by the move n. $s \rightharpoonup n$ denotes the subsequence of s containing all moves that are hereditarily justified by n. Similarly, we can define $s \rightharpoonup I$ for a set I, that contains moves in s. The subsequence $s \rightharpoonup I$ then contains all moves hereditarily justified by some move of I.

Definition 6.2.5 (Game [AM99])**.** A *game* A is specified by a structure $\langle M_A, \lambda_A, \vdash_A, P_A \rangle$ where

- $\langle M_A, \lambda_A, \vdash_A \rangle$ is an arena.
- P_A is a non-empty, prefix-closed[1] subset of L_A, called the *valid positions*, satisfying the following condition:

$$\text{if } s \in P_A \text{ and } I \text{ is a set of initial moves of } s \text{ then } s \rightharpoonup I \in P_A.$$

Definition 6.2.6 (Multiplicatives of Games [AM99])**.** Given games A and B, we can define new games $A \otimes B$ and $A \multimap B$ as follows:

$$\begin{aligned}
M_{A\otimes B} &= M_A + M_B. \\
\lambda_{A\otimes B} &= [\lambda_A, \lambda_B]. \\
* \vdash_{A\otimes B} n &\iff * \vdash_A n \vee * \vdash_B n. \\
m \vdash_{A\otimes B} n &\iff m \vdash_A n \vee m \vdash_B n. \\
P_{A\otimes B} &= \{s \in L_{A\otimes B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}
\end{aligned}$$

[1] If the sequence $s \cdot m$ is in the set, then so must be s. In particular, every prefix-closed set contains the empty sequence.

$$
\begin{array}{rcl}
M_{A \multimap B} & = & M_A + M_B. \\
\lambda_{A \multimap B} & = & [\overline{\lambda_A}, \lambda_B]. \\
* \vdash_{A \multimap B} m & \Longleftrightarrow & * \vdash_B m. \\
m \vdash_{A \multimap B} n & \Longleftrightarrow & m \vdash_A n \vee m \vdash_B n \vee \\
 & & [* \vdash_B m \wedge * \vdash_A n], \text{ for } m \neq *. \\
P_{A \multimap B} & = & \{s \in L_{A \multimap B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}.
\end{array}
$$

A few things need to be detailed more clearly here. In this case, the union $M_A + M_B$ is always assumed to be disjoint, i.e. $M_A \cap M_B = \emptyset$. The subsequence of s that consists of moves in A is denoted by $s \upharpoonright A$. Finally, the labelling function is combined as follows:

$$
[\lambda_A, \lambda_B](x) = \begin{cases} \lambda_A(x), & \text{if } x \in M_A \\ \lambda_B(x), & \text{if } x \in M_B. \end{cases}
$$

$A \otimes B$ is known as the tensor product. Each participant O and P can decide to play in subgame A or in subgame B alternatingly. It is a form of disjoint, i.e. non-communicating, parallel composition [A$^+$97]. It will happen automatically, that P can only move in the game that O has just played in, but O is allowed to switch games [Hyl97]. The unit of the tensor product is defined as $I = \langle \emptyset, \emptyset, \emptyset, \{\varepsilon\} \rangle$.

$A \multimap B$ is similar to $A \otimes B$, but the labelling function on the moves of A is inverted in order to model composed games. In these games the roles of P and O are interchanged on the left side of the arrow [A$^+$97]. In this type of multiplicative, it will automatically happen that O must start in B, then P has the choice of playing in either A or B, and O can now only move in the game in which P has just moved [Hyl97].

Definition 6.2.7 (Strategy [AM99])**.** A *strategy* σ for a game A is a non-empty set of even-length positions from P_A, satisfying the following conditions:

(s1) $sab \in \sigma \Rightarrow s \in \sigma$

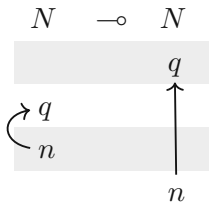
(s2) $sab, sac \in \sigma \Rightarrow b = c$, and also the justifier of b and must be the same as the justifier of c.

In order to form a cartesian closed category with games and strategies, it is necessary to define an *identity* strategy. We already intuitively introduced this concept as the copycat strategy in Subsection 6.1.5. The identity strategy for any game A is played between A and another copy of A, connected by $\multimap$, i.e. $A \multimap A$. It is formally defined by

$$
id_A = \{s \in P_{A_1 \multimap A_2} \mid \forall t \sqsubseteq^{even} s.(t \upharpoonright A_1 = t \upharpoonright A_2)\}.
$$

If t is an even-length prefix of s, we write $t \sqsubseteq^{even} s$. The strategy id_A copies the opponent's move in one copy of A into another copy of A. The next move made by the

player is then justified by a copy of the justifier of the opponent's move. For an example, we refer again to the copycat strategy. There, the strategy for $N \Rightarrow N$ (or $N \multimap N$) was displayed. Explicitly showing pointers, we get:



In the second step, the move q from the right part of the game is copied to the left part. And in the final step, the move n from the left part is copied to the right part, as well as the justification pointer from n to q.

6.2.3 Composition

We will from now denote $\sigma : A \multimap B$, to indicate that σ is a strategy for the game $A \multimap B$.

The composition of two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ is denoted by $\sigma;\tau : A \multimap C$. First, we will define how to move justification pointers.

Definition 6.2.8 ([AM99])**.** Let u be a sequence of moves in games A, B, and C together with justification pointers from all moves, except initial moves in C. Let $u \upharpoonright B, C$ be the subsequence of u consisting of all moves in B and C. If there is a move with a pointer from one of these points to A, we delete that pointer. Similarly, we define $u \upharpoonright A, B$. The sequence u is an *interaction sequence* of A, B, and C if $u \upharpoonright A, B \in P_{A \multimap B}$ and $u \upharpoonright B, C \in P_{B \multimap C}$. The set of all interaction sequences is denoted $int(A, B, C)$.

If $u \in int(A, B, C)$, then any pointer from one A-move must point to another A-move or to an initial B-move. That B-move must have a pointer to an initial C-move. A pointer from a C-move must always be to another C-move. We can define $u \upharpoonright A, C$ as the subsequence of u, that consists of all moves in A and C, except for the just described cases, where the pointer from A is mapped to the initial move of C directly.

Given the strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$, we define

$$\sigma || \tau = \{u \in int(A, B, C) \mid u \upharpoonright A, B \in \sigma \wedge u \upharpoonright B, C \in \tau\}.$$

Proposition 6.2.1 (Composite of two strategies [AM99])**.** If $\sigma : A \multimap B$ and $\tau : B \multimap C$, we define $\sigma;\tau : A \multimap C$ as

$$\sigma;\tau = \{u \upharpoonright A, C \mid u \in \sigma || \tau\}.$$

6.2.4 Constraining strategies

We will present two conditions, that can restrict strategies in a relevant way.

We already mentioned that *innocence* is related to referential transparency. We would like to explain this in more detail. Consider the following non-innocent, imperative program in pseudo-code:

```
f1(y :: Nat) := {x := y + 1; return x}

f2(y :: Nat) := {y := y + 1; return y}

g(h :: Nat -> Nat) := {x := 0;
      if isZero(h(x)) then x else x}
```

The function `g` declares a new variable `x` with an initial value of 0. The function `ifzero` forces to evaluate the function `h`, in case of lazy evaluation. If the function $\mathtt{f}_1$ is passed as an argument for the parameter `h`, it evaluates to 1 in this context. However, it overwrites the value of `x` to 1, which causes the overall function `g` to return 1. This is known as a side-effect, the inner function changes the value of an outer variable. On the other hand, when passing $\mathtt{f}_2$, we do not overwrite the value of `x`, causing `g` to return 0.

A strategy is innocent if its response only depends on the local "functional context" of the position. If we have a look at the play of `g` in Table 6.21, we see that there is only one evaluation. The evaluation of `g` does not depend on the parameter `h` in this case, because `x` is returned in both cases of the `if` ... `else` term. To explain this in more detail, we need the formal definition:

Definition 6.2.9 (Innocence [AM99])**.** Given positions $sab, ta \in L_A$, where sab has even length and $\lceil sa \rceil = \lceil ta \rceil$, there must be a unique extension to ta by the move b such that $\lceil sab \rceil = \lceil tab \rceil$. We call this extension $match(sab, ta)$. The strategy $\sigma : A$ is *innocent* if and only if it satisfies

$$sab \in \sigma \wedge t \in \sigma \wedge ta \in P_A \wedge \lceil ta \rceil = \lceil sa \rceil \Rightarrow match(sab, ta) \in \sigma.$$

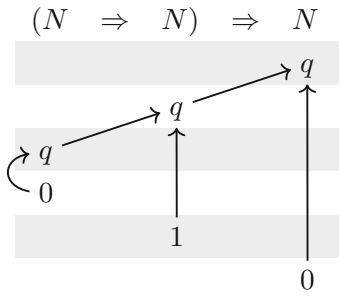
This means, that the move and pointer played by an innocent strategy σ at a position sa is determined by the P-view $\lceil sa \rceil$.

In our example, the play excluding the last step looks as follows:

$$q \cdot q \cdot q \cdot 0 \cdot 1.$$

As the last move is an O-move, the P-view, by removing the moves surrounded by pointers, is given by $q \cdot q \cdot 1$. In this case, we can even set $s = t$, and there must be a unique extension b. But in our example there would be two possible extensions of $\mathtt{f}_1$

Table 6.21: The play of the function `g`.



or $\mathtt{f}_2$. As a result, our imperative program is clearly not innocent and also not purely functional.

The second constraint is *well-bracketedness*. One example of an operator that can create program phrases that are not well-bracketed is `catch` of type `(Nat -> Nat) -> Nat`. It has the following property:

$$\texttt{catch f} = \begin{cases} 0 & \text{if } \texttt{f} \text{ calls its argument,} \\ n+1 & \text{if } \texttt{f} \text{ returns } n \text{ without calling its argument.} \end{cases}$$

We will have a look at the following two functions of type `Nat -> Nat`:

```
\x -> 1
\x -> x + 1
```

The first function does not use its argument, but the second one does. Using these functions as a parameter for `catch`, we get 2 in the first case and 0 in the second as an overall result. Looking at the plays of both strategies in Tables 6.22a and 6.22b, we see more clearly what happens. In the second case, some questions are left dangling, while an outer question has been answered. This kind of property is also not desired in regular bracketing, as the inner most bracket usually needs to be closed first. There are many other well-known control operators in various programming languages, that show a similar result, such as `goto` or `break`. The purely functional programming paradigm follows a properly nested call-return principle, which rules out such operations. The formal definition is given as follows:

Definition 6.2.10 (Well-bracketedness [AM99])**.** A strategy $\sigma : A$ is *well-bracketed* if and only if for every $sab \in \sigma$ with b as an answer, the justification pointers have the form

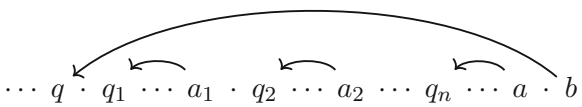


Table 6.22: The play of the `catch`-operator...

(a) ... using `\x -> 1` as a parameter for `f`.

$(N$	$\Rightarrow$	$N)$	$\Rightarrow$	N
				q
		q		
		1		
				2

(b) ... using `\x -> x + 1` as a parameter.

$(N$	$\Rightarrow$	$N)$	$\Rightarrow$	N
				q
		q		
q				
				0

where the moves $a_1, a_2, ...$ and a are answers. This means that when P gives an answer, the view only contains the most recent question. O-moves are not required to satisfy this condition.

6.2.5 Four categories of games

We now start to connect the theory of games with category theory. The category $\mathcal{G}$ has games as objects. For two objects A and B, the morphisms $A \to B$ are the strategies $A \multimap B$. The identity arrow is given by the identity strategy and the composite of two arrows σ and τ is given by $\sigma;\tau$. Furthermore, there are three subcategories of the category $\mathcal{G}$. The morphisms of the subcategory $\mathcal{G}_i$ are only the innocent strategies, the morphisms of $\mathcal{G}_b$ are Well-bracketed strategies, and the morphisms of $\mathcal{G}_{ib}$ are only innocent and well-bracketed strategies. The following proposition explicitly states the desired result.

Proposition 6.2.2 ([AM99])**.** $\mathcal{G}, \mathcal{G}_i, \mathcal{G}_b$, and $\mathcal{G}_{ib}$ are categories.

This statement entails that composition is well-defined, associative, and there is a identity strategy in all categories. Additionally, innocence and well-bracketedness are preserved in the respective categories. This was proved in [A$^+$97, Hyl97, McC12].

6.2.6 Products

Given two games A and B, the game $A\&B$ is defined as follows:

$$\begin{aligned}
M_{A\&B} &= M_A + M_B. \\
\lambda_{A\&B} &= [\lambda_A, \lambda_B]. \\
* \vdash_{A\&B} n &\iff * \vdash_A n \vee * \vdash_B n. \\
m \vdash_{A\&B} n &\iff m \vdash_A n \vee m \vdash_B n. \\
P_{A\&B} &= \{s \in L_{A\&B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B = \varepsilon\} \\
&\cup \quad \{s \in L_{A\&B} \mid s \upharpoonright B \in P_B \wedge s \upharpoonright A = \varepsilon\}
\end{aligned}$$

The projection arrows $\pi_1 : A\&B \to A$ and $\pi_2 : A\&B \to B$ are given by the copycat strategies. Given the arrows $\sigma : C \to A$ and $\tau : C \to B$, we need a unique arrow $\langle \sigma, \tau \rangle : C \to A\&B$ to complete the commutative diagram (see Definition 4.1.6). It is defined as

$$\begin{aligned}\langle \sigma, \tau \rangle = \{s \in L_{C \to A\&B} \mid s \upharpoonright C, A \in \sigma \wedge s \upharpoonright B = \varepsilon\} \\ \cup \{s \in L_{C \to A\&B} \mid s \upharpoonright C, B \in \tau \wedge s \upharpoonright A = \varepsilon\}.\end{aligned}$$

Theorem 6.2.1 ([AM99])**.** $A\&B$ is a product of A and B in each of the four categories $\mathcal{G}, \mathcal{G}_i, \mathcal{G}_b$, and $\mathcal{G}_{ib}$, with the projections given by π_1 and π_2.

6.2.7 Exponentials

Before defininining the exponential we need an intermediate construct. The !-operator has its roots in linear logic [Gir87]. Intuitively, it is used to indicate instances $A_0, A_1, A_2, ...$ (or copies) of A [Hyl97].

Definition 6.2.11 ([AM99])**.** Given a game A, the game $!A$ is defined as follows:

$$\begin{aligned}M_{!A} &= M_A \\ \lambda_{!A} &= \lambda_A \\ \vdash_{!A} &= \vdash_A \\ P_{!A} &= \{s \in L_{!A} \mid \text{for each initial move } m, s \rightharpoonup m \in P_A\}.\end{aligned}$$

We denoted the exponential between two objects A and B in general as B^A in Chapter 4. The exponential in this context can now be defined as follows:

Theorem 6.2.2 (Exponential in the category $\mathcal{G}$ [AM99])**.** For two strategies A and B, the *exponential* is given by the strategy

$$\sigma : !A \multimap B.$$

In some cases, we might want to chain multiple strategies together, e.g. $\sigma : !A \multimap B$ and $\tau : !B \multimap C$. Thus, it might be useful to transform the strategy from $\sigma : !A \multimap B$ to $\sigma^\dagger : !A \multimap !B$, where

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{for all initial } m, s \rightharpoonup m \in \sigma\}.$$

The strategy $\sigma^\dagger$ can be seen as playing several copies of σ. However, $\sigma^\dagger$ does not exist in general, but it is defined in the class of well-opened games.

Definition 6.2.12 (Well-opened [AM99])**.** A game A is *well-opened* iff for all $sm \in P_A$ with m initial, $s = \varepsilon$.

In well-opened games, the first move must be an initial move, ensuring that there is only one thread of dialogue. If B is well-opened, then $A \multimap B$ is also opened for any game A. In particular, $!A \multimap B$ is well-opened whenever B is.

6.2.8 Four cartesian closed categories of games

The category $\mathcal{C}$ is defined as follows:

$$\begin{aligned} \text{Objects} &: \text{Well-opened games} \\ \text{Morphisms } \sigma : A \to B &: \text{Strategies for } !A \multimap B \end{aligned}$$

The *identity* map for a well-opened game A is defined by the copycat strategy $d : !A \multimap A$:

$$d = \{s \in P_{!A \multimap A} \mid \forall t \sqsubseteq^{even} s.(t \upharpoonright !A = t \upharpoonright A)\}.$$

The composite of two strategies is given by the following definition.

Definition 6.2.13 (Composite morphism [AM99])**.** Given morphisms $\sigma : A \to B$ and $\tau : B \to C$, with the corresponding strategies $\sigma : !A \multimap B$ and $\tau : !B \multimap C$, the *composite morphism* $\sigma \fatsemi \tau : A \to C$ is $\sigma^{\dagger}; \tau$.

Following our given structure of cccs in Subsection 4.1.2, we need a terminal object, a product and an exponential object.

Definition 6.2.14 ([AM99])**.** The category $\mathcal{C}$ is a ccc and the constructs are given as follows:

- The *terminal object* $\mathbf{1}$ is given by the empty game I.
- The *product* $A \times B$ is given by $A \& B$.
- The *exponential* B^A is defined as $!A \multimap B$.

Further reasoning for this is given in [AM99], we will just present the central proposition.

Proposition 6.2.3 (Sub-categories [AM99])**.** $\mathcal{C}$ is a cartesian closed category, with sub-cccs $\mathcal{C}_i$ and $\mathcal{C}_b$. The category $\mathcal{C}_{ib}$ is a sub-ccc of each $\mathcal{C}, \mathcal{C}_i$, and $\mathcal{C}_b$.

As explained before in the category $\mathcal{G}$, the morphisms of $\mathcal{C}_i$ are innocent strageties, the morphisms of $\mathcal{C}_b$ are well-bracketed strategies, and the morphisms of $\mathcal{C}_{ib}$ are only innocent and well-bracketed strategies.

Having a look at our examples in Subsection 6.2.4, it makes sense, that $\mathcal{C}_{ib}$ models purely functional programming. Furthermore, if the property of innocence is lost, i.e. we have $\mathcal{C}_b$, then we are able to model functional programming with states such as in the programming language of Idealized Algol. On the other hand, if we keep the innocence property and drop the well-bracketed property ($\mathcal{C}_i$), we are able to model programming languages with control operators, such as `catch` in our example. If we drop both properties, we are able to model programming languages with states and control operators. This was introduced as the syntactic and semantic square in [AM97] and also explained in [AM99].

6.3 A Fully Abstract Model of PCF

We described in Section 4, that cartesian closed categories are able to model simply-typed λ-calculus. Indeed, this is also the case for $\mathcal{C}$ and its subcategories. The model is described in the sense of Definition 4.3.2. Furthermore, we will show soundness and adequacy of this model and how the previously unsolved issue of definability is fulfilled in this model. Finally, we will show a proof idea for full abstraction. We will mainly follow the structure of [AM99].

First, we need to model types and terms in general. A type A will be simply modeled as the object $[\![A]\!]$. Types of the form $[\![\mathtt{A} \to \mathtt{B}]\!]$ are build inducively by the strategy for the exponential $![\![\mathtt{A}]\!] \multimap [\![\mathtt{B}]\!]$.

For an open term, let the basis be $\Gamma = x_1 :: A_1, ..., x_n :: A_n$. The interpretation of the term $\Gamma \vdash M :: A$ is then given as a strategy:

$$[\![\Gamma \vdash M :: A]\!] : [\![A_1]\!] \,\&\, ... \,\&\, [\![A_n]\!] \multimap [\![A]\!].$$

When the context Γ is empty, then the empty game is used, and for a term $\vdash M :: A$ we write $[\![M :: A]\!] : 1 \to [\![A]\!]$. The interpretation of further terms is given inductively on the structure of terms.

We restrict PCF to the only base type `Nat`, which is interpreted by the game N of natural numbers. It is defined as follows:

$$\begin{array}{rcll} M_N & = & \{q\} \cup \{n \mid n \in \omega\} & \\ \lambda_N(q) & = & \mathsf{OQ} & \\ \lambda_N(n) & = & \mathsf{PA} & \\ * & \vdash_N & q & \\ q & \vdash_N & n & \text{for each } n \\ P_N & = & \{\varepsilon, q\} \cup \{qn \mid n \in \omega\}. & \end{array}$$

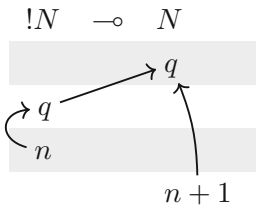
The game N has a single initial question q to which P can respond by playing a natural number (denoted by $n \in \omega$). We intuitively described it at the beginning of Section 6.1. The strategies for N are $\bot = \{\varepsilon\}$ and $[\![n]\!] = \{\varepsilon, qn\}$ for each number n. These strategies are innocent and well-bracketed, so each of the cartesian closed categories includes the interpretation of numeric constants.

Now let us look at the operation `succ`, which is interpreted as the strategy showed in Table 6.25.

This strategy gives a map $s : [\![\mathtt{Nat}]\!] \to [\![\mathtt{Nat}]\!]$, and given $\Gamma \vdash M :: \mathtt{Nat}$, we can set

$$[\![\Gamma \vdash \mathtt{succ}\ M :: \mathtt{Nat}]\!] = ([\![\Gamma \vdash M :: \mathtt{Nat}]\!] \fatsemi s) : [\![\Gamma]\!] \multimap [\![\mathtt{Nat}]\!].$$

Table 6.25: The strategy of the operator succ.



For the conditional, we use the strategy that has two typical plays, one for True and one for False.

$!N$	$\otimes$	$!N$	$\otimes$	$!N$	$\multimap$	N
						q
q						
0						
		q				
		n				
						n

$!N$	$\otimes$	$!N$	$\otimes$	$!N$	$\multimap$	N
						q
q						
$m \neq 0$						
				q		
				n		
						n

O asks the initial question, and P responds with a question for the first $!N$. If O plays 0, then P asks a question about the second $!N$ and the response is copied to the overall output. If O responds with a non-zero number m in the first $!N$, then P asks for the value of the third $!N$ and the that response is copied to the output. The map defined by the strategy is $c : N \times N \times N \to N$, since $!(N \;\&\; N \;\&\; N) = !N \otimes !N \otimes !N$. We can then interpret $\Gamma \vdash \mathtt{if}\ M\ N_1\ N_2$ as

$$[\![\Gamma \vdash \mathtt{if}\ M\ N_1\ N_2]\!] = \langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N_1]\!], [\![\Gamma \vdash N_2]\!] \rangle \fatsemi c.$$

Before we continue with the interpretation of recursive terms, we need to introduce a new concept. A category is *cpo-enriched* if the Hom-sets of that category form a cpo [Sco00]. This means we also need to assume that the collections of arrows between any two categories are sets. Hom-sets were introduced in Definition 4.1.3. The categories presented in this Section are cpo-enriched. We will not go into further detail, but the proof idea is given in [AM99].

Since all categories are cpo-enriched, it is possible to interpret recursive terms $\mathtt{fix}\ M$ with a standard denotational approach. The interpretation of a term $\Gamma \vdash M : A \to A$ determines a map $f : [\![\Gamma]\!] \times [\![A]\!] \to [\![A]\!]$ by uncurrying. The chain of maps $f_n : [\![\Gamma]\!] \to [\![A]\!]$ needed to interpret the fixed-point operator is defined as follows:

$$f_0 = [\![\Omega_A]\!] : [\![\Gamma]\!] \to [\![A]\!]$$
$$f_{n+1} = \langle \mathsf{id}_{[\![\Gamma]\!]}, f_n \rangle \fatsemi f.$$

We can now set $[\![\texttt{fix}\ M]\!]$ to be the least fixed point of this chain (see Definition 3.2.6). In particular, this is the least fixed point of the operation taking a map $g : [\![\Gamma]\!] \to [\![A]\!]$ to $\langle \mathsf{id}_{[\![\Gamma]\!]}, g\rangle \fatsemi f$. We defined the term Ω_A as $\texttt{fix}_A(\lambda x : A.x)$ in Definition 2.3.1.

A second approach to interpreting fixed-points in a more game-semantic style was given by [HO00]. For any arena A, a strategy $(A \to A) \to A$ is described, with three different types of moves in different copies of A:

- *The main O-component:* The moves of this component are justified by the opening O-move in A.
- *The P-components:* These moves are justified by an opening P-move in A, but not by P-moves that are dependent on it. There can be multiple P-components.
- *Subsidiary O-components:* Moves that are justified by a sequence of three initial moves, with labelling OQ, PQ, and OQ.

The process is carried out by copying moves and creating dual components. The first P-component is the dual of the main O-component. The following P-components are the duals of the subsidiary O-components respectively. After each P-move the duals are copies of each other. The structure is shown in Figure 6.5. The strategy can be described in three cases that can occur after an O-move:

1. The move was the opening move, P copies it to create the first P-component.
2. O opens a new subsidiary component, P copies the move to create a new P-component.
3. O moves in an existing O- or P-component, the move is copied in the dual P- or O-component.

We will use this approach to give a rather informal example computation of our addition (Listing 2.1). The steps are given in Table 6.26. As usual, the opponent starts the computation by asking the value of the function, this step is marked as part of the initial O-component. P copies this move to a new P-component. The copying of moves is best illustrated in the copycat method in Section 6.1. In the next step, O continues the play and asks for the first parameter to the function. This step is copied again from the P-component to the dual (initial) O-component. The strategy then continues as usual, meaning that O continues the play and P copies the moves to the respective dual component. In step 11, O opens a new subsidiary O-component, to which P opens another P-component. The play essentially continues the same way as in the previous iteration, except that O provides different parameters this time. Since the first parameter is 0 now, there is no further component opened and 3 is returned. From there, O copies the result to the P-components and P copies it to the O-components.

Table 6.26: The iterations of the recursive addition of $1 + 2$.

	P-component	Subs. O-component	P-component	Initial O-component	
	$(N \Rightarrow N \Rightarrow N)$	$(N \Rightarrow N \Rightarrow N)$	$(N \Rightarrow N \Rightarrow N)$	$(N \Rightarrow N \Rightarrow N)$	
1				q	O
2			q		P
3			q		O
4				q	P
5				1	O
6			1		P
7			q		O
8				q	P
9				2	O
10			2		P
11		q			O
12	q				P
13	q				O
14		q			P
15		0			O
16	0				P
17	q				O
18		q			P
19		3			O
20	3				P
21	3				O
22		3			P
23			3		O
24				3	P

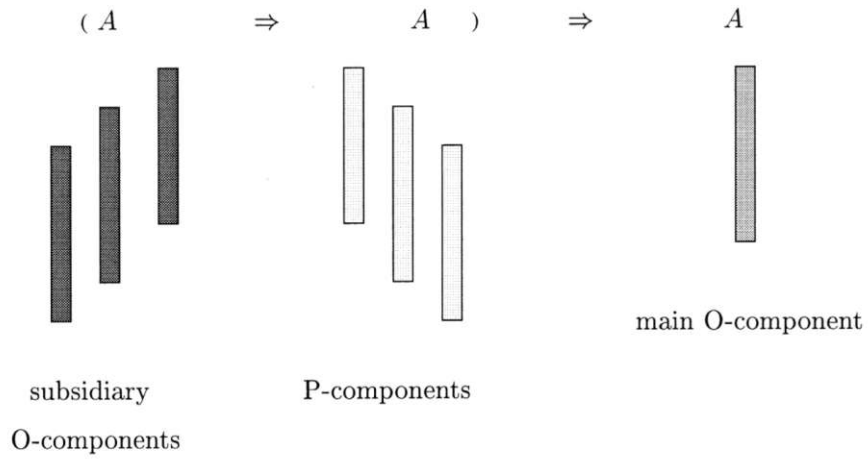


Figure 6.5: The components of `fix` :: $(A \to A) \to A$. [HO00]

6.3.1 The full abstraction proof

The detailed proof of full abstraction for the category $\mathcal{C}_{ib}$ is extensive and involves numerous steps. The overall goal is to show that the category $\mathcal{C}_{ib}$ is intensionally fully abstract, i.e. every element needs to be computable and definable in PCF. We will just give the main ideas behind the reasoning and leave the details to be looked up in [AM99].

The first major step is a proof of soundness and adequacy. It holds for all four categories $\mathcal{C}, \mathcal{C}_i, \mathcal{C}_b$, and $\mathcal{C}_{ib}$. This is done by proving soundness first:

Theorem 6.3.1 (Soundness [AM99])**.** If $M \Downarrow V$ then $[\![M]\!] = [\![V]\!]$.

This is proven by induction on the operational semantics rules, that we presented in Section 2.3

The proof of adequacy is based on a fact that follows from computational adequacy (Proposition 2.5.1): if $[\![M]\!] \neq [\![\Omega]\!] = \bot$ then $M \Downarrow$. Furthermore, Plotkin's definition of computability is used:

Definition 6.3.1 (Plotkin Computability [AM99])**.** A term is *Plotkin computable* if

- it is of the form $\vdash M :$ `Nat` and $[\![M]\!]$ implies $M \Downarrow$.
- it is of the form $\vdash M : A \to B$ and $\vdash MN : B$ is computable for all computable $\vdash N : A$.
- it is of the form $x_1 : A_1, ..., x_n : A_n \vdash M : A$ and for all closed computable $N_1 : A_1, ..., N_n : A_n$, the term $\vdash M[N_1/x_1, ..., N_n/x_n] : A$ is computable.

All terms of PCF must be shown to be Plotkin computable. In an intermediate step, computability is proven for PCF_1, which restricts the fixed-point operator of PCF to only terms of the form $\texttt{fix}\ (\lambda x.x)$. This allows to prove the following statement for PCF:

Theorem 6.3.2 ([AM99])**.** For any term M,

$$[\![M]\!] = \bigcup_{n\in\omega} [\![M_n]\!],$$

where M_n is the PCF_1 term created from M by replacing each term $\texttt{fix}\ N$ in M with $\texttt{fix}^n N$, i.e. the recursion is restricted to n iterations.

Finally, the desired result is achieved:

Theorem 6.3.3 ([AM99])**.** All terms of PCF are Plotkin computable.

The second major step is the proof of definability, i.e. for every object c in $\mathcal{C}_{ib}$ there is a term M such that $[\![M]\!] = c$. This is only possible for an extension of PCF, denoted by PCF'. In this new language the regular conditional (`if :: Nat -> Nat -> Nat -> Nat`) is extended with k-ary conditionals of the form

$$\mathsf{case}_k\ M\ N_1\ N_2\ \cdots\ N_k,$$

where N_j is selected as a result, when $M = j$. This extension is harmless, because the k-ary conditional can be easily simulated by k nested conditionals in regular PCF.

Finally, the abstraction result does not hold in the category $\mathcal{C}_{ib}$, but only in the category $\mathcal{C}_{ib}/\lesssim_{ib}$ quotiented by the intrinsic preorder. We will omit the details of this category in this work, but the intrinsic preorder is defined as

$$\sigma \lesssim_{ib} \tau \text{ iff } \forall \alpha : A \to \Sigma.(\sigma \fatsemi \alpha = \top \implies \tau \fatsemi \alpha = \top).$$

An important fact is that it needs to be proven that $\mathcal{C}_{ib}/\lesssim_{ib}$ is a model of PCF at all, as it is not known if this new category is cpo-enriched. For this, it is enough to prove the following statement:

Proposition 6.3.1. For any closed PCF terms M and N of the same type,

$$[\![M]\!] \lesssim_{ib} [\![N]\!] \iff M \precsim_{obs} N.$$

This concludes the steps of the full abstraction proof of PCF.

CHAPTER 7

Current State of the Art

7.1 Concurrent Games

Nondeterministic concurrent games and strategies were first introduced by [RW11], with the intention to formalize distributed games in which both player and opponent can interact in a distributed fashion, without enforcing alternation of moves.

The underlying structure with labelling functions is replaced with *event structures*. The labelling of O- and P-moves are modelled by polarities. In fact, player and opponent can be now seen as teams of players and opponents. We will see that it is still possible to model classical PCF with event structures, but will also see a model of an extension: probabilistic PCF. The definition is given as follows:

Definition 7.1.1 (Event structure [RW11])**.** An *event structure* $(E, Con, \leq)$ consists of a set E of events, a *causal dependency relation* $\leq$, and a non-empty *consistency relation* Con. The set E is partially ordered by $\leq$, and Con consists of finite subsets of E that satisfy

- $\{e' | e' \leq e\}$ is finite for all $e \in E$,
- $\{e\} \in Con$ for all $e \in E$,
- $Y \subseteq X \in Con \implies Y \in Con$, and
- $X \in Con \wedge e \leq e' \in X \implies X \cup \{e\} \in Con$.

The dependency relation $a \leq b$ implies that the event b can only occur after a.

Furthermore, $\mathcal{C}(E)$ is a set of *configurations* on an event structure E. Each configuration contains finite subsets $x \subseteq E$ which are

- *Consistent:* $x \in Con$, and
- *Down-closed:* $\forall e, e' \,.\, e' \leq e \in x \implies e' \in x$.

Games and strategies can be represented by an event structure with polarity. The *polarity function* $pol : E \to \{+, -\}$ assigns a polarity $+$ or $-$ to each of the events of E. Each event corresponds to a move, and the polarity expresses the assignment of moves to either player or opponent.

Let $x, y \in \mathcal{C}(E)$. If there exists $e \in E$ such that $y = x \cup \{e\}$, we say that y *covers* x. This is denoted by $x \multimap y$ or even more precisely by $x \multimap^{e} y$. The conditions of event structures ensure that there is a covering chain for any $x \in \mathcal{C}(E)$, i.e. there is a sequence of configurations $x_1, ..., x_n$ such that

$$\emptyset \multimap x_1 \multimap \ ... \multimap x_n \multimap x.$$

Covering chains do not necessarily need to be unique.

The formal definitions of game and strategy in terms of events is given in [Paq20].

Definition 7.1.2 (Game [Paq20])**.** A *game* is an event structure with polarity.

Definition 7.1.3 (Strategy [Paq20])**.** For a game A, a *strategy* on A consists of an event structure S with polarity, together with a map of event structures $\sigma : S \to A$ which is

- *receptive:* if $x \in \mathcal{C}(S)$ and $\sigma(x) \subseteq^{-} y$ for some $y \in \mathcal{C}(A)$, there exists a unique $x' \in \mathcal{C}(S)$ such that $x \subseteq x'$ and $\sigma(x') = y$;
- *courteous:* if $e, e' \in S$ are such that $e \rightarrowtriangle e'$ and $\sigma(e) \not\rightarrowtriangle \sigma(e')$, then $pol(e) = -$ and $pol(e') = +$.

The arrow $x \rightarrowtriangle y$ indicates that there is a immediate causality between events x and y:

Definition 7.1.4 (Immediate Causality [Paq20])**.** The events x and y are related by $x \rightarrowtriangle y$, if $x \leq y$ and $x \neq y$, i.e. $x < y$, and there is no event z such that $x < z < y$.

$x \subseteq^{-} y$ is defined as a subset, where every $e \in y \backslash x$ has a negative polarity, and $x, y \in \mathcal{C}(E)$ (Remember that $\mathcal{C}(E)$ is a set of subsets of E). *Receptivity* ensures that the player lets the opponent play all of the currently available moves of the game. *Courtesy* states that only additional causal dependencies can be specified.

This framework allows us to model parallel processes, as shown in the example in [Paq20]. It shows a set of three events a, b, and c having a partial order $\leq$, where $a \leq c$ and $b \leq c$. The events a and b are concurrent, i.e. they are causally independent. This is depicted in Figure 7.1.

If we take the same set of events, but define the relation $\leq$ only by $a \leq c$, then a and b are incompatible events. This is shown in Figure 7.2.

Both of these relations of event are combined in the previously defined event structures.

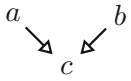


Figure 7.1: The partial order for three events, where c is directly dependent on a and b [Paq20].

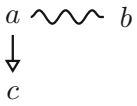


Figure 7.2: The partial order for three events, where a and b are incompatible [Paq20].

7.1.1 Symmetry

Symmetry in concurrent games was introduced by [CCW14], to be able to express plays that are essentially the same. Furthermore, symmetry allows us to play moves of a game more than once, this is not possible in the previous definition in terms of event structures [Paq20]. We will just give the basic definitions of this rather complex and detailed topic, further definitions and explanations can be found in [Paq20, CCW14, Win07].

Definition 7.1.5 (Symmetry [CCW14])**.** Let E be an event structure and the *symmetry* $\theta : x \cong_E y$ be non-empty set of bijections, i.e. a subset of $\mathcal{C}(E \times E)$, that satisfy the following conditions:

1. a) For all identities $id_x : x \cong_E x$ holds;
   b) if $\theta : x \cong_E y$ then the inverse $\theta^{-1} : y \cong_E x$ holds;
   c) and if $\theta : x \cong_E y$ and $\varphi : y \cong_E z$, then the composition is $\varphi \circ \theta : x \cong_E z$.
2. For $\theta : x \cong_E y$ whenever $x' \subseteq x$, then there is a unique $y' \subseteq y$ such that there is a restriction $\theta' : x' \cong_E y'$.
3. For $\theta : x \cong_E y$ whenever $x \subseteq x'$, then there is an extension of θ to θ' such that $\theta' : x' \cong_E y'$ for some y' with $y \subseteq y'$.

Definition 7.1.6 (Event structure with symmetry [Paq20])**.** An *event structure with symmetry (and polarity)* is a pair $\mathcal{E} = (E, \cong_E)$ of an event structure and an isomorphism family on it.

Definition 7.1.7 (Game with symmetry [Paq20])**.** A *game with symmetry* is a tuple $\mathcal{A} = (A, \cong_A, \cong_A^+, \cong_A^-)$, where A is a game and $\cong_A, \cong_A^+$, and $\cong_A^-$ are three isomorphism families on A such that

- the families $\cong_A^+$ and $\cong_A^-$ are sub-families of $\cong_A$;
- if $\theta \in \cong_A^+ \cap \cong_A^-$, then θ is the identity bijection;

- if $\theta \in \cong_A^-$ and $\theta \subseteq^- \theta'$ for some $\theta' \in \cong_A$ then $\theta' \in \cong_A^-$; and
- if $\theta \in \cong_A^+$ and $\theta \subseteq^+ \theta'$ for some $\theta' \in \cong_A$ then $\theta' \in \cong_A^+$.

Furthermore, we can also define an arena in the context of concurrent games with symmetry.

Definition 7.1.8 (Arena [Paq20])**.** An *arena* is a game with symmetry $\mathcal{A}$ whose underlying event structure with symmetry A is

- *forest-shaped:* if $a \leq b$ and $c \leq b$ then either $a \leq c$ or $c \leq a$; and
- *alternating:* if $a \rightarrowtriangle b$ then $pol(a) = -pol(b)$.

If all of the initial moves of an arena $\mathcal{A}$ have negative polarity, then the arena is called *negative*. This holds analogously for *positive* arenas. An arena $\mathcal{A}$ is *polarized*, if it is either negative or positive.

7.1.2 A model of PCF

It is possible to construct an intensionally fully abstract model of PCF using concurrent semantics with symmetry [CCW15]. PCF is a sequential language, but some independent sub-computations can be still computed in parallel.

For example, in an if-function one could evaluate the second and third arguments without waiting for the call to the first argument to terminate. To make this possible, an *immediate causality* relation is used, that maintains sequentiality. In regular game semantics, this relation is not needed, since calls are represented in a sequential order from top to bottom.

The detailed construction of such a model, the relation to Hyland-Ong games, and a full-abstraction proof are given in [CCW15].

7.2 Probabilistic Concurrent Game Semantics

We start this section with an example game $\mathcal{A}$ with consistency, polarity, and symmetry. It contains three events: a^-, b^+, and c^+. The strategy $\sigma : \mathcal{S} \to \mathcal{A}$ displayed in Figure 7.3 shows a non-deterministic play. The player waits for the opponent to make a move a and then the player chooses one of the moves b or c.

This example can be extended with a probability parameter, such that every non-deterministic branch is assigned a value. This is displayed in Figure 7.4. The probabilities p_b and p_c are positive real numbers and $p_b + p_c \leq 1$. The probability can be also less than 1, because the player can choose not to play any move.

The idea of assigning probability to games was first introduced by [Win13], and summed up in [Paq20], which we will present in this section.

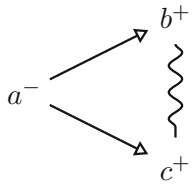


Figure 7.3: A nondeterministic strategy, where either b^+ or c^+ is played [Paq20].

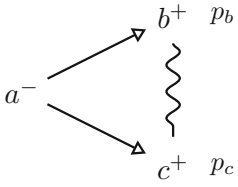


Figure 7.4: A nondeterministic strategy, where the moves b^+ and c^+ are assigned a probability [Paq20].

However, assigning probability to event structures with polarity can become more subtle, when we do not have simple non-deterministic branching. An event structure can be *tree-shaped*, for example, when we remove the incompatibility between b^+ and c^+. This strategy $\tau : \mathcal{T} \to \mathcal{A}$ is depicted in Figure 7.5. The player does no longer have a non-deterministic choice, but b and c could be both played. In this case, we would still like to know the probabilities of the two events, but we might also be interested in their probabilistic dependency. In fact, it makes sense to assign probability coefficients to configurations as opposed to individual events.

The idea of events and dependencies might already sound familiar and clearly it is a hint at conditional dependency. The basic definition, which can be found in any introductory book about probability, is given as follows:

Definition 7.2.1 (Conditional probability [GW14])**.** If some events A and B are given, where $P(B) > 0$, then the conditional probability of A given B is defined by

$$P(A|B) = \frac{P(A \cap B)}{P(B)},$$

where $P(A \cap B)$ denotes the probability that both A and B occur.

Coefficients are assigned to *positive extensions* of x, for each $x \in \mathcal{C}(E)$, i.e. we assign a coefficient to each $y \in \mathcal{C}(E)$ such that $x \subseteq^+ y$. This is denoted by the valuation $v(y|x)$, inspired by conditional probability, i.e. y will occur when x occurred. To be consistent with conditional probability, we must have the property $v(x|x) = 1$ and a *chain rule* $v(z|x) = v(y|x)v(z|y)$, when $x \subseteq^+ y \subseteq^+ z$.

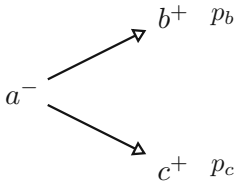


Figure 7.5: A strategy, where the moves b^+ and c^+ are not incompatible, leaving potentially both to be played in the same execution. [Paq20]

The sum $\sum_{x \subseteq^+ y} v(y|x)$ must be ≤ 1, if the extensions of x are pairwise incompatible. If the extensions are not pairwise mutually exclusive, then the overlap must be taken care of with the inclusion-exclusion principle. The third condition (3), also called *drop condition*, in the following definition represents that principle. Condition (4) formalizes, that whenever player and opponent are causally independent, then they must be probabilistically independent as well. Condition (5) ensures that the player plays *symmetric* configurations with equal probability.

Definition 7.2.2 (Conditional valuation [Paq20])**.** A *conditional valuation* on an event structure with polarity is a family $(v(y|x))_{x \subseteq^+ y}$ of coefficients in $[0, 1]$ satisfying

(1) $v(x|x) = 1$ for all $x \in \mathcal{C}(S)$;

(2) if $x \subseteq^+ y \subseteq^+ z$ then $v(z|x) = v(y|z)v(z|y)$;

(3) if $x \subseteq^+ y_1, ..., y_n$, then
$$\sum_I (-1)^{|I|+1} v(\bigcup_{i \in I} y_i | x) \leq 1$$
where I ranges over non-empty subsets of $1, ..., n$ such that $\bigcup_{i \in I} y_i \in \mathcal{C}(S)$;

(4) if $x \subseteq^+ y, x \subseteq^- z$, and $y \cup z \in \mathcal{C}(S)$, then $v(y|x) = v(y \cup z|z)$;

(5) if $\theta : x \cong_S y$ and $\theta \subseteq^+ \theta' : x' \cong_S y'$, then $v(x'|x) = v(y'|y)$.

Leaving out the conditional probabilities, we can also define a function $v : \mathcal{C}(S) \to [0, 1]$:

Definition 7.2.3 (Valuation [Paq20])**.** A *valuation* on an event structure with polarity S is a function $v : \mathcal{C}(S) \to [0, 1]$ satisfying

(1) $v(\emptyset) = 1$;

(2) $v(x) = v(y)$ when $x \subseteq^- y$;

(3) for every $x \subseteq^+ y_1, ..., y_n$,
$$v(x) \geq \sum_I (-1)^{|I|+1} v(\bigcup_{i \in I} y_i)$$
where I ranges over non-empty subsets of $\{1, ..., n\}$ such that $\bigcup_{i \in I} y_i \in \mathcal{C}(S)$;

(4) if $\theta : x \cong_S y$, then $v(x) = v(y)$.

A *conditional valuation* on $\mathcal{S}$ yields a *valuation* on $\mathcal{S}$. Conditional valuations are more explicit and intuitive. However, valuations enable a cleaner definition of composition of strategies.

Definition 7.2.4 (Probabilistic strategy [Paq20])**.** A *probabilistic strategy* on a game $\mathcal{A}$ is a strategy $\sigma : \mathcal{S} \to \mathcal{A}$ together with a valuation v_S on $\mathcal{S}$.

We will show two classes of probabilistic strategies:

- In *Markov strategies*, any two player actions are probabilistically independent, conditioned by their causal history.
- In *deterministic strategies*, the behaviour of the player is fully determined by that of the opponent.

7.2.1 Markov Strategies

We recall the strategy $\tau : \mathcal{T} \to \mathcal{A}$ from Figure 7.5. By condition (3) of the definition of valuation, the valuation v on $\mathcal{T}$ must respect $v(\{a, b\}) + v(\{a, c\}) - v(\{a, b, c\}) \leq 1$. In a Markov strategy the two moves b and c are probabilistically independent and, therefore $v(\{a, b\})v(\{a, c\})$ is suitable for $v(\{a, b, c\})$, because it always satisfies the above requirement. The formal definition of a Markov strategy is given as follows:

Definition 7.2.5 ([Paq20])**.** A probabilistic strategy $\sigma : \mathcal{S} \to \mathcal{A}$ is *Markov*, if for any $y, z \in \mathcal{C}(\mathcal{S})$ it holds that $y \cup z \in \mathcal{C}(\mathcal{S})$ and $v(y \cup z)v(y \cap z) = v(y)v(z)$.

Equivalently, this definition can be written as

$$v(y \cup z | y \cap z) = v(y | y \cap z) \cdot v(z | y \cap z),$$

where $y \cap z$ is the common history that both probabilistically independent events y and z have.

Additionally, we have a more general factorization result for more than two moves:

Theorem 7.2.1 ([Paq20])**.** A *strategy* $\sigma : \mathcal{S} \to \mathcal{A}$ is Markov if and only if for all $x \in \mathcal{C}(\mathcal{S})$

$$v(x) = \prod_{\substack{e \in x \\ pol(e) = +}} v([e], [e] \backslash \{e\}),$$

where $[e]$ is the causal history of an event e: $\{e' \in E \mid e' \leq e\}$.

Figure 7.6: A game with a conflict between a player and an opponent move. [Paq20]

7.2.2 Deterministic Strategies

For a deterministic strategy we need to recall the strategy in Figure 7.3, where b^+ and c^+ are incompatible. A player should behave in a way, that assigning probability 1 for every configuration yields a valid strategy. Of course, this cannot be the case for our example, because assigning 1 to $\{a, b\}$ and $\{a, c\}$ violates condition (3) of the definition of valuations. So for deterministic strategies, we must exclude this case and make the behavior of the player completely determined by that of the opponent.

Definition 7.2.6 ([Paq20])**.** A strategy $\sigma : \mathcal{S} \to \mathcal{A}$ is *deterministic* when for every $x \in \mathcal{C}(\mathcal{S})$, and $x \multimap^{s}$, $x \multimap^{s'}$ with $pol(s) = +$, we have $x \cup \{s, s'\} \in \mathcal{C}(\mathcal{S})$.

The notation $x \multimap^{s}$ indicates that there is an s such that $x \cup s \in \mathcal{C}(S)$.

Theorem 7.2.2 ([Paq20])**.** If $\sigma : \mathcal{S} \to \mathcal{A}$ is deterministic, then the map $v : \mathcal{C}(\mathcal{S}) \to [0, 1]$ as $v(x) = 1$ for every $x \in \mathcal{C}(\mathcal{S})$ satisfies the axioms for a valuation.

Race-free games

The other direction of the previous theorem does not hold, because of *races*. A *race* is a minimal conflict between a player and an opponent move, shown in game $\mathcal{B}$ depicted in Figure 7.6. Races do not mix well with probability, since a move might be set to a certain probability $p \in [0, 1]$, but a player might not be able to play that move. To avoid this, "race-free" games were introduced.

Definition 7.2.7 (Race-freeness [Paq20])**.** An event structure with polarity E is *race-free* if for all $x \in \mathcal{C}(E)$, with $x \subseteq^{+} y$ and $x \subseteq^{-} z$ it holds that y and z are compatible, i.e. $y \cup z \in \mathcal{C}(E)$.

Furthermore, we call games and strategies race-free, when the underlying event structure with polarity is race free.

Theorem 7.2.3 ([Paq20])**.** If $\mathcal{A}$ is a race-free game, then any strategy $\sigma : \mathcal{S} \to \mathcal{A}$ is race-free.

7.2.3 The bicategory PG

First, we need the definition of parallel composition:

Definition 7.2.8 (Parallel composition [Paq20])**.** Given a family $(A_i)_{i\in I}$ of event structures with polarity, their *parallel composition* has events

$$\|_{i\in I} A_i = \bigcup_{i\in I} A_i \times \{i\}$$

with componentwise causality and polarity. The consistent sets are those of the form $\|_{i\in I} X_i$ for I, such that $X_i \in \mathrm{Con}_{A_i}$ for each $i \in I$.

A bicategory in general, is a set of objects, that are called *0-cells*, where each pair of 0-cells x, y forms a category $B(x, y)$, where the objects are called *1-cells* and the morphisms are called *2-cells*. Furthermore, there are a few conditions to fulfill, that we will leave out here. But a detailed definition and description can be found in [nLa21].

Definition 7.2.9 (2-Cells [Paq20])**.** The 2-cells in the bicategory of arenas and probabilistic strategies are maps of strategies with an additional property for valuations:

A map of *probabilistic strategies* from $\sigma : \mathcal{S} \to \mathcal{A}^\perp \parallel B$ to $\sigma' : \mathcal{S} \to \mathcal{A}^\perp \parallel \mathcal{B}$ is a map $f : \sigma \to \sigma'$ satisfying

$$v_S(x) \leq v_{S'}(f(x))$$

for every $x \in \mathcal{C}(\mathcal{S})$. The strategy $\mathcal{A}^\perp$ is the dual of the event structure $\mathcal{A}$ with polarities. This means that the polarities of $\mathcal{A}^\perp$ are reversed.

Using all these definitions, we get a bicategory:

Theorem 7.2.4 (Bicategory PG [Paq20])**.** There is a bicategory **PG** having:

- *objects:* race-free arenas;
- *morphisms* $\mathcal{A} \to \mathcal{B}$: probabilistic strategies;
- *2-cells:* maps of probabilistic strategies.

7.2.4 Probabilistic PCF

Probabilistic PCF (PPCF) is an extension to regular PCF with a probabilistic primitive. We will give an introduction to this language, as well as its model, presented in [Paq20, CCPW18].

Syntax and Operational Semantics

The *types* of PPCF are the same as in regular PCF, i.e. we have `Nat`, `Bool`, and a type `A -> B`.

The *terms* are as follows:

```
M, N ::= \x . M | M N | x | True | False | 0 | if M N1 N2 | fix M
       | pred M | succ M | isZero M | coin
```

The only new addition is coin of type Bool, that evaluates to either True or False with a probability of $\frac{1}{2}$. To model this probability, the reduction relation of PCF is generalized to a probabilistic reduction relation $\xrightarrow{p}$ for $p \in [0,1]$. For coin, this means $\texttt{coin} \xrightarrow{1/2} b$, where $b \in \{\texttt{True}, \texttt{False}\}$.

The reduction relation is defined by the following rules, where n is of type Nat and b is of type Bool:

$$(\backslash \texttt{x} \ . \ \texttt{M})\ \texttt{N} \xrightarrow{1} \texttt{M[N/x]} \qquad \texttt{if True}\ \texttt{N}_1\ \texttt{N}_2 \xrightarrow{1} \texttt{N}_1 \qquad \texttt{if False}\ \texttt{N}_1\ \texttt{N}_2 \xrightarrow{1} \texttt{N}_2$$

$$\texttt{isZero 0} \xrightarrow{1} \texttt{True} \qquad \texttt{isZero (n + 1)} \xrightarrow{1} \texttt{False} \qquad \texttt{fix M} \xrightarrow{1} \texttt{M (fix M)}$$

$$\texttt{pred (n + 1)} \xrightarrow{1} \texttt{n} \qquad \texttt{succ n} \xrightarrow{1} \texttt{n + 1}$$

$$\frac{\texttt{M} \xrightarrow{p} \texttt{M'}}{\texttt{M N} \xrightarrow{p} \texttt{M' N}} \qquad \frac{\texttt{M} \xrightarrow{p} \texttt{M'}}{\texttt{succ M} \xrightarrow{p} \texttt{succ M'}} \qquad \frac{\texttt{M} \xrightarrow{p} \texttt{M'}}{\texttt{pred M} \xrightarrow{p} \texttt{pred M'}}$$

$$\texttt{coin} \xrightarrow{\frac{1}{2}} \texttt{b} \text{ if } \texttt{b} \in \{\texttt{True}, \texttt{False}\}$$

$$\frac{\texttt{M} \xrightarrow{p} \texttt{M'}}{\texttt{isZero M} \xrightarrow{p} \texttt{isZero M'}} \qquad \frac{\texttt{M} \xrightarrow{p} \texttt{M'}}{\texttt{if M N P} \xrightarrow{p} \texttt{if M' N P}}$$

The probability reduction $Pr(\texttt{M} \to \texttt{N})$ for any M and N is given by the sum over all applied reduction rules. As reduction is non-deterministic, it might result in countably many reduction paths of the form

$$M = M_0 \xrightarrow{p_1} \ldots \xrightarrow{p_n} M_n = N.$$

Observational preorder and equivalence is now defined as follows:

Definition 7.2.10 (Contextual preorder and equivalence [CCPW18])**.** Let M and N be PPCF terms such that $\Gamma \vdash \texttt{M} : \texttt{A}$ and $\Gamma \vdash \texttt{N} : \texttt{A}$. We write $\texttt{M} \lesssim_{ctx} \texttt{N}$ if for every context $C[\ ]$ such that $\vdash C[\texttt{P}] : \texttt{Bool}$ for every $\Gamma \vdash \texttt{P} : \texttt{A}$,

$$Pr(C[\texttt{M}] \to \texttt{b}) \leq Pr(C[\texttt{N}] \to \texttt{b})$$

for $\texttt{b} \in \{\texttt{True}, \texttt{False}\}$. The equivalence by this preorder, *contextual equivalence*, is denoted by $\simeq_{ctx}$.

Denotational Semantics

For the game-semantical model, we start with the interpretation of the PPCF ground types, which are interpreted as arenas, shown in Figure 7.7.

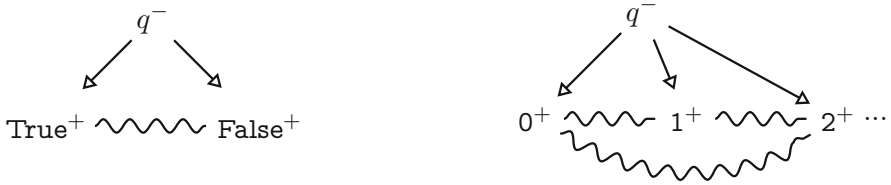


Figure 7.7: The arenas interpreting the types Bool and Nat, denoted by $[\![\mathtt{Bool}]\!]$, $[\![\mathtt{Nat}]\!]$ respectively. [Paq20]

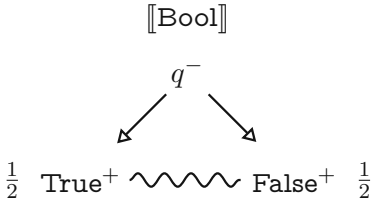


Figure 7.8: The strategy for $[\![\mathtt{coin}]\!]$. [Paq20]

From these two arenas, we can interpret higher types inductively using $[\![A \to B]\!] = [\![A]\!] \multimap [\![B]\!]$. As usual, every play starts by an opponent's question q^-, i.e. of negative polarity, and some player's answer of positive polarity. For higher types and subgames these roles can be inversed. For example in the arena for $\mathtt{Bool} \to \mathtt{Bool}$, the opponent would ask for the value of $\mathtt{Bool} \to \mathtt{Bool}$, and the player would answer with a question for Bool. In this Bool-subgame the roles of player and opponent would be reversed, i.e. the player starts the game with a question q^+ and the opponent provides a value.

The semantics of PPCF are described by the bicategory $\mathbf{PG}_!^{\mathrm{bsi}}$, where the objects are arenas and the morphisms are well-bracketed, sequential, innocent strategies. Furthermore, the exclamation mark (!) indicates that the category contains exponentials.

The standard methodology is to assign an arena $[\![A]\!]$ to each type A, and a strategy $[\![M]\!]^\Gamma : ![\![\Gamma]\!] \to [\![A]\!]$ to each open term $\Gamma \vdash M : A$. Where the semantics for the context Γ are given by the product: $[\![x_1 : A_1, ..., x_n : A_n]\!] = \&_{i=1}^n [\![A_i]\!]$. The context is the terminal object $\mathbf{1}$ if $n = 0$.

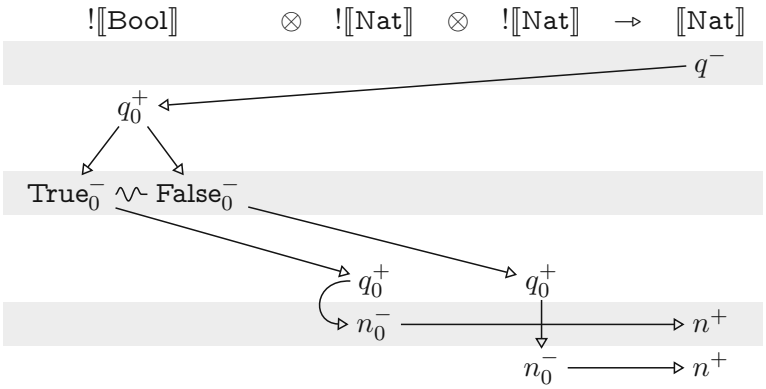
Constants are interpreted as the strategies $[\![b]\!] : 1 \to [\![\mathtt{Bool}]\!]$ and $[\![n]\!] : 1 \to [\![\mathtt{Nat}]\!]$, where $b \in \{\mathtt{True}, \mathtt{False}\}$ and $n \in \mathbb{N}$. The strategy $[\![\mathtt{Bool}]\!]$ is given by $q^- \multimap b^+$ and $[\![\mathtt{Nat}]\!]$ is given by $q^- \multimap n^+$.

The strategy for **probabilistic choice** is given by a play of $[\![\mathtt{Bool}]\!]$, that returns True or False with a probability of $\frac{1}{2}$, displayed in Figure 7.8.

More complex strategies are given in [Paq20], we will just give the example of the strategy $\mathrm{if}_{\mathtt{Nat}}$. This is shown in Table 7.1.

A major difference to strategies shown in Chapter 6 is that alternative runs are shown

Table 7.1: The strategy for if_{Nat}. [Paq20]



in parallel, as opposed to creating two separate strategies for the outcome of the Bool game.

Definability

The path to full abstraction is shown by definability for *finite* sequential innocent strategies. First, we will define *grounded causal chains* (gcc) and finite strategies.

Definition 7.2.11 (Grounded Causal Chain (gcc) [Paq20])**.** A gcc in a strategy $\sigma : \mathcal{S} \to \mathcal{A}$ is a set of events $\rho = \{s_0, ..., s_n\} \subseteq S$ such that $s_0 \rightarrowtriangle ...s_n$ and s_0 is an initial move in S.

Definition 7.2.12 (Finite [Paq20])**.** We say a sequential innocent strategy $\sigma : \mathcal{S} \to \mathcal{A}^{\perp} \parallel \mathcal{B}$ is finite when:

- for every negative $s \in \mathcal{S}$, the set $\{s' \in \mathcal{S} \mid s \rightarrowtriangle s'\}$ is finite;
- for every positive question $s \in \mathcal{S}$, all but finitely many answers in the set $\{s' \in \mathcal{S} \mid s \rightarrowtriangle s'\}$ are maximal (answers are maximal, if they are answering the latest pending question);
- there is a bound to the length of gccs in $\mathcal{S}$;
- for every $x \in \mathcal{C}(\mathcal{S}), v(x) \in [0, 1] \cap \mathbb{Q}$;

Similar to the definability results in Chapter 6, PPCF needs to be extended with with a k-ary conditional:

$$\mathsf{case}_k\ M\ [N_1 \mid ... \mid N_k].$$

Finite definability is given as follows:

Theorem 7.2.5 (Finite definability [Paq20])**.** Let A be a PPCF type, and let $\sigma : \mathcal{S} \to [\![A]\!]$ be a finite, innocent sequential strategy such that v_s is non-zero. Then there is a (PPCF + case) term M such that $\vdash M : A$ and $[\![M]\!] \cong \sigma$.

From here, the proof is similar to the original abstraction proof that we presented in Chapter 6. The proof of finite definability and full abstraction is shown in more detail in [CCPW18], where also an alternative full abstraction result by *relational collapse* can be found.

CHAPTER 8

Conclusion and Further Work

8.1 Conclusion

We showed the historical development of semantical models of programming languages with a major focus on the language PCF. The idea of full abstraction, in order to model observationally equivalent program phrases with denotationally equal phrases, was anticipated in the 1970s. The first major publications were made by [Mil77] and [Plo77]. Using the models introduced by [Sco70] was not quite precise enough to achieve the desired full abstraction result. A later refinement [Sco82], while being very promising, was shown to fail at definability. There are functions recognized in the model, that could not be expressed in the programming language.

Later, category theory was used as a mathematical basis for mainly sequential approaches. Especially, the property of a cartesian closed category is very useful, as a CCC is a model of untyped λ-calculus. Although sequential functions and algorithms did not achieve the desired results, they were a step in the right direction.

Game semantics had a longstanding tradition in logic [Car17], from intuitionistic logic [Lor67], and from linear logic [LS91, Bla92] to analyis of classical provability [Coq95]. They provided a crucial new component to sequential analysis of PCF, that solved the full abstraction problem: assigning moves to a player or an opponent. First papers using this idea were released in 1997 [A$^+$97, Hyl97], and investigated further in [AM99, HO00], which are the central works discussing game semantics for PCF.

Since then, many extensions and further investigations were introduced, which we will discuss in the following section. We chose to present one branch in more detail: concurrent and, in particular, probabilistic games using event structures [RW11, CCHW18, Paq20]. The latter shows a probabilistic extension to PCF, that includes a term for a probabilistic coin toss, with the possibility for non-determinism for any program term. As it was shown, this language also has a fully abstract model.

8.2 Further Work

8.2.1 Imperative Languages

Game semantics can be applied to the imperative language paradigm as well as the functional paradigm presented in this work. The programming language Idealized Algol (IA) is used as a starting point to give an example of an imperative function paradigm. In this paradigm, a new challenge has to be taken into account, when trying to substitute terms for each other, because imperative languages contain states.

IA can be constructed from PCF by adding two base types. The type `com` for commands, that change the state and the type `var` which stores natural numbers. Furthermore, there is an operation for sequential composition. A full abstraction result was shown in [AM99]. Idealized Algol can be extended to allow concurrent execution [ghib].

A similar path to probability as in PCF has also been followed by [DH02]. They presented a probabilistic extension to IA, also using a term for coin toss with probability of 1/2.

8.2.2 Session Types

An alternative formulation to game semantics are *session types*. They revolve around the same concept of sending messages between entities. Instead of games and strategies, the messages are based on session types and π-calculus. The relation of these two semantic models is shown in [CY19]. Game semantics based on event-structures, as we presented them in Chapter 7, can be seen as a link between synchronous π-calculus and asynchronous game semantics.

Session types go back to [THK94] and [HVK98], and they were proposed to formalize information structures in concurrent processes. In both cases, game semantics or session types, protocols are two-party. Output actions are corresponding to player moves and input actions are corresponding to opponent moves.

Although the relation between these two formalisms was already known quite early [HO95], the precise connection was only formally described in [CY19]. Furthermore, a translation was described and implemented.

8.2.3 Measurable Concurrent Games

Probabilistic concurrent games, such as the ones in Chapter 7, can be further generalized by allowing continuous types, such as a type for real numbers. In other words, the discrete probability is generalized to a continuous setting. The notion of valuation function presented in Chapter 7 does not generalize well for this application, so sub-probability measures with kernels are introduced. A detailed formal description of measurable games are given in [Paq20].

8.2.4 Trace Semantics

Trace semantics have been defined to model various state based systems with forms of probability or non-determinism. Different underlying structures were presented that can be used to formally model this type of semantics, e.g. coinduction [HJS07] or coalgebras [Jac04, JSS12]. The advantage of these semantics is that the opponent can be treated as an adversary rather than a partner, this useful when a real-life-program interacts with other modules and security is of concern. The semantics enable to hide certain information from the opponent, who is seen more as a run-time system than a syntactic context. An introduction to traces is given in [ghib].

8.2.5 Applications

There is a good overview of different applications of game semantics [Ghi09]. One of the lines of research presented is games-based software model checking. Interesting programming language fragments are selected for which the game-semantical representation is finite, and on those fragments various model-checking techniques are used to verify certain properties.

Another line is synthesizing digital circuits from behavioral specifications, also known as hardware synthesis. In this field a definitive solution has not yet been found, but game semantics offer a promising approach to solving this problem. Structuring hardware computations using different process calculi has shown good intermediate abstractions for hardware.

Index

Bibliography

[A+97] Samson Abramsky et al. Semantics of interaction: an introduction to game semantics. *Semantics and Logics of Computation*, 14(1), 1997.

[Abr14] Samson Abramsky. Axioms for definability and full completeness. *CoRR*, abs/1401.4735, 2014.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for pcf. *Information and Computation*, 163(2):409 – 470, 2000.

[AM97] Samson Abramsky and Guy McCusker. *Linearity, Sharing and State: A Fully Abstract Game Semantics for Idealized Algol with Active Expressions*, pages 297–329. Birkhäuser Boston, Boston, MA, 1997.

[AM99] Samson Abramsky and Guy McCusker. Game semantics. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, pages 1–55, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Awo10] Steve Awodey. *Category theory*. Oxford university press, 2010.

[BC82] Gérard Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(3):265–321, 1982.

[BCL85] Gérard Berry, Pierre-Louis Curien, and Jean-Jacques Lévy. Full abstraction for sequential languages: the state of the art. *Algebraic methods in semantics*, 89132, 1985.

[Ber78] Gérard Berry. Stable models of typed λ-calculi. In *International Colloquium on Automata, Languages, and Programming*, pages 72–89. Springer, 1978.

[Ber79] Gérard Berry. *Modeles completement adéquats et stables des lambda-calculs typés*. PhD thesis, Éditeur inconnu, 1979.

[Bir40] Garrett Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940.

[Bla92] Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56(1):183 – 220, 1992.

[Car17] Felice Cardone. Games, full abstraction and full completeness. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017.

[CCF94] R. Cartwright, P.L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297 – 401, 1994.

[CCHW18] Simon Castellan, Pierre Clairambault, Jonathan Hayman, and Glynn Winskel. Non-angelic concurrent game semantics. In *International Conference on Foundations of Software Science and Computation Structures*, pages 3–19. Springer, Cham, 2018.

[CCPW18] Simon Castellan, Pierre Clairambault, Hugo Paquet, and Glynn Winskel. The concurrent game semantics of probabilistic pcf. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 215–224, New York, NY, USA, 2018. Association for Computing Machinery.

[CCW14] Simon Castellan, Pierre Clairambault, and Glynn Winskel. Symmetry in concurrent games. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[CCW15] S. Castellan, P. Clairambault, and G. Winskel. The parallel intensionally fully abstract games model of pcf. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 232–243, 2015.

[CF92] Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, page 328–342, New York, NY, USA, 1992. Association for Computing Machinery.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversation*. Princeton University Press, 1941.

[Coc16] Robin Cockett. Basic programming for computable functions, bpcf. `https://pages.cpsc.ucalgary.ca/~robin/class/521/types/fixtypes.pdf`, 2016. Accessed: 2020-10-24.

[Coq95] Thierry Coquand. A semantics of evidence for classical arithmetic. *The Journal of Symbolic Logic*, 60(1):325–337, 1995.

[Cro94] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1994.

[Cur86] Pierre-Louis Curien. *Categorical combinators, sequential algorithms and functional programming*. Research notes in theoretical computer science. Pitman [u.a.], London [u.a.], 1986.

[Cur12] P-L Curien. *Categorical combinators, sequential algorithms, and functional programming*. Springer Science & Business Media, 2012.

[CY19] Simon Castellan and Nobuko Yoshida. Two sides of the same coin: Session types and game semantics: A synchronous side and an asynchronous side. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[DH02] Vincent Danos and Russell S. Harmer. Probabilistic game semantics. *ACM Trans. Comput. Logic*, 3(3):359–382, July 2002.

[Dyb04] Peter Dybjer. Computability via pcf, lecture notes. `http://www.cse.chalmers.se/edu/year/2017/course/DAT140_Types/PCF.pdf`, February 2004. [Online; accessed 23-August-2020].

[ghia] Game Semantics [1/4] - Dan R. Ghica - OPLSS 2018. `https://www.youtube.com/watch?v=EpGGenaS-mQ`. Accessed: 2020-11-19.

[ghib] Game Semantics an elementary approach. `https://www.cs.uoregon.edu/research/summerschool/summer18/lectures/ghica1.pdf`. Accessed: 2020-11-19.

[Ghi09] D. R. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 17–26, 2009.

[Gir86] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theoretical computer science*, 45:159–192, 1986.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

[Gun92] Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

[GW05] Dina Goldin and Peter Wegner. The church-turing thesis: Breaking the myth. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *New Computational Paradigms*, pages 152–168, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[GW14] Geoffrey Grimmett and Dominic Welsh. *Probability: an introduction*. Oxford University Press, 2014.

[Hed19] Jules Hedges. The game semantics of game theory. *CoRR*, abs/1904.11287, 2019.

[HJS07] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4), Nov 2007.

[HO95] JME Hyland and CHL Ong. calculus, dialogue games and pcf. In *Proc. 7th acm Conf. Functional Prog. Lang. Comp. Architecture*, pages 96–107, 1995.

[HO00] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for pcf: I, ii, and iii. *Information and Computation*, 163(2):285–408, 2000.

[Hu18] Nick Hu. Cartesian closed categories and the simply-typed lambda-calculus. `https://nickhu.co.uk/uthesis.pdf`, 2018. [Online; accessed 30-December-2020].

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[Hyl97] Martin Hyland. Game semantics. *Semantics and logics of computation*, 14:131, 1997.

[Jac04] Bart Jacobs. Trace semantics for coalgebras. *Electronic Notes in Theoretical Computer Science*, 106:167–184, 2004. Proceedings of the Workshop on Coalgebraic Methods in Computer Science (CMCS).

[JM96] Trevor Jim and Albert R Meyer. Full abstraction and the context lemma. *Siam Journal on Computing*, 25(3):663–696, 1996.

[JS93] Achim Jung and Allen Stoughton. Studying the fully abstract model of pcf within its continuous function model. In Groote J.F. Bezem M, editor, *Typed Lambda Calculi and Applications*, pages 230–244. Springer, 1993.

[JSS12] Bart Jacobs, Alexandra Silva, and Ana Sokolova. Trace semantics via determinization. In Dirk Pattinson and Lutz Schröder, editors, *Coalgebraic Methods in Computer Science*, pages 109–129, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[KP93] G. Kahn and G.D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1):187 – 277, 1993.

[Lam86] J. Lambek. Cartesian closed categories and typed λ-calculi. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 136–175, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[Loa01] Ralph Loader. Finitary pcf is not decidable. *Theoretical Computer Science*, 266(1-2):341–364, 2001.

[Lor67] P. Lorenzen. Ein dialogisches konstruktivitätskriterium. *Journal of Symbolic Logic*, 32(4):516–516, 1967.

[LS91] Y. Lafont and T. Streicher. Games semantics for linear logic. In *Proceedings 1991 Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 43,44,45,46,47,48,49,50, Los Alamitos, CA, USA, jul 1991. IEEE Computer Society.

[McC12] Guy McCusker. *Games and full abstraction for a functional metalanguage with recursive types*. Springer Science & Business Media, 2012.

[Mil75] Robin Milner. Processes: a mathematical model of computing agents. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 157–173. Elsevier, 1975.

[Mil77] Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[Moo13] Filip Moons. The scott topology. 2013.

[MT16] Andrzej S. Murawski and Nikos Tzevelekos. An invitation to game semantics. *ACM SIGLOG News*, 3(2):56–67, May 2016.

[Mul86a] Ketan Mulmuley. Fully abstract submodels of typed lambda calculi. *Journal of Computer and System Sciences*, 33(1):2 – 46, 1986.

[Mul86b] Ketan D Mulmuley. Full abstraction and semantic equivalence. 1986.

[Mye97] Roger B Myerson. *Game theory: analysis of conflict*. Harvard university press, 1997.

[Nic94] Hanno Nickau. Hereditarily sequential functionals. In Anil Nerode and Yu. V. Matiyasevich, editors, *Logical Foundations of Computer Science*, pages 253–264, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[nLa21] nLab authors. bicategory. `http://ncatlab.org/nlab/show/bicategory`, March 2021. Revision 53.

[Ong95] C. Ong. Correspondence between operational and denotational semantics: the full abstraction problem for pcf. In *LICS 1995*, 1995.

[Pao06] Luca Paolini. A stable programming language. *Information and Computation*, 204(3):339 – 375, 2006.

[Paq20] Hugo Paquet. *Probabilistic concurrent game semantics*. PhD thesis, Ph. D. thesis, University of Cambridge, 2020.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[Plo77] G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.

[Roj15] Raúl Rojas. A tutorial introduction to the lambda calculus. *arXiv preprint arXiv:1503.09060*, 2015.

[RW11] S. Rideau and G. Winskel. Concurrent strategies. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 409–418, 2011.

[Sco69] Dana Scott. *A Construction of a Model for the [lambda] Calculus.* 1969.

[Sco70] Dana Scott. *Outline of a mathematical theory of computation.* Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[Sco82] Dana S. Scott. Domains for denotational semantics. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming*, pages 577–610, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[Sco00] Phill J Scott. Some aspects of categories in computer science. In *Handbook of algebra*, volume 2, pages 3–77. Elsevier, 2000.

[Slo95] Kenneth Slonneger. *Formal Syntax and Semantics of Programming Language*, chapter 10: Domain theory and Fixed-Point Semantics, pages 341–394. Addison-Wesley Publishing Company, 1995.

[Str66] Christopher Strachey. Towards a formal semantics. 1966.

[Sza20] Zoltán Gendler Szabó. Compositionality. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, fall 2020 edition, 2020.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[Wei] Eric W. Weissstein. Topological basis. From MathWorld - A Wolfram Web Resource. `https://mathworld.wolfram.com/TopologicalBasis.html`. Accessed: 2021-02-20.

[Wil12] Stephen Willard. *General topology.* Courier Corporation, 2012.

[Win07] Glynn Winskel. Event structures with symmetry. *Electronic Notes in Theoretical Computer Science*, 172:611–652, 2007. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

[Win13] Glynn Winskel. Distributed probabilistic and quantum strategies. *Electronic Notes in Theoretical Computer Science*, 298:403–425, 2013. Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIX.