Game Semantics and Subtyping

Juliusz Chroboczek

Doctor of Philosophy
University of Edinburgh
2003

Abstract

Game Semantics is a relatively new framework for the description of the semantics of programming languages. By combining the mathematical elegance of Denotational Semantics with explicitly operational concepts, Game Semantics has made possible the direct and intuitive modelling of a large range of programming constructs.

In this thesis, we show how Game Semantics is able to model subtyping. We start by designing an untyped λ-calculus with ground values that explicitly internalises the notion of typing error. We then equip this calculus with a rich typing system that includes quantification (both universal and existential) as well as recursive types.

In a second part, we show how to interpret the untyped calculus; after equipping the domain of the interpretation with an ordering — the *liveness ordering* — loosely inspired from implication on process specifications, we show how our interpretation is both sound and computationally adequate.

In a third part, we introduce a notion of *game* which we use for interpreting types, and show how the liveness ordering on games is suitable for interpreting subtyping. Finally, we prove that under the (unproved) assumption that recursive types are compatible with quantification, our interpretation is sound with respect to both subtyping and typing.

Acknowledgements

I would first like to thank my supervisor, Samson Abramsky, who got me interested in Game Semantics in the first place; none of this work would have been possible without his advice.

A number of people have had a direct or indirect influence on this thesis. I would particularly like to thank Stuart Anderson, Andrea Asperti, Adriana Compagnoni, Pierre-Louis Curien, Vincent Danos, Roberto Di Cosmo, Fabio Gadducci, Russ Harmer and Paul-André Melliès.

I am also very grateful for the help of my examiners, Guy McCusker and Ian Stark.

On a different note, I would like to express my gratefulness to my parents. And I couldn't possibly fail to mention Harley, Conor, Melanie, Carsten, Kostas, Corinna, Tim, Francesca and Francesco, Frédéric, Francesco and Gabriele.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Juliusz Chroboczek*)

Table of Contents

Chapter 1

Introduction

1.1 Game Semantics

When programming languages became important, their designers soon felt the lack of tools for defining and describing them. For the needs of defining syntax, the Backus-Naur Form (BNF) was soon found to be adequate, and was extensively used in the definition of ALGOL 60 [Nau63], but the semantics of programming languages proved more difficult to define; in the case of ALGOL 60, semantics was defined informally. Some later languages were defined using abstract machines, such as Landin's J-Machine [Lan64] and the Vienna Definition Language used for PL/I, or using the unrepeatable transition-based notation of the Algol-68 definition [Wij75]. A number of logic-based specification methods were also developed, the best-known of which is Hoare's logic [Hoa69], later applied to the definition of Pascal [HW73].

Except for Hoare's logic, these early semantic description methods were *operational*, in that they explicitly described transitions in the state of a program's environment, rather than trying to define the meaning of the program. While operational definitions are often easy to write in such a way that they faithfully reflect the intent of the author, they are not directly compositional, meaning that the semantics of a program cannot always be easily derived from the semantics of its parts. They also make the task of proving that two definitions are equivalent (in some suitable sense) very difficult.

In the early 1970s, Christopher Strachey and Dana Scott developed a framework for semantic description initially known as *Mathematical* and later as *Denotational Semantics* [Sto77]. Denotational Semantics ascribes to a program (fragment) an element of a well-defined mathematical structure, such as a lattice or Scott domain, and does so in a fully compositional manner.

While Denotational Semantics turned out to be very successful a tool for studying programming languages, it did not quite faithfully reflect their operational semantics. The technical statement of this failure has become known as the *full-abstraction* problem, which, roughly speaking, says that the equivalence on terms induced by denotational semantics does not, in most cases, exactly coincide with the observational equivalence induced by the operational semantics. The best-known instance of this problem, full-abstraction for the purely functional call-by-name language PCF [Plo77, Mil77], has remained unsolved for decades. Accurately reflecting simultaneously the purely functional and sequential nature of PCF had to wait for Game Semantics [AJM00, HO00].

An alternative view is that Game Semantics combines the elegant mathematical structure of Denotational Semantics with the direct modelling of programmers' intuitions usually reserved to more operational frameworks. Indeed, Denotational Semantics only directly models purely functional calculi. More expressive languages, containing non-functional features such as non-local transfer of control (jumps) or state (imperative variables) require a translation, explicit or implicit, into a purely functional calculus [Mog90]. Game Semantics, being more direct, has allowed for the modelling of a large number of non-functional programming language features in a very short time [AM94, AM95, AM96, Lai97, AHM98, Har99].

In this thesis, we propose to show how to directly model in Game Semantics a ubiquitous feature of programming languages — subtyping.

1.2 Subtyping

Most programming languages include a notion of *type*. One possible point of view is that a programming language intrinsically consists of a countable collection of *terms* (program fragments); types provide a way of categorising terms so that terms that may safely be used in the same contexts are classed together. Languages that include types are said to be *statically typed*; languages that don't are *dynamically typed*.

> **A note about terminology** *Calculi used for modelling programming languages are traditionally divided into* typed calculi *and* untyped calculi. *The terminology used for describing programming languages is different: programming languages based on untyped calculi, such as the Lisp family of languages [McC62, ANS94, RC92] or Smalltalk [GR83], are commonly referred to as* dynamically typed, *by opposition to* statically typed *languages. The term* untyped *is usually reserved to languages in which values of different types may be confused, such as Assembler on an untagged architecture (in which, say, an integer cannot be distinguished from a machine address), or string-processing*

languages such as SNOBOL or Perl (where the empty string can typically be confused with the integer 0). In this thesis, we try as much as possible to follow traditional terminology, using typed *and* untyped *for calculi and* statically *and* dynamically typed *for programming languages.*

Many programming languages order types with a relation known as *subtyping*, which roughly corresponds to the same intuition as the set-theoretic notion of inclusion (subtyping will be further introduced in Section 2.1). In many cases, the term "subtyping" is not used in the description of the language, but the notion is still present: for example, in ML [MTHM90, LRVD99] the type $\forall\alpha.\alpha$ is the supertype of all types; ML also considers subtyping of record types. In Java [GJS96], subtyping is called *extension of interfaces*, while in the more mainstream language C++ [Str91] it is hidden within the notion of class inheritance. One of the few language definitions that explicitly speak of subtyping is Modula-3 [CDG$^+$89].

It is apparent from the list of programming languages above that subtyping is an essential feature of typed object-oriented languages. While object-orientedness is an elusive quality* [AC96, Pit93], one may reasonably argue that it involves two essential ingredients: values carrying enough runtime information to perform dispatch (for example, runtime type tags, or references to "method traits"), and the ability to uniformly manipulate heterogeneous collections of data. In order to achieve the latter in statically typed languages, subtyping, while not absolutely required, is very convenient.

Designers of dynamically-typed languages are not concerned with typing problems related with uniform access to heterogeneous collections of data. This observation goes a long way towards explaining why the dynamics of dynamically typed object-oriented programming systems are often more expressive than those of statically typed ones. Of particular interest from the point of view of dynamics are the object-oriented programming systems of Lisp family languages, notably Flavors [WM81], CLOS [BDG$^+$89], and their respective "meta-object protocols" [KdRB91]. Dynamics of object-oriented languages, while an interesting area, are outside the scope of this thesis, and will not be considered further.

*Object-oriented: a characteristic of a graphical user-interface. It means that you command the computer by working with on-screen objects rather than issuing written commands. *Consumer Reports*, September 1993.

1.3 Untyped Game Semantics and subtyping

Game models usually start by defining a category of *games*, used for interpreting types, and then interpret terms as *strategies* over these games (arrows in the category). This approach is convenient and elegant, as a number of routine manipulations of strategies and games can be neatly packaged within the category-theoretical structure.

Unfortunately, just like the notions of element and subset, the notions of subsetting and subtyping do not have an immediate categorical formulation. Indeed, in the usual categorical models, these notions are not invariant by isomorphism, and are therefore finer than what can directly be described categorically. This observation should of course not be taken to imply that they cannot be treated in a category theoretic framework, only that in our view the advantages of category theory would be offset by the technical difficulties relating to finding the right categorical framework.

We therefore choose an approach analogous to the familiar inclusive subsets model (described in Section 2.1.1). We start with a universal space of strategies, which will be used to interpret an untyped calculus. We then define a notion of game over our space of strategies; games will form a partially ordered set in which we will interpret types. Unlike the inclusive subsets model, we give a direct definition for games and the ordering on them; in effect, we give a definition of subtyping that does not refer to terms.

For the untyped calculus, we define a model (Chapter 4) that explicitly accounts for erroneous situations. Games, used for interpreting types, can be seen as being safety (as opposed to liveness) specifications over strategies, with subtyping corresponding to implication of specifications. These specifications are shown to be closed under arbitrary disjunction and conjunction, which leads to a potentially very rich type system. Interpreting recursive types requires a supplementary set of tools, using methods drawn from metric topology.

This approach has its limits. It is immediately apparent that an approach intrinsically based on denotations of terms cannot model type systems that make finer distinctions, such as complexity of evaluation [Hof99].

Perhaps a more disturbing limitation is that we only consider safety properties, and don't take liveness into account. This leads to a failure to precisely capture the uniformity of quantifiers (Section 5.2.2), and prevents us from expressing in the model the strong normalisation of some typed calculi such as the simply typed λ-calculus (Section 2.2.3.1), or System F [GTL89] and its variants [Com95].

1.4 Structure of this thesis

In Chapter 2, we present a very elementary introduction to λ-calculi, show how to describe their semantics, how to use types in them, and a number of common features of more advanced type systems for λ-calculi. In Chapter 3, we start by introducing an untyped calculus with explicit accounting for runtime errors; these errors will provide a criterion for soundness of a type system, and will be directly reflected in the semantics. We then introduce a typing system with subtyping for this calculus, and attempt to convince the reader that this type system is, indeed, interesting enough to serve as an example of our method. It is important to stress at this point that problems related to the decidability of typechecking or type inference in this system are outside the scope of this thesis.

In Chapter 4, we introduce a set of *strategies*, and carry out a number of constructions over it. We then introduce an ordering on strategies — the *liveness ordering*, which is inspired by Abramsky's *back-and-forth inclusion relation* [Abr97, Prop. 5.2, 5.3] — and show how to interpret the untyped calculus in that ordered set. The interpretation is shown to be *inequationally sound*, in that the liveness ordering is finer than the observational ordering up to the interpretation.

In Chapter 5, we introduce a set of *games*, and show that it has some very good properties, most notably those of being a complete lattice and a complete ultrametric space. We then show a number of properties that relate the two structures, and assume an additional property without proof (Conjecture 101).

In Chapter 6, we show how to interpret types in this set, and, again, prove the soundness of the interpretation, both with respect to typing and subtyping. Finally, Chapter 7 concludes the thesis and outlines a few potential directions for further research.

Chapter 2

Background

This section provides the reader with a modest amount of background information. In Section 2.1, we introduce subtyping informally, and sketch common ways of modelling systems with subtyping. Then, in Section 2.2, we give a more formal introduction to λ-calculi, first in an untyped framework, then in a typed one. We then proceed to introducing richer type systems, giving in passing a more detailed description of subtyping.

2.1 Subtyping

A familiar notion from set theory is that of *subset*. The set N of natural (non-negative) integers, for example, is a subset of the set Z of relative integers,

$$N \subseteq Z.$$

This means that any element of N, say

$$5 \in N,$$

is *eo ipso* an element of Z:

$$5 \in Z.$$

The analogous notion in programming languages and calculi is called *subtyping*: a type A is a subtype of B (written $A \leq B$) when any term of type A is also of type B. Subtyping is an important feature of numerous programming languages, and can be considered as one of the essential ingredients of this elusive quality, object-orientedness. A more formal introduction to subtyping within λ-calculi will be given in Section 2.2.4.1.

For ground types, subtyping behaves just like subsetting, and a type $\mathbf{N}$ of natural integers is a subtype of a type of relative integers $\mathbf{Z}$, written

$$\mathbf{N} \leq \mathbf{Z}.$$

In other words, subtyping of ground types is inclusion of the carriers.

Consider, however, what happens at higher order. A natural idea would be to define the carrier of an arrow type as the union of all the graphs of maps belonging to this type, and expect subtyping to coincide with inclusion of carriers. For example, the carrier of $\mathbf{N} \to \mathbf{N}$ would be $N \times N$, that of $\mathbf{Z} \to \mathbf{N}$ be $Z \times N$, and we would therefore expect to have $\mathbf{N} \to \mathbf{N} \leq \mathbf{Z} \to \mathbf{N}$.

Let, however, f be the function that maps an integer n to $n + 1$. Obviously, this function is of type $\mathbf{Z} \to \mathbf{Z}$, as it maps arbitrary integers to integers, as well as $\mathbf{N} \to \mathbf{Z}$ (it maps natural integers to integers) and $\mathbf{N} \to \mathbf{N}$ (it maps natural integers to natural integers). But $f(-5) = -4$, so f does not, in general, map relative integers to natural integers, and cannot therefore be of type $\mathbf{Z} \to \mathbf{N}$.

On the other hand, it is immediate that any function of type $\mathbf{N} \to \mathbf{N}$ is *a fortiori* of type $\mathbf{N} \to \mathbf{Z}$, as any result of type $\mathbf{N}$ is also of type $\mathbf{Z}$. It is also true that any function of type $\mathbf{Z} \to \mathbf{N}$ is of type $\mathbf{N} \to \mathbf{N}$, as we may safely forget about the larger domain $\mathbf{Z}$ of the function and restrict it to natural integers. This leads us to the typing rule

$$\begin{array}{l} \text{if } A' \leq A \text{ and } B \leq B' \\ \quad \text{then } A \to B \leq A' \to B' \end{array}$$

The important point is the inversion of subtyping on the left-hand side of the arrow, which contradicts our earlier intuition, and shows that subtyping cannot simply be modelled as inclusion of carriers.

2.1.1 Models of subtyping

Types as ideals The simplest and most intuitive model for subtyping is the so-called *types as ideals*, or *inclusive subsets* model [Car88]. In this model, we start with an untyped model of the underlying calculus, typically a universal domain D (so that $D \to D \subseteq D$) and an interpretation $[\![\cdot]\!]$ mapping terms to elements of D. A ground type $\mathbf{N}$ is modelled as the set of denotations of all terms of type $\mathbf{N}$:

$$[\![\mathbf{N}]\!] = \{[\![M]\!] \mid \ M \text{ has type } \mathbf{N}\},$$

while an arrow type $A \to B$ is modelled as (roughly) the set of denotations of terms that map terms of type A to terms of type B:

$$[\![A \to B]\!] = \{d \in D \mid \forall e \in D, \text{ if } e \in [\![A]\!], \text{ then } d(e) \in [\![B]\!])\}.$$

Typing can then be interpreted as $\in$,

$$M \text{ is of type } A \text{ implies } [\![M]\!] \in [\![A]\!]$$

and therefore subtyping can be interpreted as inclusion:

$$A \leq B \text{ implies } [\![A]\!] \subseteq [\![B]\!].$$

Without doubt, this model is both technically elementary and close to the intuition. However, it interprets terms as sets of (denotations of) types, and therefore remains somewhat close to the syntax. Thus, it may be argued that while it adequately formalises of the intuitions linked to subtyping, and provides information on the structure of types (types are order ideals), it doesn't give a direct, convincing explanation of typing and subtyping.

Partial Equivalence Relations A similar model of subtyping is the Partial Equivalence Relation (P.E.R.) model [BL90]. A P.E.R. R over a set D is a transitive, symmetric (but not necessarily reflexive) binary relation over D; in effect, it is an equivalence relation over a subset of D, known as the *range* of R. As above, terms are interpreted in a universal domain D. A ground type $\mathbf{N}$ is interpreted as the discrete P.E.R. the range of which is the carrier of $\mathbf{N}$,

$$[\![\mathbf{N}]\!] = \{([\![M]\!], [\![M]\!]) \mid M \text{ is of type } \mathbf{N}\},$$

while an arrow type $A \to B$ is interpreted as the set of couples of denotations of terms that map A-related terms to B-related terms:

$$\begin{aligned}&[\![A \to B]\!] = \\ &\quad \{(d, d') \in D \times D \mid \forall (e, e') \in D \times D, \text{ if } (e, e') \in [\![A]\!], \text{ then } (d(e), d'(e')) \in [\![B]\!]\}.\end{aligned}$$

As with the inclusive sets model, typing is interpreted by $\in$, and subtyping by inclusion.

The additional structure of P.E.R.s allows for a more faithful modelling of the uniformity of quantifiers than with the simple inclusive sets model. However, just like the ideals above, P.E.R.s are simply sets of denotations of terms; thus, the P.E.R. model, just like than the inclusive subsets model, does not provide a purely semantic explanation of subtyping.

2.2 λ-calculi

Consider the expression

$$x - y.$$

This can represent a function of x and y, a value, where x and y are fixed values that have been defined earlier, a function of x, where y is fixed, or a function of

y. Mathematicians usually distinguish between those by naming functions:

$$f(x) = x - y; \qquad g(y) = x - y.$$

This is often sufficient as long as there is a clear distinction between functions and values, but becomes inconvenient and leads to heavy notation as soon as *higher-order functions* — functions operating on functions — are present.

Higher-order functions have many practical applications. They were found to be a powerful way of increasing the expressivity of a programming language, and, indeed, many programming languages, starting with Algol 60 [Nau63] and Lisp 1.5 [McC62], include higher-order functions in some (often restricted) form. They have also been found to be helpful for describing semantics of programming languages, and are heavily used in Denotational Semantics [Sto77] (for an elaborate example of a denotational description of a programming language, see Appendix A of the Scheme report [RC92]). An elementary introduction to higher-order functions is provided by Gunter [Gun92, Section 7.1].

The λ-notation for functions was introduced by Alonzo Church [Chu41] and is a solution to the problem of writing higher-order functions. Church introduced a specific symbol λ for binding variables, which allows for the description of anonymous functions (functions that have not been given a name). With the help of the λ-notation, the functions f and g above can be written as

$$f = \lambda x.x - y; \qquad g = \lambda y.x - y.$$

Manipulation of higher-order functions is then as natural as manipulation of ground values; for example, the following (anonymous) third-order function maps a second order function to its value at the identity:

$$\lambda x.(x\ \lambda y.y).$$

Church originally presented *pure* λ-calculi, in which every term is a function. As pure calculi lead to some semantic difficulties which are outside the scope of this thesis, we will only consider calculi with ground values, such as integers, booleans *etc.*

2.2.1 Untyped λ-calculi

The syntax of terms in the untyped λ-calculus we will consider consists of a countable set of variables $x, y, z, \ldots$, of functions $\lambda x.M$ for any term M, and of function applications $(M\ N)$ for terms M and N. Furthermore, we will suppose the existence of two ground values $\mathbf{tt}$ and $\mathbf{ff}$; the actual number of ground values does not make any difference for our purposes. The syntax of terms can be described by the BNF grammar

$$\begin{aligned} M, N ::= x \mid \lambda x.M \mid (M\ N) \\ \mid \mathbf{tt} \mid \mathbf{ff} \end{aligned}$$

Note that variables play two different rôles in this calculus. In the term $\lambda x.x$, the occurrence of the variable x is *bound* by the binder λx; in particular, the name of such a variable has no importance whatsoever, and the term $\lambda x.x$ can be taken as equivalent to the term $\lambda y.y$. On the other hand, in the term $\lambda x.y$, the variable y occurs *free*, and can be considered as an arbitrary constant.

More formally, the set $\mathrm{FV}(M)$ of free variables of a term M is defined by induction on the syntax of M as follows:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\}; \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) \backslash \{x\}; \\ \mathrm{FV}((M\ N)) &= \mathrm{FV}(M) \cup \mathrm{FV}(N); \\ \mathrm{FV}(\mathbf{tt}) &= \emptyset; \\ \mathrm{FV}(\mathbf{ff}) &= \emptyset. \end{aligned}$$

The idea here is that free variable occurrences are allowed to interact with the environment. On the other hand, bound variables are the private business of the term, and their names are of no interest to the environment; this is formalised by the notion of *α-equivalence*: two terms are said to be α-equivalent if they only differ in the names of their bound variables. More exactly, α-equivalence is the least congruence on terms relating $\lambda x.M$ and $\lambda y.(M[x\backslash y])$, for any term M and variables x and y such that $y \notin \mathrm{FV}(M)$. Two α-equivalent terms are equivalent for all purposes, and from now on we will implicitly take λ-terms up to α-equivalence.

In order to allow the environment to act on the free variables of a term, we need the notion of *substitution*. Given a term M and a variable x (possibly free in M), the substitution of x in M by a term P, written $M[x\backslash N]$, is inductively defined on the syntax of M as follows:

$$
\begin{aligned}
x[x\backslash P] &= P \\
(\lambda x.M)[x\backslash P] &= \lambda x.M \\
(\lambda y.M)[x\backslash P] &= \lambda y.(M[x\backslash P]) \qquad \text{if } x \neq y,\ y \notin \mathrm{FV}(P) \\
(M\ N)[x\backslash P] &= (M[x\backslash P]\ N[x\backslash P]) \\
\mathbf{tt}[x\backslash P] &= \mathbf{tt} \\
\mathbf{ff}[x\backslash P] &= \mathbf{ff}
\end{aligned}
$$

Note how the third rule may, due to the side condition $y \notin \mathrm{FV}(P)$, require that a bound variable be α-renamed.

2.2.2 Operational semantics, contextual equivalence

Before we can start thinking about modelling programming languages in enlightening ways, we need to define a reference semantics for it. This reference semantics will be *operational*, in that it directly defines how computation proceeds rather than attempting to define the meaning of terms with respect to some mathematical structure.

The simplest way of defining this operational semantics is by defining a *one step reduction* relation $\rightsquigarrow$ on terms; this style of presentation is known as the *small-step* style. For the λ-calculus with the syntax above, a common choice — the *call-by-name* semantics — consists of the β-reduction rule

$$(\lambda x.M\ N) \rightsquigarrow M[x\backslash N]$$

as well as the structural rule

$$\frac{M \rightsquigarrow M'}{(M\ N) \rightsquigarrow (M'\ N)}$$

A term M is said to be in normal form when it can no longer be reduced, *i.e.* when there exists no term M' such that $M \rightsquigarrow M'$.

Another popular presentation style, the *big-step* style, consists in inductively defining a relation $\Downarrow$ that relates a term to its normal form (if any). For the call-by-name λ-calculus, we could write

$$\frac{M \Downarrow \lambda x.M' \quad M'[x\backslash N] \Downarrow P}{(M\ N) \Downarrow P}$$

Having defined the semantics, we are ready to define a notion of *observational equivalence*. Intuitively, two terms are observationally equivalent if there is no way to observe a difference between the two. In order to formalise this intuition, we first need to choose a notion of observation; in our case, we will choose *termination at ground type*; namely, the set of observations will consist of only one observation, the one defined by the set of terms M such that $M \Downarrow \mathbf{tt}$. We will then say that two terms M and N are *observationally equivalent*, written $M \cong N$, if and only if they yield the same observations in any context; precisely,

$$\forall C[\cdot]\, C[M] \Downarrow \mathbf{tt} \text{ iff } C[N] \Downarrow \mathbf{tt}.$$

Furthermore, a term N is *observationally superior* to a term M, written $M \lesssim N$, if N yields an observation whenever M does:

$$\forall C[\cdot]\, C[M] \Downarrow \mathbf{tt} \text{ implies } C[N] \Downarrow \mathbf{tt}.$$

Of course, this defines a preorder, the associated equivalence being observational equivalence:

$$M \cong N \text{ iff } M \lesssim N \text{ and } N \lesssim M.$$

2.2.3 Types in λ-calculi

The calculi in the preceding section were untyped in that no language-level feature distinguished between, say, functions and ground values. Untyped systems are useful and powerful*, but such a distinction is often useful, both as a conceptual tool, and in order to prevent the writing of "meaningless" terms.

2.2.3.1 Typed λ-calculi: the Church presentation of types

When Church originally considered the issue of typing, he introduced what is now known as the *simply-typed λ-calculus*. The main new feature consists in defining a set of types ranged over by A, and decorating every λ-binder with the intended type of the bound variable; the syntax of λ-abstractions now becomes:

$$M ::= \lambda x : A.M$$

Note, however, that unlike in the untyped calculus, not all terms in the syntax are well-formed, and defining the class of *well-typed* terms requires some more machinery.

First of all, of course, we need to define a set of types. Types are defined by the syntax

$$A, B ::= \mathbf{Bool} \mid A \to B$$

*As the success of Zermelo-Fraenkel untyped set theory clearly shows.

where **Bool** is the type of the ground value **tt**, while $A \to B$ is the type of functions from A to B.

Terms are not in general typed *in vacuo*, as information needs to be made available about free variables. A *type environment* is a partial map from variables to types with finite domain; we write $\emptyset$ for the everywhere undefined environment, and

$$E, x : A$$

for the environment that is equal to E except that it maps x to A. We write $\mathrm{dom}(E)$ for the domain of an environment (this is a finite set of variables).

A *judgement* is a triple (E, M, A), usually written as

$$E \vdash M : A$$

where E is an environment, M a term and A a type. Such a judgement is intended to mean that under the assumptions contained in E, M has type A.

Rather than defining directly the set of well-typed terms, we define the set of correct type judgements. Such judgements are the ones that can be generated using the following rules:

$$E \vdash \mathbf{tt} : \mathbf{Bool} \qquad E \vdash \mathbf{ff} : \mathbf{Bool}$$

$$E, x : A \vdash x : A$$

$$\frac{E, x : A \vdash M : B}{E \vdash \lambda x : A.M : A \to B}$$

$$\frac{E \vdash M : A \to B \quad E \vdash N : A}{E \vdash (M\ N) : B}$$

The first two rules simply state that the constants **tt** and **ff** are of type **Bool** under any environment. The third one says that any assumption can be extracted from an environment. The last one is the expected rule for function application.

The fourth rule, dealing with introduction of function abstraction, is somewhat more complicated. The body of the abstraction, M, will typically have a free occurrence of the variable x. The rule says that if M can have type B in an environment in which x is assumed to have type A, then the function $\lambda x : A.M$ can have type $A \to B$. Note how the assumption $x : A$ is discarded, and how the type A is explicitly introduced in the function.

2.2.3.2 Type Assignment: the Curry presentation of types

The typed calculus, outlined above, makes types an intrinsic feature of the syntax of terms. In particular, there is no canonical typed identity term, but identities

$I_A = \lambda x : A.x$ over any type A. While this is natural from a set-theoretic standpoint, an operational understanding of the calculus leads one to conclude that all these terms do the same thing — namely return their argument unchanged.

An alternative approach — known as Type Assignment — consists in keeping the syntax of terms unchanged and defining a relation between terms and types. The definition of types and type environments remains unchanged, but a type judgement is now a triple

$$E \vdash M : A$$

where E is still an environment, A a type, but M is an *untyped* term. Term formation follows the following rules:

$$E \vdash \mathbf{tt} : \mathbf{Bool} \qquad E \vdash \mathbf{ff} : \mathbf{Bool}$$

$$E, x : A \vdash x : A$$

$$\frac{E, x : A \vdash M : B}{E \vdash \lambda x.M : A \to B}$$

$$\frac{E \vdash M : A \to B \quad E \vdash N : A}{E \vdash (M\ N) : B}$$

the only change being in the fourth rule, which does not explicitly introduce the type A in the term.

This seemingly small difference has some important consequences. First and foremost, a given term does not have a unique type in a given environment; for example

$$\vdash \lambda x.x : A \to A$$

is true for any type A.

More practically, while typing algorithms for typed calculi tend to be trivial, they are often difficult in Type Assignment; introduction of new features into Type Assignment systems often leads to semi-decidability of typing. However, problems of type checking and type reconstruction are outside the scope of this thesis, and we will use Type Assignment systems throughout.

2.2.4 Common extensions to the language of types

The simply-typed calculi presented above are very elementary. In practice, however, simply-typed calculi are not sufficient. In particular, they often fail to be expressive enough (either making it impossible to type the function that the programmer has in mind, or giving it a type that makes it less general than intended), or force the use of an indirect encoding for features that seem intuitively primitive. This section shows some extensions to λ-calculi that help solve these problems.

2.2.4.1 Subtyping and subsumption

In the systems presented above, there is no relationship between types. However, as explained informally in Section 1.2, it is quite common to have types A and B such that, whenever it is true that

$$E \vdash M : A$$

then

$$E \vdash M : B$$

For example, consider a programming language with a type of natural integers $\mathbf{N}$, and a type of relative integers $\mathbf{Z}$. As any natural integer is *a fortiori* a relative integer, we would expect that any term that types as $\mathbf{N}$ also type as $\mathbf{Z}$.

One common way of treating such situations is to introduce a binary relation on types $\leq$ known as *subtyping*. A new sort of judgement is introduced, the *subtyping judgement*, which has the form

$$E \vdash A \leq B$$

where E is a typing environment, and A and B are types. If such a subtyping judgement is verified, A is said to be a *subtype* of B; B is then said to be a *supertype* of B. A new typing rule is added, known as *subsumption*, that states that types can be replaced by their supertypes:

$$\frac{E \vdash M : A \quad E \vdash A \leq B}{E \vdash M : B}$$

In the presence of subtyping, it is natural to have a type of all terms. This type is known as *top*, written $\top$; the syntax of types therefore becomes:

$$A, B ::= \mathbf{Bool} \mid A \to B \mid \top$$

with the subtyping rule

$$E \vdash A \leq \top.$$

Many programming languages contain a feature known as *coercion*. In such languages, an expression such as

$$2 + 3.5$$

is transparently interpreted as though it were

$$\mathbf{Float}(2) + 3.5$$

where **Float** is some mapping between the type of 2 and the type of 3.5. While subsumption can be interpreted as a special case of coercion [CG92], it is our view that subsumption and coercion correspond to subtly different programming-language features. Coercion corresponds to any mapping that happens with no explicit programmer intervention, while subsumption should only be used to model those mappings that do not involve changes of representation. Thus, we believe that subsumption is more adapted to modelling most current object-oriented languages.

An interesting feature of subtyping is its treatment of function types. The rule for proving subtyping of arrow types is

$$\frac{E \vdash A' \leq A \quad E \vdash B \leq B'}{E \vdash A \to B \leq A' \to B'}$$

which says that the arrow constructor is covariant in its right-hand-side argument, but contravariant in its left-hand-side argument. This latter condition is fairly natural at first order, if one considers that a function

$$f : A \to B$$

may have its range restricted, so as to become

$$f : A' \to B$$

where A' is a subset of A.

2.2.4.2 Products and record calculi

While not absolutely necessary, the ability to manipulate ordered heterogeneous collections of data is often convenient. The simplest way to enable this is to providing *ordered pairs* and *product types*. A more complex, but often more convenient, way of achieving the same result is to provide *records* and *record types*. As we shall see, the two features turn out to be equivalent.

Product types Given terms M and N, the pair of M and N is written

$$(M, N)$$

The most important operation on pairs is *projection*. Two new terms, the left projection π_l and the right projection π_r are introduced, with reduction rules

$$\pi_l(M, N) \rightsquigarrow M$$
$$\pi_r(M, N) \rightsquigarrow N$$

or, equivalently, in big-step style,

$$\frac{M \Downarrow (N, P) \qquad N \Downarrow V}{\pi_l(M) \Downarrow V}$$

$$\frac{M \Downarrow (N, P) \qquad P \Downarrow V}{\pi_r(M) \Downarrow V}$$

In order to type pairs, a new type constructor, the *product* constructor $\times$, is introduced. The type language now becomes

$$A, B ::= \mathbf{Bool} \mid A \to B \mid \top \mid A \times B$$

Just like the identity, the terms π_l and π_r do not have unique types. Instead, given arbitrary types A and B, they can be typed as

$$\vdash \pi_l : A \times B \to A$$

$$\vdash \pi_r : A \times B \to B$$

Product types are introduced by the following rule:

$$\frac{E \vdash M : A \quad E \vdash N : B}{E \vdash (M, N) : A \times B}$$

and subtyped covariantly:

$$\frac{E \vdash A \leq A' \quad E \vdash B \leq B'}{E \vdash A \times B \leq A' \times B'}$$

Record types While products are a convenient low-level construct, manipulating large collections of data using products leads to very heavy and error-prone notations. Most programming languages allow the programmer to manipulate collections of named data using *records*.

We suppose given a countable set L of *labels*. For any finite family of labels $(l_i)_{i=1\cdots n}$ and family of terms $(M_i)_{i=1\cdots n}$, the record constituted of the *fields* M_i labelled by the l_i is written

$$\langle l_i = M_i \rangle^{i=1\cdots n}$$

For example, the record consisting of the fields l with value $\mathbf{tt}$ and f with value $\lambda x.x$ will be written

$$\langle l = \mathbf{tt}, f = \lambda x.x \rangle$$

The operation on records that corresponds to projection is called *field selection*. A field with label l is selected from a record M using the notation $M.l$. The semantics of this operation are defined by

$$\langle l_i = M_i \rangle^{i=1\cdots n}.l_j \rightsquigarrow M_j \qquad \text{where } j \in 1 \cdots n$$

It is also possible to (functionally) update a field in a record. The operation of field update is written $M.l := B$, where l is a label and B a term. The semantics of this operation is defined by

$$\langle M_i \rangle^{i=1\cdots n}.l_j := M'_j \rightsquigarrow \langle l_i = N_i, l_j = N \rangle^{i=1\cdots n, i \neq j} \qquad \text{where } j \in 1 \cdots n$$

Records are typed by *record types*. Given a finite family of labels $(l_i)_{i=1\cdots n}$ and family of types $(B_i)_{i=1\cdots n}$, the record type consisting of the B_i labelled by the l_i is written

$$\langle l_i : B_i \rangle^{i=1\cdots n}.$$

A record $\langle l_i = M_i \rangle^{i=1\cdots n}$ has type $\langle l_i : B_i \rangle^{i=1\cdots n}$ as soon as every M_i has the respective type B_i; formally,

$$\frac{E \vdash M_i : B_i \text{ for all } i = 1 \cdots n}{E \vdash \langle l_i = M_i \rangle^{i=1\cdots n} : \langle B_i : M_i \rangle^{i=1\cdots n}}$$

Intuitively, a record $\langle l_i = M_i \rangle^{i=1\cdots n}$ of type $\langle l_i : B_i \rangle^{i=1\cdots n}$ can be subsumed both by subsuming its fields and by discarding some of its fields (*i.e.* forgetting about their existence). Whence the subtyping rule

$$\frac{E \vdash B_j \leq B'_j \text{ for all } j = 1 \cdots n'}{E \vdash \langle l_i : B_i \rangle^{i=1\cdots n} \leq \langle l_j : B_j \rangle^{i=1\cdots n'}} \; n \leq n'$$

Products or records: who cares? Obviously, products can be represented by records. Let l and r be two distinct labels. For the product type $A \times B$, write $\langle l : A, r : B \rangle$. For the pair (M, N), write $\langle l = M, r = N \rangle$. The projections can then be written as

$$\pi_l = \lambda x.(x.l)$$
$$\pi_r = \lambda x.(x.r)$$

The converse is a little more difficult. Let label be an enumeration of labels, *i.e.* a one-to-one function from natural integers to labels, and let **top** be a (closed) term of type $\top$. We will represent the record type

$$\langle l_i : B_i \rangle^{i=1\cdots n}$$

by the product type

$$C_0 \times C_1 \times C_2 \times \cdots C_k \times \top$$

where $k = \max\{j \mid l_i = \text{label}(j) \text{ for some } i \in 1 \cdots n\}$ and $C_j = B_i$ if $l_i = \text{label}(j)$ for some $j \in 1 \cdots n$, $C_j = \top$ otherwise.

The record $\langle l_i = M_i \rangle^{i=1\cdots n}$ is represented by the pair

$$(M_0, (M_1, \cdots (M_k, \mathbf{top})))$$

where k is defined as above, and for any $j \in 1 \cdots k$, $M_j = b_i$ if $l_i = \text{label}(j)$, $M_j = \mathbf{top}$ otherwise, where $\mathbf{top}$ is some term of type $\top$. The operation $M.l$ is represented by the application $\pi_l(\pi_r \cdots \pi_r(M))$, where the number of occurrences of π_r is equal to the unique integer j such that $l = \text{label}(j)$.

2.2.4.3 Second order calculi

Universal quantification As we saw before, the identity term $I = \lambda x.x$ does not have a single type; instead, for any type A it has type $A \to A$. In order to avoid having to manipulate the various types of the identity, and to increase opportunities for modularity, it is useful to introduce a way of *abstracting* the type A away; this is done by introducing the *bounded universal quantifier* $\forall$ into our type language, as well as a notion of type variable. Assuming a countable set of type variables X, the syntax of types now becomes:

$$A, B ::= \mathbf{Bool} \mid A \to B \mid \top \mid X \mid \forall X \leq A.B[X]$$

where the notation $B[X]$ is only used to stress the fact that X may appear free in B.

A term is of type $\forall X \leq A.B[X]$ if, up to some nonessential changes, it may be used as a term of any type $B[C]$ for any type C as long as $C \leq A$. Traditionally, this is expressed by introducing an explicit $\forall$-elimination rule, for example, in our Curry-style presentation,

$$\frac{E \vdash M : \forall X \leq A.B[X]}{E \vdash M : B[C]} \text{ for any } C \text{ such that } E \vdash C \leq A$$

However, a more abstract way of expressing the same thing is to say that $\forall X \leq A.B[X]$ is a subtype of $B[C]$ for any $C \leq A$. This statement gives just the right information to allow us to apply subsumption in exactly the same circumstances where the rule above can be applied, and this thesis will use the subtyping presentation of $\forall$-elimination throughout.

What about introduction of $\forall$? It seems that it cannot be handled without the use of a specific rule. A term M can be assigned the type $\forall X \leq A.B[X]$ if, for any type $C \leq A$, we are able to prove that $M : B[C]$; of course, we need to be able to do this without any other information on the type C. Formally,

$$\frac{E, X \leq A \vdash M : B[X]}{E \vdash M : \forall X \leq A.B[X]} \; X \text{ does not appear free in } E$$

where the side condition "X does not appear in E" corresponds to the fact that we don't use any additional hypotheses about X.

Existential quantification Dual to the universal quantifier $\forall$ is the *existential quantifier* $\exists$. While $\forall$ is used for writing a type that may be used as any of a collection of types, $\exists$ specifies a type that corresponds to an unspecified member of a collection of types. The existential quantifier is useful in combination with record (or product) types for implementing (partially) abstract datatypes [MP88].

Suppose, for example, that we have a ground type of reals $\mathbf{R}$, and that we would want an abstract type of complex numbers. While it would, of course, be possible to use a pair of reals, say $\langle \mathrm{re} = 1, \mathrm{im} = 2\rangle$ for the complex number $1+2i$, this would enforce a particular representation for complex numbers, and would not allow switching to, say, a polar representation. Alternatively, we might use a triple of an unknown implementation value and two methods for extracting the real and imaginary parts. The number $a = 1 + 2i$ could then be written as

$$a_1 = \langle \mathrm{impl} = \langle \mathrm{re} = 1, \mathrm{im} = 2\rangle, \mathrm{re} = \lambda x.(x.\mathrm{re}), \mathrm{im} = \lambda x.(x.\mathrm{im})\rangle$$

or, using a polar representation internally,

$$\begin{aligned} a_2 = \langle \mathrm{impl} = \left\langle \mathrm{r} = \sqrt{5}, \mathrm{theta} = \frac{\pi}{3} \right\rangle, \\ \mathrm{re} = \lambda x.(x.\mathrm{r}) \times \cos(x.\mathrm{theta}), \\ \mathrm{im} = \lambda x.(x.\mathrm{r}) \times \sin(x.\mathrm{theta})\rangle \end{aligned}$$

In both cases, the application ($a.\mathrm{re}\ a.\mathrm{impl}$) would yield the real part of a, namely 1.

Of course, we want to assign to both values the same type. In order to do so, we need to write a type that does not explicitly contain the type of a.impl; the type of the impl field is said to be *hidden*. This can be done by typing complex numbers as

$$\mathbf{C} = \exists X \leq \top.\, \langle \mathrm{impl} : X, \mathrm{re} : X \to \mathbf{R}, \mathrm{im} : X \to \mathbf{R}\rangle$$

Dually to what we do with universal types, we introduce the $\exists$ quantifier by using subtyping:

$$\frac{E \vdash C \leq A}{E \vdash B[C] \leq \exists X \leq A.B[X]}$$

Both quantifiers, as well as the typing rules pertaining to them, will be explored in more detail in Section 3.2.2.

2.2.4.4 Recursive types

The typing system that we have presented until now does not allow for self-application of terms: for example, it does not assign a non-trivial type to the term $\lambda x.xx$. Indeed, in order to give a non-trivial type to xx, x needs to be simultaneously assigned a type T and the type $T \to T$; therefore, the type T needs to satisfy the recursive equation

$$T = T \to T.$$

Counterintuitive as it may appear from a set-theoretic standpoint, self-application has many important uses in programming languages. For example, in order to use an object-oriented style in traditional structured programming languages, an object o with a method m is often represented as a record with a field m; the value of m would be a function taking o as an argument (this is known as the *self-application interpretation* of objects).

In order to solve such recursive equations on types, we need *recursive* types. Technically, a new type constructor μ, known as the *fixpoint constructor*, is introduced, leading to the type language

$$\begin{aligned} A, B ::= &\ \mathbf{Bool} \mid A \to B \mid \top \mid X \\ \mid &\quad \forall X \leq A.B[X] \mid \exists X \leq A.B[X] \\ \mid &\quad \mu X.B[X] \end{aligned}$$

Intuitively, a recursive type should satisfy the isomorphism

$$\mu X.B[X] \equiv B[\mu X.B[X]]$$

However, this is not always the case, and, in fact, it is enough to have a retraction

$$\begin{aligned} \text{fold} : B[\mu X.B[X]] \to \mu X.B[X] \\ \text{unfold} : \mu X.B[X] \to B[\mu X.B[X]] \end{aligned}$$

where unfold is a left inverse, but not necessarily a right inverse of fold. In this thesis, however, we will consider recursive types as being not only isomorphic to their unfolding, but actually *equal*; in other words, we will solve recursive equations exactly.

Recursive types and subtyping interact in interesting ways [AC96, Car97, AC93]. Let $(X, \leq)$ be a partially ordered set, and $f : X \to X$ and $g : X \to X$ two maps in this space (not necessarily monotone) such that for any $x \in X$,

$f(x) \leq g(x)$; then it is not in general true that for all fixpoints a of f and b of g (or, for that matter, for any such fixpoints), $f(a) \leq g(b)$.

Accordingly, subtyping recursive types covariantly would not, in general, be sound. Let

$$A[X] = \langle x : \mathbf{Bool}, f : X \rightarrow \mathbf{Bool} \rangle,$$

and

$$B[X] = \langle x : \mathbf{Bool}, y : \mathbf{Bool}, f : X \rightarrow \mathbf{Bool} \rangle.$$

Obviously, $B[X] \leq A[X]$; however, the subtyping relation $\mu X.B[X] \leq \mu X.A[X]$ would not be sound. Indeed, let

$$a = \langle x = \mathbf{tt}, f = \lambda z.(z.x) \rangle : \mu X.A[X],$$

$$b = \langle x = \mathbf{tt}, y = \mathbf{tt}, f = \lambda z.(z.y) \rangle : \mu X.B[X],$$

and

$$g = \lambda u.((u.f)\ a).$$

Then g is of type $\mu X.A[X] \rightarrow \mathbf{Bool}$, but the application $(g\ b)$ is unsafe.

As shown by Amadio and Cardelli [AC93], there is one case in which recursive types can be subtyped covariantly. The sound rule, however, cannot be explained intuitively without reference to complete metric spaces and contractive maps, notions which we will not be considering until Section 5.4; it is therefore offered to the reader with no further comment or apology.

A type $A[X]$ is said to be *covariant* in X if X only appears free in $A[X]$ in positive positions; we then write $A[X^+]$. In the case of our type language, this means that, forgetting for now about the more complex types, $A[X^+]$ is defined by the grammar

$$A[X^+] ::= X \mid Y \mid B[X^-] \rightarrow C[X^+]$$

where Y represents any type variable other than X, and $A[X^-]$ — the set of terms *contravariant* in X — is defined by the grammar

$$A[X^-] ::= X \mid Y \mid B[X^+] \rightarrow C[X^-].$$

Terms $A[X^+]$ covariant in X have the property that whenever $X \leq Y$, $A[X] \leq A[Y]$. It appears that covariant subtyping is, indeed, sound for recursive types built covariantly:

$$\frac{E, X \leq \top \vdash A[X^+] \leq B[X^+]}{E \vdash \mu X.A[X^+] \leq \mu X.B[X^+]}\ A[X^+] \text{ and } B[X^+] \text{ covariant in } X.$$

2.2.5 Further extensions to λ-calculi

Many other extensions to λ-calculi are possible. The semantic rules of the untyped calculus can be changed to model different parameter passing conventions [Plo75, Abr90]. The untyped calculus can be extended to model imperative features, such as state or non-local jumps [Rey81, CCF94].

Furthermore, many more extensions to the type system can be considered. Of particular interest are features that introduce computation at the level of types, as in system F^ω, as well as in the presence of dependent types.

All these extensions are outside the scope of this thesis. A few more comments on this subject will be made in Chapter 6.

Chapter 3

A calculus with subtyping

This chapter introduces a call-by-name λ-calculus with two ground values which will serve as a basis for the rest of this thesis. The calculus contains all of the features presented in the previous chapter, as well as an explicit notion of failure of reduction.

3.1 The untyped calculus

The syntax of untyped terms is defined by the following grammar:

$$\begin{aligned} M, N, N' ::= & x \mid \lambda x.M \mid (M\ N) \\ & \mid \langle M, N \rangle \mid \pi_l(M) \mid \pi_r(M) \\ & \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M\ \mathbf{fi} \\ & \mid \mathbf{tt} \mid \mathbf{ff} \mid \mathbf{top} \end{aligned}$$

The first three constructs, namely variables, functional abstraction and function application, are familiar from the λ-calculus,. $\langle M, N \rangle$ represents a *pair* of terms. The $\pi_l(M)$ and $\pi_r(M)$ constructs represent the respective projections from products. Finally, the last four constructs correspond to the boolean values for truth and falsehood, the conditional, as well as a term **top** which will be used for runtime errors. We write Ω for the looping term $\Omega = ((\lambda x.(xx))\ (\lambda x.(xx)))$.

The set of free variables of a term M, written $\mathrm{FV}(M)$, is inductively defined as follows:

$$
\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M)\backslash\{x\} \\
\mathrm{FV}((M\ N)) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \\
\mathrm{FV}(\langle M, N\rangle) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \\
\mathrm{FV}(\pi_l(M)) = \mathrm{FV}(\pi_r(M)) &= \mathrm{FV}(M) \\
\mathrm{FV}(\mathbf{tt}) = \mathrm{FV}(\mathbf{ff}) = \mathrm{FV}(\mathbf{top}) &= \emptyset \\
\mathrm{FV}(\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ N'\ \mathbf{fi}) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \cup \mathrm{FV}(N')
\end{aligned}
$$

The operation of substitution $M[x\backslash P]$ consists in replacing all free occurrences of a variable x in a term M by a term P. It is inductively defined as follows:

$$
\begin{aligned}
x[x\backslash P] &= P \\
(\lambda x.M)[x\backslash P] &= \lambda x.M \\
(\lambda y.M)[x\backslash P] &= \lambda y.(M[x\backslash P]) \qquad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(P) \\
(M\ N)[x\backslash P] &= (M[x\backslash P]\ N[x\backslash P]) \\
\langle M, N\rangle[x\backslash P] &= \langle M[x\backslash P], N[x\backslash P]\rangle \\
\pi_l(M)[x\backslash P] &= \pi_l(M[x\backslash P]) \\
\pi_r(M)[x\backslash P] &= \pi_r(M[x\backslash P]) \\
\mathbf{tt}[x\backslash P] &= \mathbf{tt} \\
\mathbf{ff}[x\backslash P] &= \mathbf{ff} \\
\mathbf{top}[x\backslash P] &= \mathbf{top} \\
(\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ N'\ \mathbf{fi})[x\backslash P] &= \mathbf{if}\ M[x\backslash P]\ \mathbf{then}\ N[x\backslash P]\ \mathbf{else}\ N'[x\backslash P]\ \mathbf{fi}
\end{aligned}
$$

3.1.1 Operational semantics without errors

The simplest way of defining the operational semantics is by using a *one step reduction* relation $\rightsquigarrow$ on terms; this style of presentation is known as the *small-step* style. For our calculus, a common choice — the *call-by-name* semantics — consists of the β-reduction rule

$$((\lambda x.M)\ N) \rightsquigarrow M[x\backslash N]$$

as well as two symmetric rules for products

$$\pi_l(\langle M, N\rangle) \rightsquigarrow M \qquad \pi_r(\langle M, N\rangle) \rightsquigarrow N$$

and two rules for reduction of the conditional

$$\textbf{if } \textbf{tt} \textbf{ then } N \textbf{ else } N' \textbf{ fi} \rightsquigarrow N,$$
$$\textbf{if } \textbf{ff} \textbf{ then } N \textbf{ else } N' \textbf{ fi} \rightsquigarrow N'.$$

Finally, there is a structural rule that allows reduction within a term under some conditions; these conditions are defined by a set of *evaluation contexts*

$$\begin{aligned} E[\cdot] ::= & \ ([\cdot]\ N) \mid \pi_l([\cdot]) \mid \pi_r([\cdot]) \\ \mid & \quad \textbf{if } [\cdot] \textbf{ then } N \textbf{ else } N' \textbf{ fi} \end{aligned}$$

and we allow reduction to happen in evaluation contexts:

$$\frac{M \rightsquigarrow M'}{E[M] \rightsquigarrow E[M']}$$

In general, we will be interested in computations that take more than one step. The *reduction relation* $\rightsquigarrow^*$ is the transitive reflexive closure of $\rightsquigarrow$.

The notion of *reduction to a value* is defined by specifying a set of *values*, which can be seen as the possible outcomes of satisfactory computations. In our case, the set of values is defined as:

$$V ::= \textbf{top} \mid \textbf{tt} \mid \textbf{ff} \mid \lambda x.M \mid (M, N)$$

We say write $M \downarrow V$ if $M \rightsquigarrow^* V$ where V is a value. We write $M \downarrow$ when there exists a value V such that $M \downarrow V$, and $M \uparrow$ otherwise.

Another way of describing the semantics of this calculus is the *big-step presentation*. In that presentation, a relation $\Downarrow$ is defined by the rules in Fig. 3.1. The two presentations are related by the following lemma.

Lemma 1 *Given a closed term M and a closed value V, $M \downarrow V$ if and only if $M \Downarrow V$.*

Proof: by induction on the length of the derivation of $M \Downarrow V$ and that of the reduction $M \rightsquigarrow^* V$. ■

The concepts and notations above have been carefully chosen to make this equivalence work. As we shall see in the next paragraph, an analogous property would not hold if we had omitted to restrict $\downarrow$ to values.

$$\mathbf{tt} \Downarrow \mathbf{tt} \qquad\qquad \mathbf{ff} \Downarrow \mathbf{ff}$$

$$\lambda x.M \Downarrow \lambda x.M$$

$$\langle M, N\rangle \Downarrow \langle M, N\rangle$$

$$\frac{M \Downarrow \lambda x.M' \quad M'[x\backslash N] \Downarrow P}{(M\ N) \Downarrow P}$$

$$\frac{M \Downarrow \langle N, P\rangle \qquad N \Downarrow N'}{\pi_l(M) \Downarrow N'} \qquad \frac{M \Downarrow \langle N, P\rangle \qquad P \Downarrow P'}{\pi_r(P) \Downarrow P'}$$

$$\frac{M \Downarrow \mathbf{tt} \qquad N \Downarrow N'}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi} \Downarrow N'} \qquad \frac{M \Downarrow \mathbf{ff} \qquad P \Downarrow P'}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi} \Downarrow P'}$$

Figure 3.1: Big-step reduction rules without errors

3.1.2 Failure of reduction: loops and errors

Of course, the relation $\Downarrow$ is not total; a number of terms of our calculus do not reduce to values. This is expected, as we have done nothing whatsoever to prevent the formation of meaningless terms.

Let δ be the term $\lambda x.(x\ x)$. It is easily seen that the term $(\delta\ \delta)$ does not reduce to a value (a derivation of $(\delta\ \delta) \Downarrow$ cannot possibly have finite length). This failure of reduction can be seen in small-step style, in which $(\delta\ \delta)$ leads to an infinite sequence of one-step reductions:

$$(\delta\ \delta) \leadsto (\delta\ \delta) \leadsto (\delta\ \delta) \cdots$$

Another example of a term that fails to reduce is

$$M = \mathbf{if}\ \lambda x.x\ \mathbf{then\ tt\ else\ ff\ fi}$$

Indeed, there is no reduction rule that can be the last in a derivation of $M \Downarrow$. In this case, small-step semantics shows that the reduction remains "stuck" at a non-value term:

$$\text{there is no } M' \text{ such that } M \leadsto M'.$$

The big-step presentation does not allow one to distinguish between those two cases. Yet, intuitively, there is a big difference: in the case of $(\delta\ \delta)$, we witness a computation that fails to terminate due to looping, while in the the case of the conditional term, a computation fails to complete due to a term that is in error. This distinction is essential for the study of type systems, as we would expect a type system to eliminate some or all of the erroneous computations, but not necessarily looping computations.

3.1.3 Trappable vs. untrappable errors

While the distinction between error and looping appears naturally in the calculus, typical programming languages make a more fine-grained distinction which is not naturally present. Cardelli [Car97] distinguishes between "trappable" and "untrappable" errors, and we will follow Cardelli's terminology.

A trappable error is an exceptional situation which may happen during the execution of a correct program. Typically, trappable errors include arithmetic errors, such as division by zero, and exceptional situations occurring in parts of the universe that are not under direct control of the programmer, such as incorrect input given to the program or a hardware failure. Trappable errors are unavoidable, and a programmer should be able to expect to trap them reliably and gracefully recover when they occur.

Untrappable errors, on the other hand, are consequences of mistakes in the program. In concrete terms, a runtime system is not required to detect untrappable errors, and if one does occur, anything might happen, including an Operating System trap, erratic behaviour later on, a nuclear holocaust, or nothing at all. Of course, we will want to avoid untrappable errors, and this will be done by the means of a typing system. A typing system is said to be *safe* if it ensures that untrappable errors never happen.

This distinction is by no means artificial, and designers of untyped programming languages have found the need to draw it. The Common Lisp standard [ANS94], for example, uses the terminology "it is an error to..." for untrappable errors (which, in a dynamically typed language such as Common Lisp, it is the programmer's responsibility to avoid), and "an error is signalled" for trappable errors.

On the other hand, the distinction is conventional, and different languages may choose a different approach in the same situation. For example, dereferencing a null pointer is an untrappable error in the "C" programming language [ISO90] (an implementation is not required to detect the condition), while it is a trappable error in Java [GJS96] (and is not an error at all in most Lisp family languages!).

3.1.4 Errors and denotational semantics

It is not immediately obvious how to model errors in denotational semantics. Consider for example the domain of booleans presented in Fig. 3.2(a). One choice would be to add an error value **error** "on the side," as in Fig. 3.2(b); another one would be to add a value **top** as a top element, as in Fig. 3.2(c).

It is our view that errors on the side model trappable errors, while errors as top values model untrappable errors. Consider, indeed, the addition to our calculus of a term **ignore-errors** that would satisfy

$$\frac{M \Downarrow \mathbf{tt}}{\mathbf{ignore\text{-}errors}\ M \Downarrow \mathbf{tt}} \qquad \frac{M \Downarrow \mathbf{ff}}{\mathbf{ignore\text{-}errors}\ M \Downarrow \mathbf{tt}}$$

$$\frac{M \Downarrow \mathbf{top}}{\mathbf{ignore\text{-}errors}\ M \Downarrow \mathbf{ff}}$$

Denotationally, such a term would have to map **tt** to **tt** while mapping **top** to **ff**, which would be a non-monotone semantics. On the other hand, modelling an analogous term using **error** instead of **top** would cause no problem at all.

In a calculus that would include both trappable and untrappable errors, the domain of Booleans would have two distinct error values, as in Fig. 3.2(d). As trappable errors have been modelled before [CCF94], we shall restrict ourselves to a single error value, and adopt the domain of Fig. 3.2(c).

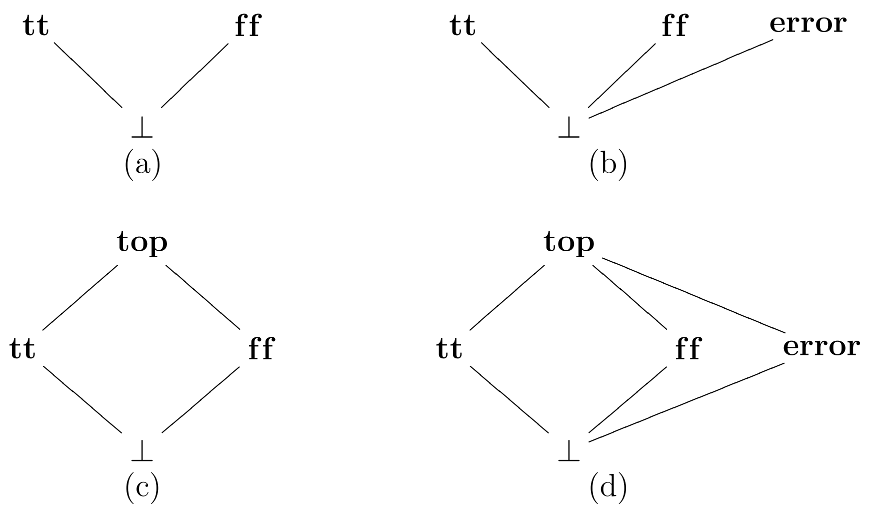


Figure 3.2: Four domains of Booleans

The addition of a top value to Scott domains was a common feature of early Denotational Semantics [Sto77]. However, this value does not seem to be used for modelling anything, but is just added to domains in order to make them into complete lattices, which, unlike Scott domains, are a structure familiar to every mathematician.

3.1.5 Operational semantics with errors

The discussion above leads us to model untrappable errors by introducing into the syntax an explicit error term **top** (this notation is used to distinguish between **top** and $\top$, which will be a type).

The simplest way to add errors to the calculus would be to add rules to cause all "stuck" terms to reduce to **top**. In the small-step presentation, this only requires the single rule

$$M \rightsquigarrow \mathbf{top} \qquad \begin{array}{l}\text{whenever } M \text{ is not a value and} \\ \quad\text{there is no } M' \neq \mathbf{top} \\ \quad\text{such that } M \rightsquigarrow M'.\end{array}$$

Unfortunately, this does not quite work [Chr00, Section 2.3] . Indeed, let $\mathbf{I_{Bool}}$ be the "identity on the Booleans,"

$$\mathbf{I_{Bool}} = \lambda x.\mathbf{if}\ x\ \mathbf{then}\ x\ \mathbf{else}\ x\ \mathbf{fi},$$

and $\mathbf{Y}$ the fixpoint combinator

$$\mathbf{Y} = \lambda x.(\lambda y.x(yy))(\lambda y.x(yy)).$$

We'd expect the fixpoint of $\mathbf{I_{Bool}}$ to behave like a Boolean, and thus to yield an error *e.g.* when applied to a value (as in the term $((Y\ \mathbf{I_{Bool}})\ \mathbf{tt})$). A simple-minded operational semantics such as the one above, however, will cause this term to loop.

We solve this problem by introducing the notion of *initial component*. An initial component is a finite sequence over the set $\{1, l, r\}$ which specifies the type of context at which reduction happens. The empty component, written ϵ, specifies reduction in a context of ground type. Given a component c, the component $1 \cdot c$ specifies reduction at a context of arrow type; the components $l \cdot c$ and $r \cdot c$ specify, respectively, reduction on the left- and right-hand sides of a product type.

You may want to think of the syntax tree of a type, where all the edges leading from an arrow node have been labelled with 0 and 1 respectively, while the edges leading from a product node have been labelled with l and r. The path from the root to a node at which a result is to be produced carries an initial component.

3.1.6 Operational semantics

As introduced in the previous paragraph, an *initial component* is a string over $\{1, l, r\}$. We define a family of big-step reduction relations $\Downarrow_c$, where c is an initial component, by the set of rules in Fig. 3.3.

We say that M *reduces to* V *w.r.t.* c when $M \Downarrow_c V$. We say that M and N *reduce to the same value w.r.t.* c when there exists a term V such that $M \Downarrow_c V$ and $N \Downarrow_c V$.

The values that a term may reduce to are controlled by the initial component at which reduction happens. More precisely, reduction at ϵ may only reduce to a

$$\mathbf{tt} \Downarrow_\epsilon \mathbf{tt} \qquad \mathbf{ff} \Downarrow_\epsilon \mathbf{ff} \qquad \mathbf{tt} \Downarrow_c \mathbf{top} \quad (c \neq \epsilon) \qquad \mathbf{ff} \Downarrow_c \mathbf{top} \quad (c \neq \epsilon)$$

$$\lambda x.M \Downarrow_{1\cdot c} \lambda x.M \qquad \lambda x.M \Downarrow_c \mathbf{top} \quad (c \neq 1 \cdot c')$$

$$\langle M, N\rangle \Downarrow_{l\cdot c} \langle M, N\rangle \qquad \langle M, N\rangle \Downarrow_{r\cdot c} \langle M, N\rangle$$

$$\langle M, N\rangle \Downarrow_c \mathbf{top} \quad (c \neq l \cdot c' \text{ or } r \cdot c')$$

$$\frac{M \Downarrow_{1\cdot c} \lambda x.M' \qquad M'[x\backslash N] \Downarrow_c P}{(M\ N) \Downarrow_c P}$$

$$\frac{M \Downarrow_{1\cdot c} \mathbf{top}}{(M\ N) \Downarrow_c \mathbf{top}}$$

$$\frac{M \Downarrow_{l\cdot c} \langle N, P\rangle \qquad N \Downarrow_c N'}{\pi_l(M) \Downarrow_c N'} \qquad \frac{M \Downarrow_{r\cdot c} \langle N, P\rangle \qquad P \Downarrow_c P'}{\pi_r(M) \Downarrow_c P'}$$

$$\frac{M \Downarrow_{l\cdot c} \mathbf{top}}{\pi_l(M) \Downarrow_c \mathbf{top}} \qquad \frac{M \Downarrow_{r\cdot c} \mathbf{top}}{\pi_r(M) \Downarrow_c \mathbf{top}}$$

$$\frac{M \Downarrow_\epsilon \mathbf{tt} \qquad N \Downarrow_\epsilon N'}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi} \Downarrow_\epsilon N'} \qquad \frac{M \Downarrow_\epsilon \mathbf{ff} \qquad P \Downarrow_\epsilon P'}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi} \Downarrow_\epsilon P'}$$

$$\frac{M \Downarrow_\epsilon \mathbf{top}}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi} \Downarrow_\epsilon \mathbf{top}}$$

$$\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi} \Downarrow_c \mathbf{top} \quad (c \neq \epsilon)$$

Figure 3.3: Big-step semantics with errors and initial components

ground value, reduction at $1 \cdot c$ only to a function, and reduction at $l \cdot c$ or $r \cdot c$ only to a pair.

Lemma 2 *If M is a closed term and c an initial component such that $M \Downarrow_c V$, then one of the following is true:*

- $V = \mathbf{top}$*; or*
- $c = \epsilon$ *and* $V = \mathbf{ff}$ *or* $V = \mathbf{tt}$*; or*
- c *is of the form* $1 \cdot c'$ *and* V *is of the form* $\lambda x.M'$*; or*
- c *is of the form* $l \cdot c'$ *or* $r \cdot c'$*, and* V *is of the form* $\langle N, P\rangle$*.*

Proof: by induction on the derivation of $M \Downarrow_c V$. ■

Lemma 3 *If $\langle M, N\rangle \Downarrow_c P$, then either c is of the form $l \cdot c'$ or $r \cdot c'$, or $P = \mathbf{top}$.*
If $\lambda x.M \Downarrow_c N$, then either c is of the form $1 \cdot c'$ or $N = \mathbf{top}$.
If $\mathbf{tt} \Downarrow_c N$ or $\mathbf{ff} \Downarrow_c N$, then either $c = \epsilon$ or $N = \mathbf{top}$.

Proof: by exhaustive verification of all the rules applying to the term being reduced. ■

3.1.7 Observational preorder

In order to complete the definition of the semantics of our calculus, we need to introduce a notion of equivalence of terms. This is usually done by defining a set of *observations*, which is then used to define a congruent preorder on terms known as the *observational preorder*.

Of course, we will want the preorder to take into account the difference between looping and reduction to **top**. In a view to doing so, we need to consider not only the usual reduction to **tt** and to **ff**, but also reduction to **top**. We choose our set of observations to consist of the observations "reduction to top at ϵ," "reduction to **tt** at ϵ" and "reduction to **ff** at ϵ," ordered as in Fig. 3.4 (note the analogy with Fig. 3.2(c)).

The observational preorder is then defined, as usual, to be the least discrete (largest) preorder that makes contexts monotone.

Definition 4 (Observational preorder)

$$M \lesssim N \text{ iff } \forall C[\cdot] \begin{cases} C[M] \Downarrow_\epsilon \mathbf{top} \Rightarrow C[N] \Downarrow_\epsilon \mathbf{top}; \\ C[M] \Downarrow_\epsilon \mathbf{tt} \Rightarrow C[N] \Downarrow_\epsilon \mathbf{tt} \text{ or } C[N] \Downarrow_\epsilon \mathbf{top}; \\ C[M] \Downarrow_\epsilon \mathbf{ff} \Rightarrow C[N] \Downarrow_\epsilon \mathbf{ff} \text{ or } C[N] \Downarrow_\epsilon \mathbf{top}. \end{cases}$$

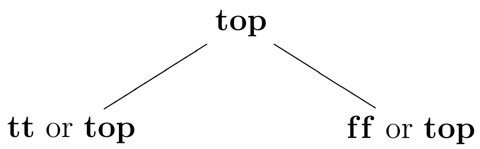


Figure 3.4: Observations

We say that two terms M and N are observationally equivalent*, and write $M \cong N$, when $M \lesssim N$ and $N \lesssim M$.*

As usual in calculi with ground values, the observational preorder can be defined by just one well-chosen observation. In our case, this is the top observation; informally, this lemma says that terms are equivalent if and only if they generate errors in the same set of contexts.

Lemma 5 *$M \lesssim N$ if and only if $\forall C[\cdot]\ C[M] \Downarrow_\epsilon \mathbf{top} \Rightarrow C[N] \Downarrow_\epsilon \mathbf{top}$.*

Proof: the condition is necessary by the first clause in Definition 4. To prove that it is sufficient, let M, N be two terms such that for any context $C[\cdot]$,

$$C[M] \Downarrow_\epsilon \mathbf{top} \Rightarrow C[N] \Downarrow_\epsilon \mathbf{top}$$

Let $D[\cdot]$ be a context; we want to prove that

$$\begin{aligned} D[M] \Downarrow_\epsilon \mathbf{top} &\Rightarrow D[N] \Downarrow_\epsilon \mathbf{top}; \\ D[M] \Downarrow_\epsilon \mathbf{tt} &\Rightarrow D[N] \Downarrow_\epsilon \mathbf{tt} \text{ or } D[N] \Downarrow_\epsilon \mathbf{top}; \\ D[M] \Downarrow_\epsilon \mathbf{ff} &\Rightarrow D[N] \Downarrow \mathbf{ff} \text{ or } D[N] \Downarrow_\epsilon \mathbf{top}. \end{aligned}$$

If $D[M] \Downarrow_\epsilon \mathbf{top}$, the result is immediate.

Assume that $D[M] \Downarrow_\epsilon \mathbf{tt}$. Let

$$E[\cdot] = \mathbf{if}\ D[\cdot]\ \mathbf{then\ top\ else}\ \Omega\ \mathbf{fi}$$

$E[M] \Downarrow_\epsilon \mathbf{top}$; this implies that $E[N] \Downarrow_\epsilon \mathbf{top}$, and therefore $D[N] \Downarrow_\epsilon$.

Clearly, $D[N]$ cannot reduce to $\mathbf{ff}$, as then $E[N]$ would diverge. By Lemma 2, either $D[N] \Downarrow_\epsilon \mathbf{top}$, or $D[N] \Downarrow_\epsilon \mathbf{tt}$, which proves the desired property.

In the remaining case $D[M] \Downarrow_\epsilon \mathbf{ff}$, we take $E'[\cdot]$ to be

$$E'[\cdot] = \mathbf{if}\ D[\cdot]\ \mathbf{then}\ \Omega\ \mathbf{else\ top\ fi}$$

and proceed analogously. ■

On the other hand, choosing reduction to $\mathbf{tt}$ and reduction to $\mathbf{ff}$ as the only observations would not be sufficient as this would identify $\mathbf{top}$ and Ω.

A detailed analysis of the semantics of λ-calculi was done by Abramsky [Abr90]. Abramsky distinguishes between the *standard theory* of the λ-calculus, or *call-by-name* semantics, in which Ω and $\lambda x.\Omega$ are equivalent, and the *lazy* theory, in which they are not, and which is arguably closer to a number of programming languages. As we shall see (Corollary 56 on page 84), our calculus identifies **top** and $\lambda x.$**top**, which gives it a call-by-name "feel"; we will model it by using the simple unlifted function space which is characteristic of the standard theory. On the other hand, it enjoys the most important property of the lazy theory, namely that of distinguishing between Ω and $\lambda x.\Omega$; indeed, as we saw before,

$$\mathbf{if}\ \Omega\ \mathbf{then\ tt\ else\ ff\ fi} \Uparrow_\epsilon$$

while

$$\mathbf{if}\ \lambda x.\Omega\ \mathbf{then\ tt\ else\ ff\ fi} \Downarrow_\epsilon \mathbf{top}.$$

3.2 Types

In this section, we introduce a type assignment for the calculus defined earlier. The type system has second order features, which means that type environments need to hold information about both type variables and variables.

There is a large body of literature on first- and second-order type systems with subtyping [Rey88, Com95, AC93, SP94]. Unlike most such systems, however, the system that we introduce is a Curry-style type assignment, which allows us to separate the treatment of dynamics from that of types; this allows for a better structuring of the semantics. In addition, this system minimises the number of typing rules, preferring instead to introduce subtyping rules (examples are given in Section 3.2.2 at the end of this chapter). This is important, as proving the soundness of typing rules is somewhat more tedious than proving that of subtyping rules (Chapter 6).

Typechecking is quite likely to be undecidable in this system. This is not as deadly a flaw as one might first imagine, as it is quite reasonable in a system with subtyping to allow implementations to "bump up" the types of terms they cannot deal with; evaluation of such terms proceeds in a dynamically typed manner. A programmer whose program has type $\top$ is made aware that his code may likely but not necessarily contain typing errors. At any rate, the study of algorithms for typechecking and type inference of systems with subtyping is a wide area [Com95, AC93, PWO97], and one that is outside the scope of this thesis.

The syntax of types is defined as follows:

$$\begin{aligned}A, B ::= &\mathbf{Bool} \mid \top \mid X \mid A \times B \mid A \to B \\ &\forall X \leq A.B[X] \mid \exists X \leq A.B[X] \mid \mu X.C[X]\end{aligned}$$

where the type $C[X]$ is *guarded* in X (see below). We write $\bot$ for the type $\forall X \leq \top.X$, and **TyVar** for the set of type variables.

The set of types *guarded* in a type variable X is defined by the following grammar:

$$\begin{aligned}C, D ::= &\mathbf{Bool} \mid \top \mid Y \mid A \times B \mid A \to B \\ &\forall Y \leq C.D \mid \exists X \leq C.D \mid \mu Y.E\end{aligned}$$

where $Y \neq X$, A and B are arbitrary (not necessarily guarded) types and E is guarded both in X and Y

The set of types *covariant* in a type variable X is defined by the grammar

$$\begin{aligned}A[X^+], B[X^+] ::= &\mathbf{Bool} \mid \top \mid X \mid Y \mid A[X^+] \times B[X^+] \mid C[X^-] \to B[X^+] \\ &\forall Y \leq C[X^-].B[X^+, Y] \mid \exists Y \leq A[X^+].B[X^+, Y] \\ &\mu Y.B[X^+, Y^+] \mid \mu Y.D[Y]\end{aligned}$$

where $Y \neq X$, $C[X^-]$ is *contravariant* in X, X does not appear free in $D[Y]$. The set of types *contravariant* in X is, similarly, defined by the grammar

$$\begin{aligned}A[X^-], B[X^-] ::= &\mathbf{Bool} \mid \mathbf{top} \mid Y \mid A[X^-] \times B[X^-] \mid C[X^+] \to C[X^-] \\ &\forall Y \leq C[X^+].B[X^-, Y] \mid \exists Y \leq A[X^-].B[X^-, Y] \\ &\mu Y.B[X^-, Y^+] \mid \mu Y.D[Y]\end{aligned}$$

where, again, $Y \neq X$, $C[X^+]$ is covariant in X, and X does not appear free in $D[Y]$.

3.2.1 Type environments and typing judgements

A type environment is a (partial) map with finite domain that maps both type variables and variables to types. We write $\emptyset$ for the empty environment,

$$E, x : A$$

for the environment that is equal to E except that it maps x to A, and

$$E, X \leq B$$

for the environment that is equal to E except that it maps X to B.

There is more than one class of *typing judgement*. A judgement of the form

$$E \vdash \diamond$$

where E is an environment, states that the environment E is well formed. A judgement of the form

$$E \vdash A$$

states that the type A is well formed under the well formed environment E. A judgement of the form

$$E \vdash A \leq B$$

states that A and B are well formed types under the well formed environment E, and that, under E, A is provably a subtype of B. Finally, a type judgement of the form

$$E \vdash M : A$$

states that the term M is typable under the well-formed environment E, and that it may be assigned type A.

We write $\mathrm{dom}(E)$ for the domain of the environment E; $\mathrm{dom}(E)$ is a finite set of type variables and variables.

Fig. 3.5 presents the rules used for forming types and type environments. Fig. 3.6 presents the rules used for typing terms; the last of these relies crucially on the rules for proving subtyping between types which are presented in Figures 3.7 through 3.9.

Fig. 3.9 presents two additional rules used for dualising quantifiers. In this figure, we use the abbreviation

$$E \vdash A = B$$

to mean that $E \vdash A \leq B$ and $E \vdash B \leq A$.

3.2.2 Derived rules

Unlike most calculi, our calculus does not include folding and unfolding of recursive types or introduction and elimination rules for quantifiers. Instead, subtyping rules are provided for recursive types and quantifiers; the said rules can then be derived from subsumption with the help of the rules in figure 3.9

$$\emptyset \vdash \diamond$$

$$\frac{E \vdash A}{E, X \leq A \vdash \diamond}$$

$$\frac{E \vdash A}{E, x : A \vdash \diamond}$$

$$\frac{E, X \leq A \vdash \diamond}{E, X \leq A \vdash X}$$

$$\frac{E \vdash \diamond}{E \vdash \mathbf{Bool}}$$

$$\frac{E \vdash A \quad E \vdash B}{E \vdash A \times B}$$

$$\frac{E, X \leq A \vdash B[X]}{E \vdash \forall X \leq A.B[X]}$$

$$\frac{E, X \leq A \vdash B[X]}{E \vdash \exists X \leq A.B[X]}$$

$$\frac{E, X \leq \top \vdash B[X]}{E \vdash \mu X.B[X]}$$

Figure 3.5: Rules for the formation of environments and types

$$\frac{E, x : A \vdash \diamond}{E, x : A \vdash x : A}$$

$$\frac{E, x : A \vdash M : B}{E \vdash \lambda x.M : A \to B}$$

$$\frac{E \vdash M : A \to B \qquad E \vdash N : B}{E \vdash (M\ N) : B}$$

$$\frac{E \vdash \diamond}{E \vdash M : \top} \quad \text{if } \mathrm{FV}(M) \subseteq \mathrm{dom}(E)$$

$$E \vdash \mathbf{tt} : \mathbf{Bool} \qquad E \vdash \mathbf{ff} : \mathbf{Bool}$$

$$\frac{E \vdash M : \mathbf{Bool} \qquad E \vdash N : A \qquad E \vdash P : A}{E \vdash \mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi} : A}$$

$$\frac{E \vdash M : A \quad E \vdash N : B}{E \vdash \langle M, N \rangle : A \times B}$$

$$\frac{E \vdash M : A \times B}{E \vdash \pi_l(M) : A} \qquad \frac{E \vdash M : A \times B}{E \vdash \pi_r(M) : B}$$

$$\frac{E, X \leq A \vdash M : B[X]}{E \vdash M : \forall X \leq A.B[X]}\ X \text{ does not appear in } E$$

$$\frac{E \vdash M : A \qquad E \vdash A \leq B}{E \vdash M : B}$$

Figure 3.6: Typing rules

$$E \vdash A \leq A$$

$$\frac{E \vdash A \leq B \qquad E \vdash B \leq C}{E \vdash A \leq C}$$

$$E \vdash A \leq \top \qquad E \vdash \top \leq A \to \top$$

Figure 3.7: Subtyping rules

$$\frac{E \vdash A' \leq A \qquad E \vdash B \leq B'}{E \vdash A \to B \leq A' \to B'}$$

$$\frac{E \vdash A \leq A' \quad E \vdash B \leq B'}{E \vdash A \times B \leq A' \times B'}$$

$$\frac{E \vdash C \leq A}{E \vdash \forall X \leq A.B[X] \leq B[C]}$$

$$\frac{E \vdash C \leq A}{E \vdash B[C] \leq \exists X \leq A.B[X]}$$

$$\frac{E, X \leq A \vdash B[X] \leq B'[X]}{E \vdash (\forall X \leq A.B[X]) \leq (\forall X \leq A.B'[X])}$$

$$\frac{E, X \leq A \vdash B[X] \leq B'[X]}{E \vdash (\exists X \leq A.B[X]) \leq (\exists X \leq A.B'[X])}$$

$$\frac{E \vdash A \leq A'}{E \vdash (\forall X \leq A'.B[X]) \leq (\forall X \leq A.B[X])}$$

$$\frac{E \vdash A \leq A'}{E \vdash (\exists X \leq A.B[X]) \leq (\exists X \leq A'.B[X])}$$

$$\frac{E, X \leq \top \vdash A[X^+] \leq B[X^+]}{E \vdash \mu X.A[X^+] \leq \mu X.B[X^+]} \quad A[X^+] \text{ and } B[X^+] \text{ covariant in } X.$$

Figure 3.8: Subtyping rules (continued)

$$E \vdash \mu X.B[X] = B[\mu X.B[X]]$$

$$E \vdash \forall X \leq A.(B[X] \to C) = (\exists X \leq A.B[X] \to C) \qquad X \text{ not free in } C$$

$$E \vdash \exists X \leq A.(B[X] \to C) = (\forall X \leq A.B[X] \to C) \qquad X \text{ not free in } C$$

Figure 3.9: Subtyping rules: recursive types and dualisation of quantifiers

3.2.2.1 Universal quantifier

Usual Curry-style presentations of the rule of $\forall$-elimination consist of something like [AC96, Section 13.1]:

$$\frac{E \vdash M : \forall X \leq A.B[X] \qquad E \vdash A' \leq A}{E \vdash M : B[A']}$$

This rule is not part of our presentation. However, a single application of subsumption shows that it can be derived from the subtyping rule for the universal quantifier:

$$\frac{E \vdash M : \forall X \leq A.B[X] \qquad \dfrac{E \vdash A' \leq A}{E \vdash \forall X \leq A.B[X] \leq B[A']}}{E \vdash M : B[A']}$$

3.2.2.2 Existential quantifier

The rules for introduction and elimination of the bounded existential quantifier $\exists$ are usually presented as follows [AC96, Section 13.2]:

$$\frac{E \vdash M : B[C] \qquad E \vdash C \leq A}{E \vdash \textbf{pack } X \leq A = C \textbf{ with } M : B[X] \ : \ \exists X \leq A.B[X]}$$

$$\frac{E \vdash M : \exists X \leq A.B[X] \qquad E \vdash X \leq A, x : B[X] \vdash N : D}{E \vdash \ \textbf{open } M \textbf{ as } X \leq A, x : B \textbf{ in } N : D \ : \ D} \ X \text{ not free in } D$$

In a Curry-style presentation such as ours, it is natural to consider the **pack** and **open** constructs as syntactic sugar. The rules thus become, respectively,

$$\frac{E \vdash M : B[C] \qquad E \vdash C \leq A}{E \vdash M : \exists X \leq A.B[X]}$$

$$\frac{E, X \leq A, x : B[X] \vdash N : D \qquad E \vdash M : \exists X \leq A.B[X]}{E \vdash (\lambda x.N)M : D} \ X \text{ not free in } D$$

The former is a direct application of the subtyping rule for existentials and subsumption. The latter may be derived as follows:

$$\frac{\dfrac{\dfrac{\dfrac{E, X \leq A, x : B[X] \vdash N : D}{E, X \leq A \vdash \lambda x.N : B[X] \to D}}{E \vdash \lambda x.N : \forall X \leq A.(B[X] \to D)}}{E \vdash \lambda x.N : (\exists X \leq A.B[X]) \to D} \qquad E \vdash M : \exists X \leq A.B[X]}{E \vdash (\lambda x.N)M : D}$$

Note in particular how the dualisation rule for quantifiers is used.

3.2.2.3 Recursive types

The usual presentation of the folding and unfolding rules for quantifiers is

$$\frac{E \vdash M : \mu X.B[X]}{E \vdash M : B[\mu X.B[X]]}$$

$$\frac{E \vdash M : B[\mu X.B[X]]}{E \vdash M : \mu X.B[X]}$$

These rules can be recovered in our calculus by a simple application of the first equality in Fig. 3.9.

3.2.3 Type safety

The main property that we require a typing system to have is that of *type safety*. The exact statement of type safety depends on the calculus used, but intuitively it says that *well-typed terms cannot go wrong.*

Obviously, as every term in our calculus types to $\top$, we will have to choose a reasonable notion of "good" type, one that is not a supertype of $\top$. Hopefully, **Bool** is "good," so we should be able to prove the following property:

If $\vdash M : \mathbf{Bool}$ then it is not the case that $M \Downarrow_\epsilon \mathbf{top}$.

This property will appear as an immediate consequence of the soundness of our interpretation; see Corollary 108 on page 124.

Chapter 4

Game Semantics for an untyped calculus

In this chapter, we show how to interpret the untyped calculus presented in Section 3.1 in game semantics. After a quick informal introduction to game semantics, we formalise all the notions that we will need. We then spell out the interpretation, and prove it to be sound.

4.1 Introduction to Game Semantics

In Game Semantics, a term is considered as a black box *system* that interacts with its *environment*. The abstraction of the system is known as *Player*, and typically represents the term under study, while the environment is known as *Opponent*, and represents the rest of the universe: the term's context, the computer it is running in, the user of the computer, and any supernova in the neighbourhood. We do not assume anything about the Opponent's behaviour*.

4.1.1 Main ideas

Player and Opponent interact by exchanging tokens of information known as *moves*; one might want to think of those as messages in Smalltalk [GR83] and other object-oriented systems based on the message-passing paradigm, or as (visible) actions in process calculi such as CSP [Hoa85] or CCS [Mil80]. A typical interaction, in which Player represents the function that adds 1 to its argument, might proceed as follows:

O[0]: What's the result?

*The situation in which Player is playing against Opponent is not quite symmetric, as Player's behaviour is defined by a *strategy*, while we don't consider the symmetric notion of a *counter-strategy* (unlike, for example, Cartwright *et al.* [CCF94])

P[1]: What's the argument?

O[2]: The argument is 5.

P[3]: The result is 6. *(exeunt)*

Notice that the first move (numbered [0]) is Opponent's, and that moves alternate: all even-numbered moves are Opponent's moves, while odd-numbered ones are Player's.

Moves will need to specify whether they initiate a new dialog or continue a former one. To this end, every move is optionally decorated with a *justification pointer* that refers to a previously played move. The previous dialog therefore becomes:

O[0]: What's the result? This move is unjustified.

P[1]: What's the argument? This move is justified by move [0].

O[2]: The argument is 5. This move is justified by move [1].

P[3]: The result is 6. This move is justified by move [0].

Justifications follow certain rules. First of all, a Player's move is always justified by an Opponent's one, and conversely, an Opponent's move can only be justified by a Player's one (a player cannot justify himself). Furthermore, only a question can be unjustified, while an answer is always justified by the question it answers. Finally, suppose that one depicts a question by an open bracket and an answer by a closed bracket; then the sequence is properly bracketed (*i.e.* every prefix of the sequence of brackets contains at least as many opening brackets as closing ones), and the closed bracket associated to an answer is "matched" with the open bracket associated to the question it answers[†].

Distinguishing between arguments and results is done by decorating each move with a *component*; if you think of the interaction considered above as happening at type $\mathbf{N} \to \mathbf{N}$, results are exchanged in the right hand side of the arrow, known as component 1; arguments are exchanged in component 0. At a more complex type, such as $(\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$, the three components would be 00, 01 and 1 (think of the syntax tree of the type). With the addition of explicit components, the dialog above has lost all of its theatrical nature:

O[0]: Question in component 1. This move is unjustified.

[†]This discipline needs to be broken in order to model non-local jumps, such as exceptions or continuations.

P[1]: Question in component 0. This move is justified by move [0].

O[2]: Answer 5 in component 0. This move is justified by move [1].

P[3]: Answer 6 in component 1. This move is justified by move [0].

There is a handy graphical representation of positions, an example of which is given in Fig. 4.1, in which moves are written in cells in a table. Later moves are presented below earlier ones, and the figure should thus be read vertically. Columns of the table represent components of the type, and the lines connecting moves represent justification pointers. In Fig. 4.2, Player represents the curried left addition function (the addition function that computes its left argument first)

$$\lambda x.\lambda y.x + y$$

of type $\mathbf{N} \to \mathbf{N} \to \mathbf{N}$, the components of which are 0, 10 and 11. Opponent applies the term to the values 5 and 6.

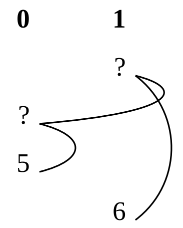


Figure 4.1: $\lambda x.x + 1$ playing against 5

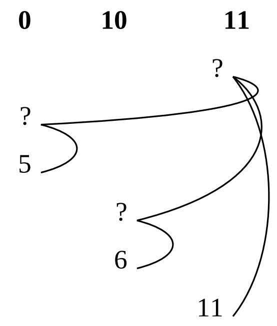
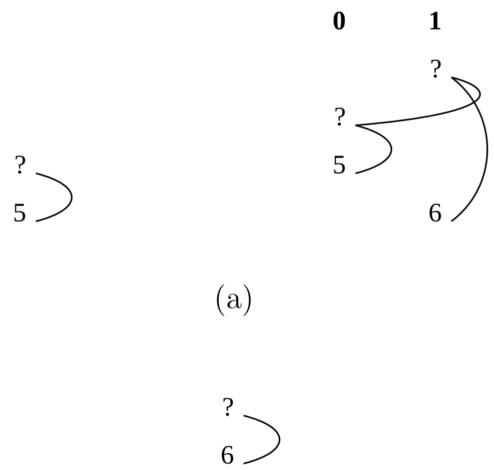


Figure 4.2: $\lambda x.\lambda y.x + y$ playing against 5 and 6

A set of rules that specify the way Player responds to an Opponent's move is known as a *strategy*. Terms are interpreted as strategies.

4.1.2 Composition

Composition and application of terms is modelled by copying of moves from one strategy to another. Consider, for instance, the composition $((\lambda x.x + 1)\ 5)$, and the two positions in Fig. 4.3(a). One way of understanding this is that the initial question is asked to the composed term; this is copied to the function in its result (1) component. The function then asks a subsidiary question in its argument (0) component, question which is copied to an initial question asked to the argument. The argument's reply is then copied back to the function, which at last replies to the initial question.



(a)

(b)

Figure 4.3: Composition of strategies

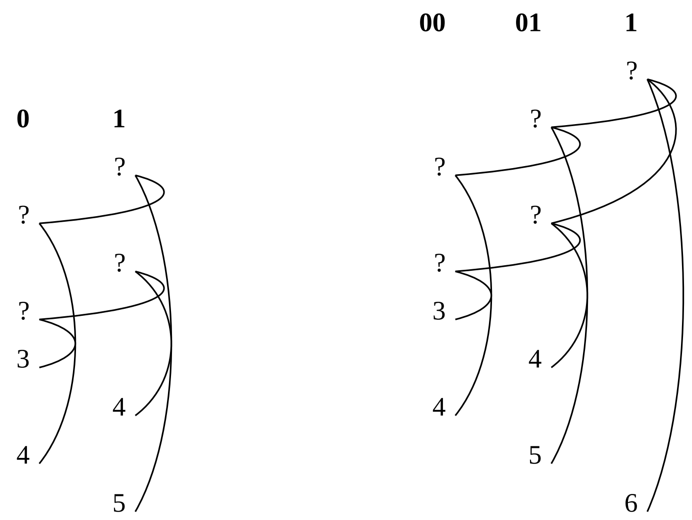
It is important that this internal interaction is not visible to the environment. Technically, this is achieved through a mechanism similar to parallel composition with hiding, together with some renaming. Concretely, moves exchanged between the function and its argument are hidden, while moves in the function's result component are renamed to be in a principal component, leading to the position in Fig. 4.3(b).

4.1.3 Copy-cat

An important class of strategies are strategies that simply copy moves from one component into another. Such strategies serve to model the proper routing of data between arguments of an application and instances of variables; equivalently, they represent combinators such as **I**, **S**, **K** or **Y**.

4.1.4 Further comments

In general, a single term can be invoked multiple times simultaneously. This is represented in Game Semantics by multiple initial questions being active (unanswered) simultaneously. An example of this is given in Fig. 4.4(b), where Player represents the term $\lambda f.(f\ (f\ 3))+1$, and Opponent applies it to the term $\lambda x.x+1$; the type is $(\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$, and the components 00, 01 and 1. Fig. 4.4(a) shows the corresponding position in the argument $\lambda x.x+1$; one can see that two unrelated threads are active simultaneously.



(a) (b)

Figure 4.4: A double invocation of a position

A more complex example can be given by a computation of a recursive function. In Fig. 4.5, Player represents the helper function for the computation of a recursive factorial, *viz.* the term

$$\lambda f.\lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \cdot f(x-1)\ \mathbf{fi}$$

The type is $(\mathbf{N} \to \mathbf{N}) \to \mathbf{N} \to \mathbf{N}$, the components of which are 00, 01, 10 and 11. Opponent applies the term to the Y combinator and to the value 1.

It is important to realise that Player need not reply to a move by opponent; this is used to model non-termination. If Player represents the looping term $\Omega = (\mathbf{Y}\ \mathbf{I})$, for example, it fails to reply to any move; due to the strict alternation moves, there is no way Opponent can preempt such a looping Player.

Fig. 4.6 shows the left identity function being applied to Ω (and some other value); after Ω is invoked, the addition function never gets a chance to play any

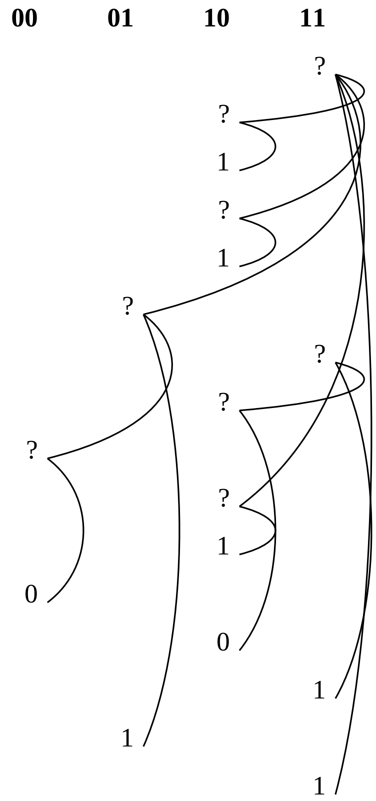


Figure 4.5: A position of the helper function for recursive factorial computation

other move[‡].

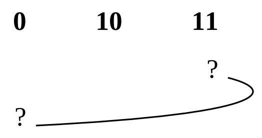


Figure 4.6: The left addition function playing against a looping term

Finally — and this feature is a novel aspect of the games used in this thesis — Player can actually refuse a move from Opponent. This will be used to model untrappable typing errors (Section 3.1.3). For example, if Player represents the integer 5, it will not accept any move that is a non-null component (indicating that it has been invoked by an Opponent representing a context with a hole of non-ground type).

A refusal can propagate backwards. Consider, for instance, the composition (**I** (**I** 5)), and suppose that Opponent attempts to play an initial move in a non-null component. The first Identity attempts to copy this move to the second one; the second one attempts to copy it to the integer 5, which refuses it; the refusal propagates backwards all the way to Opponent.

4.2 Concrete representation

This section presents a concrete representation for the notions presented in the previous section. While it introduces no new notions, it fixes many notations that will be used in the sequel. Most of the definitions are special cases of the notions used by McCusker [McC96, Section 3.1], but many have a more concrete structure, and therefore "low level" flavour; this concrete structure will be needed when we measure positions and games in Section 5.4.3.

In order to carry out our constructions, we make heavy use of sequences. We write ϵ for the empty sequence (the sequence of length 0), and $s \cdot t$ for the concatenation of sequences s and t. We write $|s|$ for the length of the sequence s, which is a natural integer. Given a sequence $s \neq \epsilon$, we write s_{-1} for the prefix of s of length $|s| - 1$, *i.e.* the sequence s without its last element.

The set M of *unjustified moves* is inductively defined as follows:

- q, $a^{\mathbf{tt}}$, $a^{\mathbf{ff}}$ are unjustified moves;

[‡]Which is the main reason why games faithfully model sequential calculi.

- if m is an unjustified move, then so are m_0, m_1, m_l, m_r.

As a shorthand, we will write $m_{e \cdot f \cdot g}$ for $((m_g)_f)_e$ (note the inversion). An unjustified move is therefore of the form t_c, where t is one of q (the *question*), $a^{\mathbf{tt}}$ (the *answer true*) or $a^{\mathbf{ff}}$ (the *answer false*), and c, the *component* of t_c, a finite sequence over $\{0, 1, l, r\}$. An unjustified move m is said to be *in* component c if it is of the form m'_c for some unjustified move m'; in other words, if c is a prefix of the component of m. We write M for the set of unjustified moves.

> **Note:** *in examples, we will often supplement the previous collection of moves with moves a^n, for any integer n. These moves have all the same properties as the moves $a^{\mathbf{tt}}$ and $a^{\mathbf{ff}}$.*

Attributing moves to Opponent and Player, and defining whether they are questions or answers is done by providing a *labelling* function λ over moves. The function

$$\lambda : M \to \{O, P\} \times \{Q, A\}$$

is inductively defined by:

$$\begin{aligned} \lambda(q) &= OQ; \\ \lambda(a^{\mathbf{tt}}) = \lambda(a^{\mathbf{ff}}) &= PA; \\ \lambda(m_1) = \lambda(m_l) = \lambda(m_r) &= \lambda(m); \\ \lambda(m_0) &= \overline{\lambda}(m); \end{aligned}$$

where $\overline{\lambda}$ is derived from λ by inverting the Opponent/Player nature of moves; formally,

$$\begin{array}{ll} \overline{\lambda}(m) = PQ & \text{if } \lambda(m) = OQ \\ \overline{\lambda}(m) = PA & \text{if } \lambda(m) = OA \\ \overline{\lambda}(m) = OQ & \text{if } \lambda(m) = PQ \\ \overline{\lambda}(m) = OA & \text{if } \lambda(m) = PA \end{array}$$

When we introduce positions later in this section, some moves will be justified by others. In order to determine which moves may be played without justification, and which moves can be justified by which other moves, a relation $\vdash \subseteq M + M \times M$, the *enabling relation*, is defined. We write $\vdash m$ for $m \in \vdash$, and $m \vdash n$ for $(m, n) \in \vdash$. The relation $\vdash$ is inductively defined by

- $\vdash q$, $q \vdash a^{\mathbf{tt}}$ and $q \vdash a^{\mathbf{ff}}$;
- if $\vdash m$ then $\vdash m_1$, $\vdash m_l$ and $\vdash m_r$;

- if $\vdash m$ and $\vdash n$ then $m_1 \vdash n_0$;

- if $m \vdash n$ then $m_1 \vdash n_1$, $m_0 \vdash n_0$, $m_l \vdash n_l$ and $m_r \vdash n_r$.

We say that a move m is *initial* when $\vdash m$, and that a move m *enables* a move n when $m \vdash n$.

The following properties of $\vdash$ are immediate consequences of its definition:

(e1) if $\vdash m$ then $\lambda(m) = OQ$ and there is no n such that $n \vdash m$;

(e2) if $m \vdash n$ where n is an answer, then m is a question;

(e3) if $m \vdash n$, then either m is Opponent's and n Player's, or m is Player's and n Opponent's.

These properties correspond to McCusker's axioms (e1), (e2) and (e3) [McC96, Section 3.1]. Therefore, the triple $(M, \lambda, \vdash)$ is a *Computational Arena* in the sense of McCusker. In fact, it is isomorphic to the Arena that McCusker associates to the type

$$\mu X\ .\ \mathbf{Bool}\,\&\,(X \otimes X)\,\&\,(X \multimap X),$$

or, in an intuitionistic setting (the distinction being irrelevant at the level of Arenas),

$$\mu X\ .\ \mathbf{Bool}\ \times\ (X \times X)\ \times\ (X \to X),$$

where **Bool** is the arena of Boolean values.

A *justified move*, or simply *move*, is either an unjustified move or a pair (m, i) where m is an unjustified move and i a natural integer, known as the move's *justification*. A *justified sequence* is a sequence $m_1 \cdots m_i$, where every m_k is a move which either has no justification and $\vdash m_k$, or has a justification j such that $j < k$ and $m_j \vdash m_k$. If the move m_k has a justification j, then m_j is said to *justify* m_k. We write ϵ for the sequence of length 0.

Example 6 *Assume that the set of tokens is extended with tokens a^4 and a^5, used for representing answers carrying the integers 4 and 5. The example dialogue in Fig. 4.1 is then represented as the justified sequence*

$$q_1 \cdot (q_0, 0) \cdot (a^5_0, 1) \cdot (a^6_1, 0)$$

A justified sequence s is *well-formed* if:

(w1) players alternate: if $s = t \cdot m \cdot n \cdot u$, then either m is a Player's move and n is an Opponents move, or m is an Opponent's move and n is a Player's move;

(w2) it is *well bracketed*: any prefix t of s contains at most as many answers as questions, and any answer is justified by the matching question.

If we were interested in full abstraction, we would need to further constrain the set of justified sequences that we accept; as we are not, we impose no supplementary conditions on justified sequences. We therefore use the term *legal position*, or simply *position*, as a synonym for well-formed justified sequence.

4.3 Strategies

Terms will be interpreted by *strategies*. A strategy is the set of positions that can actually happen for a given player.

It is not reasonable to expect any set of positions to constitute a strategy. First of all, a strategy should only contain positions that can be reached; this is captured by *even-prefix-closedness*. Furthermore, a strategy needs to fully specify the behaviour of Player against any Opponent, indicating uniquely both whether a move is played, and, if any, which move it is. This is captured by *determinacy*.

Definition 7 *A set s of positions is*

- prefix-closed *if $p \cdot q \in s$ implies $p \in s$ for any sequences p and $p \cdot q$;*
- even-prefix-closed *if $p \cdot q \in s$ and $|p|$ even imply $p \in s$ for any sequences p and $p \cdot q$;*
- deterministic *if for any position $p \in s$, if $|p|$ is odd then, for any moves m and n,*
  - *$p \cdot m \in s$ and $p \cdot n \in s$ imply $m = n$; and*
  - *$p \cdot m \in s$ implies $p \notin s$.*

Given a set A of positions, we write $\mathrm{Pref}(A)$ for the *prefix closure* of A, *i.e.* the smallest prefix-closed superset of A, or, alternatively, the set of all prefixes of elements of A.

Definition 8 *A strategy is a non-empty even-prefix-closed deterministic set of positions. We write $\mathcal{S}$ for the set of strategies.*

It is to be noted that even-length and odd-length positions in a strategy correspond to subtly different intuitions. An even length position may occur during a play of a strategy s if and only if it is in s; on the other hand, an odd length position p may occur during a play of a strategy s either if it is in s or there is a suffix of p that is in s. This assymetry will have to be taken into account when we introduce the liveness ordering in Section 4.4, but will disappear when we introduce games in Chapter 5. This interpretation of even and odd positions has wider implications, as witnessed by the work of Harmer and McCusker [Har99, Chapter 4].

An obvious consequence of the determinacy condition is that once a strategy fails to respond to a move, no further moves can be played. In other words, while a strategy s may contain odd-length positions, such positions cannot be extended.

Lemma 9 *If p is a position in a strategy s, then p has no odd-length proper prefix in s.*

Proof: consider the set of all even-length prefixes of p; this set can be written as $\{p_i \cdot m_i \mid i \in I\}$ for some finite set I; clearly, $\{p_i \mid i \in I\}$ is the set of strict odd-length prefixes of p. As s is even-prefix closed, $p_i \cdot m \in s$ for any $i \in I$, and therefore, as s is deterministic, $p_i \notin s$ for all $i \in I$. ■

We say that a position is of *arrow shape* when it consists of moves in components 1 and 0, and that a strategy is of arrow shape when it is a set of positions of arrow shape.

4.3.1 Projecting and renaming positions

In order to compositionally define notions such as the composition of strategies, and later to define arrow and product games, we will often need to extract and combine sub-positions from positions.

Definition 10 *Let $p \cdot m \cdot q \cdot n \cdot r$ be a position; then the move m is said to* hereditarily justify *n if there exists a sequence of moves $m_1 \cdots m_k$ in q such that m justifies m_1, m_1 justifies m_2,... and m_k justifies n.*

If $p \cdot n$ is a position, and n is a justified move in the component 1 (resp. in the component 0, resp. in the component l, resp. in the component r), then the hereditary justifier *of n is the last move m in p such that m is in the component 1 (resp. in the component 0, resp. in the component l, resp. in component r) that hereditarily justifies n.*

Definition 11 *Given a position p, the* projection *of p on the component 1, written $p \upharpoonright 1$, is inductively defined as follows:*

- *$\epsilon \upharpoonright 1 = \epsilon$;*

- *$p' \cdot m_1 \upharpoonright 1 = (p' \upharpoonright 1) \cdot m'$, where m' is m with its justification pointer, if any, renamed to point at the image of the hereditary justifier of m_1; and*

- *$p' \cdot m \upharpoonright 1 = p' \upharpoonright 1$ if m is not in component 1.*

The projection of p onto the component 0 is defined analogously:

- *$\epsilon \upharpoonright 0 = \epsilon$;*

- *$p' \cdot m_0 \upharpoonright 0 = (p' \upharpoonright 0) \cdot m'$, where m' is m with its justification pointer, if any, renamed to point at the image of the hereditary justifier of m_0;*

- *$p' \cdot m \upharpoonright 0 = p' \upharpoonright 0$ if m is not in component 0.*

The projection of p onto component l, is defined by:

- *$\epsilon \upharpoonright l = \epsilon$;*

- *$p' \cdot m_l \upharpoonright l = (p' \upharpoonright l) \cdot m'$, where m' is m with its justification pointer, if any, renamed to point at the image of the hereditary justifier of m_l;*

- *$p' \cdot m \upharpoonright l = p' \upharpoonright l$ if m is not in component l.*

The projection of p onto component r is defined analogously:

- *$\epsilon \upharpoonright r = \epsilon$;*

- *$p' \cdot m_r \upharpoonright r = (p' \upharpoonright r) \cdot m'$, where m' is m with its justification pointer, if any, renamed to point at the image of the hereditary justifier of m_r;*

- *$p' \cdot m \upharpoonright r = p' \upharpoonright r$ if m is not in component r.*

It is natural to think of projecting strategies pointwise. Projecting does not, in general, yield a strategy, as determinacy is not preserved.

Definition 12 *Given a strategy s, the projection $s \upharpoonright 1$ (resp. $s \upharpoonright l$, resp. $s \upharpoonright r$) is defined as the set of all positions $p \upharpoonright 1$ (resp. $p \upharpoonright l$, resp. $p \upharpoonright r$) where $p \in s$. This operation does not in general define a strategy (determinacy is not preserved by this operation).*

Projection has a right inverse, which, naturally enough, we call injection. Given a move m_c, the injection of m into the component 1 is the move $m_{1 \cdot c}$; analogously, the injection of m into l (resp. r) is $m_{l \cdot c}$ (resp. $m_{r \cdot c}$). The injection of a strategy s into the component 1 (resp. l, resp. r) consists of the positions in s with all its moves injected into component 1 (resp. component l, resp. component r).

Definition 13 *Given a strategy s, we write:*

- $\mathbf{K}(s)$ *for the injection of s in the component* 1*;*
- $\mathbf{I}_l(s)$ *(resp. $\mathbf{I}_r(s)$) for the injection of s in the component l (resp. r).*

4.3.2 Interleaving

Somewhat opposite to projection is the notion of *interleaving*. Given two strategies s and t, we will often want to consider the strategy that contains all the behaviours of both s and t. However, taking the union of s and t would only allow us to use the resulting strategy as s or t, while in general we would want to interleave behaviours of s and t.

Definition 14 *Given positions p and q, a position $r = r_0 \cdot r_1 \cdots r_{n-1}$ is an* interleaving *of p and q if there exist sequences of integers $(i_k)_{k<m}$ and $(j_k)_{k<m'}$ such that*

- *the sets $\{i_k \mid k < m\}$ and $\{j_k \mid j < m'\}$ constitute an exhaustive partition of the set of integers less than the length of r;*
- *for any $k < m$ (resp. for any $k < m'$), if the move r_{i_k} (resp. r_{j_k}) carries a justification pointer, then the justifier is a r_{i_l} for some l (resp. a r_{k_l} for some l).*
- *the sequences (r_{i_l}) and (r_{j_l}), with the justification pointers adjusted, are respectively equal to p and q.*

The notion of being an interleaving being associative and commutative, it generalises without trouble to interleavings of finite families of positions. Furthermore, as positions are finite, it is an essentially finitary notion; we therefore define being an interleaving of an infinite family of positions as being an interleaving of a finite subset thereof.

4.3.3 Composition of strategies

In this section, we define composition of strategies — an operation the intent of which is to model composition (and therefore application) of terms.

Composition is only interesting on a certain class of strategies — the strategies of *arrow shape* —, which, intuitively, is the class of strategies that represent maps. We recall that a position is said to be of arrow shape if it consists of moves that are either in component 1 or in component 0; a strategy is of arrow shape if all the positions it contains are of arrow shape.

Given two positions of arrow shape q and q', we say that q and q' *agree* when $q \upharpoonright 1 = q' \upharpoonright 0$ (as in Fig. 4.3). The obvious way of defining composition of two strategies s and t would be to range over agreeing positions $q \in s$ and $q' \in t$ and hide the common component; unfortunately, this approach fails to give a correct definition for two reasons: it doesn't properly match hidden moves, and it doesn't correctly take into account *infinite chattering*.

In order to solve the first problem, we introduce the notion of *interaction sequence* (following Abramsky and McCusker [AM96] or McCusker [McC96]). Intuitively, an interaction sequence is analogous to a position of arrow shape, but instead of having moves in two components 0 and 1, it has moves in *three* components, which we write α, β and γ[§]. Such an interaction sequence can then be projected on the left (on components α, β), on the right (on components β, γ) or the middle components can be hidden (by projecting on components α and γ). All of those operations will need to do some fairly natural manipulations of justification pointers.

Definition 15 *An* interacting move *is a pair* (m, κ)*, where* m *is a (justified) move and* κ *is one of* α*,* β *or* γ*.*

Let u be a sequence of interacting moves. The sequence of moves $u \upharpoonright \alpha, \beta$ is defined as the sequence of moves in u that are labelled with either α or β, renamed to be in components 0 and 1 respectively; justification pointers are kept if they point at a move in α or β, and removed otherwise. Clearly, $u \upharpoonright \alpha, \beta$ is a sequence of moves of arrow shape, but not necessarily a position.

Similarly, the sequence $u \upharpoonright \beta, \gamma$ is defined as the sequence of moves in u that are in either β or γ renamed to be in components 0 and 1 respectively, with justification pointers of moves justified by moves in α removed.

Finally, the sequence $u \upharpoonright \alpha, \gamma$ is defined as the sequence of moves in u that are either in α or γ renamed to be in components 0 and 1 respectively, with justification pointers of moves justified by moves in β replaced by a pointer to the hereditary justifier in γ.

Definition 16 *An* interaction sequence *u is a finite sequence of interacting moves such that all moves in u are justified except initial moves in component γ and both $u \upharpoonright \alpha, \beta$ and $u \upharpoonright \beta, \gamma$ are positions.*

[§] Abramsky and McCusker do not have this notational problem, as they work in a framework that is typed *a priori*. Thus, in their framework every interaction sequence lives at some type $A \to B \to C$, and they simply use the names A, B and C for the components

We say that an ω-indexed sequence of interaction sequences $(u_i)_{i\in\omega}$ is *increasing* when for any integers $i \leq j$, u_i is a prefix of u_j. We say that such a sequence is *unbounded* when there is no bound on the length of the u_i.

Definition 17 *Let s and t be two strategies. A position p is in the set $s;t$ if either*

1. *there exists and interaction sequence u such that $u \restriction \alpha, \gamma = p$ and positions $q \in s$ and $q' \in t$ such that $u \restriction \alpha, \beta = q$, $u \restriction \beta, \gamma = q'$; or*

2. *there exists an unbounded increasing sequence of interaction sequences $(u_i)_{i\in\omega}$ such that for any i, $u_i \restriction \alpha, \gamma = p$ and there exist positions $q_i \in s$ and $q'_i \in t$ such that either*

   (a) $u_i \restriction \alpha, \beta = q_i$ and $u_i \restriction \beta, \gamma = q'_i$; or

   (b) $u_i \restriction \alpha, \beta = q_i \cdot m_1$ and $u_i \restriction \beta, \gamma = q'_i$ for some move m_1 in component 1; or

   (c) $u_i \restriction \alpha, \beta = q_i$ and $u_i \restriction \beta, \gamma = q'_i \cdot n_0$ for some move n_0 in component 0.

The two cases in this definition partition the positions in $s;t$ into those that arise from boring, finitary composition (case (1)) and those that arise from infinite chattering (case (2)). A position of even-length — one that ends in a Player move — can only arise from case (1).

We say that a move in an interaction sequence u is *visible* if it is in one of the components α or γ; we say that it is *invisible* otherwise, *i.e.* if it is in component β. We say that a visible move belongs to Player (resp. to Opponent) if it belongs to Player (resp. in Opponent) in one of the sequences $u \restriction \alpha, \beta$ or $u \restriction \beta, \gamma$.

Lemma 18 *If a position $p \in s;t$ arises from case (1) in Definition 17, then,*

- *p, q and q' are all of even length and u ends in a visible move; or*

- *p is of odd length and exactly one of q and q' is of odd length.*

Proof: note that $|p|$ has the same parity as $|q| + |q'|$. Suppose first that $|q|$ and $|q'|$ are both even; then $|p|$ is even. As both q and q' end in a Player's move, the last move of u cannot be in component β.

We therefore only need to show that q and q' cannot both be of odd length. Suppose otherwise; then q and q' both end in an Opponent's move, and by the same argument as above, u ends in a visible Opponent's move; but then, p is of even length, and ends in an Opponent's move, which is impossible. ■

On the other hand, infinite chattering can only lead to a position ending in a Opponent's move — one that has odd length.

Lemma 19 *If a position $p \in s; t$ arises from case (2) in Definition 17 then*

- *for all i, q_i and q'_i are of even length;*
- *there exists i_0 such that for all $i > i_0$,*

$$u_i = u_{i_0} \cdot m_0 \cdots m_{j_i}$$

where all the m_k are in component β;

- *p is of odd length.*

Proof: The first point is an immediate consequence of Lemma 9. The second point is due to the non-hidden moves in u being (up to renaming) the moves in p, which is constant. Hence, there exists an integer i_0 such that for $i \geq i_0$, only cases (2.2) or (2.3) apply. But then, by Lemma 9 on page 52, $|q_i|$ and $|q'_i|$ have opposite parity; as $|u_i \upharpoonright \alpha, \gamma|$ has the same parity as $|q_i| + |q'_i|$, p is of odd length. ■

Lemma 20 *The composition $s; t$ of two strategies s and t is a strategy.*

Proof: let us first show that $s; t$ is even-prefix closed. Suppose first that p arises from case (1) in Definition 17, and let u, $q \in s$ and $q' \in t$ be the associated interaction sequence and positions. As s and t are strategies, and therefore even-prefix-closed, any prefix u' of u ending in a visible move such that $u \upharpoonright \alpha, \beta$ and $u \upharpoonright \beta, \gamma$ are of even length is again an interaction sequence of s and t. Therefore, all even-length prefixes of p are in $s; t$.

Suppose now that p arises from case (2) in Definition 17. By Lemma 19, p is of the form $p_0 \cdot m$, where p_0 is of even length. Let u be the shortest interaction sequence such that $u \upharpoonright \alpha, \gamma = p$, $u \upharpoonright \alpha\beta = q$ and $u \upharpoonright \beta\gamma = q'$ where either (i) $q \in s$ and $q' \cdot m \in t$ or (ii) $q \cdot m \in s$ and $q' \in t$. Clearly, the last move in u is in α, and the penultimate is in γ (in case i) or α (in case ii). Let u_0 be u without its last two moves, q_0 and q'_0 be q and q' respectively without the last two moves; then, $q_0 \in s$, $q'_0 \in t$, and u_0 gives rise to p without its last move. ■

In order to prove that composition preserves determinacy, we will first need to show that every position in $s; t$ arises from a unique interaction sequence.

Lemma 21 *Let u and v be two interaction sequences for $s; t$ such that $u \upharpoonright \alpha, \beta = v \upharpoonright \alpha, \beta$ and $u \upharpoonright \beta, \gamma = v \upharpoonright \beta, \gamma$. Then $u = v$.*

Proof: by induction on the length of u and v. The base case — that of u and v only having moves in components β and γ – is immediate.

Suppose now that the property is true for all (equal-length) prefixes of u and v. If u ends in a visible move, the property is immediate by the determinacy of s and t. If u ends in a move in β, then this move is either a Player's move in $u \upharpoonright \alpha, \beta$, or else a Player's move in $u \upharpoonright \beta, \gamma$; the determinacy of s in the first case, or that of t in the second guarantee the property. ■

Lemma 22 *Let u and v be two interaction sequences for $s;t$ that both end in a visible Opponent's move. If $u \upharpoonright \alpha, \gamma = v \upharpoonright \alpha, \gamma$, then $u = v$.*

Proof: we show by induction that for all matching prefixes u_0 of u and v_0 of v, $u_0 \upharpoonright \alpha, \beta = v_0 \upharpoonright \alpha, \beta$ and $u_0 \upharpoonright \beta, \gamma = v_0 \upharpoonright \beta, \gamma$, which will allow us to conclude by Lemma 21. The base case, that of u_0 and v_0 being entirely composed of moves in γ, is immediate.

Suppose now that the property is true for all strict prefixes of u_0 and v_0. If the last move of u_0 is visible, the conclusion is immediate. If the last move is in component β, then it is either a Player's move in $u \upharpoonright \alpha, \beta$ or a Player's move in component $u \upharpoonright \beta, \gamma$; the determinacy of s in the first case, and that of t in the second case allow us to conclude. ■

Lemma 23 *Composition of strategies preserves determinacy.*

Proof: let p be an odd-length position. Suppose that $p \cdot m$ in $s;t$; then it is an immediate consequence of Lemma 22 that if $p \cdot n \in s;t$ then $m = n$.

We are now going to prove that $p \notin s;t$. The position $p \cdot m$ arises in $s;t$ from an interaction sequence u and positions $q \in s$ and $q' \in t$. Suppose that m is a move in component 0 (the case of m being in component 1 is analogous).

Suppose first that p arises in $s;t$ from case (1) in Definition 17, and let v be the associated interaction sequence and $r \in s$, $r' \in t$ be the associated positions. Now r is of odd length, and by Lemma 9, the collection s' defined by

$$s' = s \backslash \{r\} \cup \{r \cdot m\}$$

is itself a strategy. Note that $v \cdot m'$, where m' is m with the component 0 replaced with α, is an interaction sequence for $s';t$; and the position that arises from m' is $p \cdot m$. Hence either $v \cdot m'$ is an interaction sequence for $s;t$, which is impossible by the determinacy of s, or else p arises from two distinct interaction sequences in $s';t$, which is impossible by Lemma 22.

Suppose now that p arises in $s;t$ from case (2) in Definition 17, and let u be one of the u_i in that case; let $q \in s$ and $q' \in t$ be positions such that $q \upharpoonright 1 = q'_{-1} \upharpoonright 0$ (the other case is symmetric). Now q' ends in a hidden move m_0; as before, let s' be the strategy

$$s' = s \backslash \{q\} \cup \{q \cdot m_1\}$$

and the composition $s';t$ brings about the same contradiction as above. ∎

We can now show that the $\mathbf{K}(\cdot)$ construct (Definition 13 on page 54) can be interpreted as creating constant mappings.

Lemma 24 *For any strategies s and t, $s;\mathbf{K}(t) = \mathbf{K}(t)$.*

Proof: ($\supseteq$) suppose $p \in \mathbf{K}(t)$; then $p = \mathbf{K}(p')$, where $p' \in t$. This implies that $p \upharpoonright 0 = \epsilon = \epsilon \upharpoonright 1$, and as $\epsilon \in s$, the interaction sequence u such that $u \upharpoonright \alpha, \beta = \epsilon$ and $u \upharpoonright \beta, \gamma = p$ is an interaction sequence for $s; K(t)$.

($\subseteq$) suppose $p \in s; \mathbf{K}(t)$. As all positions in $K(t)$ have no moves in component 0, p arises from case (1) in Definition 17; let u be the associated interaction sequence. As $u \upharpoonright \beta = \epsilon$, $u \upharpoonright \alpha\beta = \epsilon$, and hence $u \upharpoonright \beta, \gamma = p \in K(t)$. ∎

4.3.4 Associativity of composition

The conceptually elegant way of proving associativity of composition would be to introduce a notion $s;t;x$ of *double composition*, and prove that for all strategies s, t, x, $(s;t);x = s;t;x = s;(t;x)$. Following Harmer [Har99, Sec. 3.2.3], however, we take the simpler approach of only defining *double interaction* as a generalisation of Definition 16, and directly prove the equality $(s;t);x = s;(t;x)$.

We start by generalising Definition 15.

Definition 25 *A* double-interacting move *is a pair (m, κ), where m is a (justified) move and κ is one of α, β, γ or δ.*

The notion of projection generalises naturally to sequences of double-interacting moves; in this case, we have projections onto three components which yield sequences of interacting moves, and projections onto two components, which yield sequences of moves.

Definition 26 *An* double-interaction sequence *z is a finite sequence of double-interacting moves such that all moves in z are justified except initial moves in component δ and both $u \upharpoonright \alpha, \beta, \gamma$ and $u \upharpoonright \beta, \gamma, \delta$ are interaction sequences.*

In order to prove that composition is associative, given two matching interaction sequences that arise from the two compositions $(s;t);x$,we will want to reconstruct the double-interaction sequence that conceptually arises from the double-composition $s;t;x$; this is the essence of Lemma 28 below. We first need to show that double-interaction sequences behave in a fairly regular way.

Lemma 27 *In a double-interaction sequence, two moves in respective components β and δ cannot occur without an intervening move in one of components α or γ.*

Proof: as a prefix of an interaction sequence is still an interaction sequence, we are going to show that it is impossible to have an interaction sequence z that ends in two moves in respective components β and δ or in respective components δ and β.

Suppose first that z ends in moves m_1 and m_2 in respective components δ and β. There are two cases to consider.
(i) Suppose first that m_1 is an Opponent's move in $z \upharpoonright \beta, \gamma, \delta$. As $z \upharpoonright \beta, \delta$ is a position, m_2 must be a Player's move in $z \upharpoonright \beta, \delta$, and hence in $z \upharpoonright \beta, \gamma, \delta$. Hence, $|z \upharpoonright \beta, \gamma|$ is even, $|z \upharpoonright \gamma, \delta|$ is odd, and $|z \upharpoonright \beta, \delta|$ is even. But then, $|z \upharpoonright \beta, \delta| + 2|z \upharpoonright \gamma| = |z \upharpoonright \beta, \gamma| + |z \upharpoonright \gamma, \delta|$, hence $|z \upharpoonright \beta, \delta|$ is odd, which contradicts the previous statement.
(ii) Suppose now that m_1 is a Player's move in $z \upharpoonright \beta, \gamma, \delta$. By the same argument as above, m_2 must be an Opponent's move in $z \upharpoonright \beta, \gamma, \delta$, hence m_2 is a Player's move in $z \upharpoonright \alpha, \beta$. Hence, $|z \upharpoonright \alpha, \beta|$ is even, $|z \upharpoonright \beta, \delta|$ is odd, and $|z \upharpoonright \alpha, \delta|$ is even. But then, $|z \upharpoonright \beta, \delta| + 2|z \upharpoonright \alpha| = |z \upharpoonright \alpha, \beta| + |z \upharpoonright \alpha, \delta|$, hence $|z \upharpoonright \beta, \delta|$ is even, which contradicts the previous statement.

Suppose now that z ends in moves m_1 and m_2 in respective components β and δ. Again, there are two cases to consider.
(i) Suppose first that m_1 is an Opponent's move in $z \upharpoonright \beta, \gamma, \delta$. As $z \upharpoonright \beta, \delta$ is a position, m_2 must be a Player's move in $z \upharpoonright \beta, \delta$, and hence in $z \upharpoonright \beta, \gamma, \delta$. Hence, $|z \upharpoonright \beta, \gamma|$ is odd, $|z \upharpoonright \gamma, \delta|$ is even, and $|z \upharpoonright \beta, \delta|$ is even. But then, $|z \upharpoonright \beta, \gamma| + 2|z \upharpoonright \delta| = |z \upharpoonright \beta, \delta| + |z \upharpoonright \gamma, \delta|$, hence $|z \upharpoonright \beta, \gamma|$ is even, which contradicts the previous statement.
(ii) Suppose now that m_1 is a Player's move in $z \upharpoonright \beta, \gamma, \delta$. By the same argument as above, m_2 must be an Opponent's move in $z \upharpoonright \beta, \gamma, \delta$. But then, m_1 is an Opponent's move in $|z \upharpoonright \alpha, \beta|$. Hence $|z \upharpoonright \alpha, \beta|$ is odd, $|z \upharpoonright \beta, \delta|$ is odd, $|z \upharpoonright \alpha, \delta|$ is odd. As $|z \upharpoonright \alpha, \delta| + 2|z \upharpoonright \beta| = |z \upharpoonright \alpha, \beta| + |z \upharpoonright \beta, \delta|$, $|z \upharpoonright \alpha, \delta|$ is even, which contradicts the statement above. ■

Lemma 28 (Zipping) *Let u, v be two interaction sequences such that $u \upharpoonright \alpha, \beta = v \upharpoonright \alpha, \gamma$. Then there exists a unique double-interaction sequence z (the* left zipping *of u and v) such that*

$$z \upharpoonright \alpha, \gamma, \delta = u \textit{ and } z \upharpoonright \alpha, \beta, \gamma = v.$$

Dually, let u, w be two interaction sequences such that $u \upharpoonright \beta, \gamma = v \upharpoonright \alpha, \gamma$. Then there exists a unique double-interaction sequence z (the right zipping *of u and w) such that*

$$z \upharpoonright \alpha, \beta, \delta = u \textit{ and } z \upharpoonright \beta, \gamma, \delta = v.$$

Proof: we only prove the first property, as the proof of the second one is similar.

Let us first prove the existence of z. We proceed by induction on the length of u and v; the base case $u = v = \epsilon$ is immediate.

Suppose that a zipping exists for all respective prefixes u', v' of u and v one of which at least is a strict prefix. There are four cases to consider.

(i) the last move of u is in component γ. Let z' be the zipping of u_{-1} and v, and z be z' augmented with the last move of u renamed to be in component δ. Then z is a zipping of u and v.

(ii) the last move of v is in component β. Let z' be the zipping of u and v_{-1}, and z be z' augmented with the last move of v. Then z is a zipping of u and v.

(iii) the last move of u is in component β, and case (ii) above does not apply. As $u \upharpoonright \alpha, \beta = v \upharpoonright \alpha, \gamma$, the last move in v is in γ, and furthermore $u_{-1} \upharpoonright \alpha, \beta = v_{-1} \upharpoonright \alpha, \gamma$. Let z' be the zipping of u_{-1} and v_{-1}, and let z be z' augmented with the last move of u renamed to be in component γ (or, equivalently, the last move of v). Then z is a zipping of u and v.

(iv) the last move of u is in component α, and case (ii) above does not apply. Then the last move of v is in component α, and furthermore $u_{-1} \upharpoonright \alpha, \beta = v_{-1} \upharpoonright \alpha, \gamma$. Let z' be the zipping of u_{-1} and v_{-1}, and let z be z' augmented with the last move of u (or, equivalently, the last move of v). Then z is a zipping of u and v.

In order to prove unicity of the zipping, we are going to prove that for any double-interaction sequences z and z', if

$$z \upharpoonright \alpha, \beta, \gamma = z' \upharpoonright \alpha, \beta, \gamma, \text{ and } z \upharpoonright \alpha, \gamma, \delta = z' \upharpoonright \alpha, \gamma, \delta$$

then $z = z'$. We proceed by induction on the length of z and z', the base case $z = z' = \epsilon$ being immediate.

Suppose $z \neq \epsilon$, $z' \neq \epsilon$, and the property above is true for all strict prefixes of z and z'. Suppose first that the last move in z is in component α; as $z \upharpoonright \alpha, \beta, \gamma = z' \upharpoonright$

α, β, γ, the last move in z' cannot be in either β or γ. As $z \upharpoonright \alpha, \gamma, \delta = z' \upharpoonright \alpha, \gamma, \delta$, it cannot be in δ either; hence, the last move in z' is in component α. Either of the two properties guarantees that z and z' have the same last move, and the induction hypothesis ensures that $z = z'$.

The case of the last move of z being in component γ is similar. As $z \upharpoonright \alpha, \beta, \gamma = z' \upharpoonright \alpha, \beta, \gamma$, the last move in z' cannot be in either α or β, and as $z \upharpoonright \alpha, \gamma, \delta = z' \upharpoonright \alpha, \gamma, \delta$, it cannot be in δ either. Hence, the last move in z' is in component γ, it is therefore the same as the last move in z, and by the induction hypothesis, $z = z'$.

Suppose now that the last move of z is in component δ; the same argument as above shows that the last move in z' cannot be in either component α or γ. Suppose it is in β. As $z \upharpoonright \alpha, \beta, \gamma = z' \upharpoonright \alpha, \beta, \gamma$, z' must end in a sequence containing a move in δ and no intervening move in α; furthermore, as $z \upharpoonright \alpha, \gamma, \delta = z' \upharpoonright \alpha, \gamma, \delta$, z' must end in a sequence containing a move in δ and no intervening move in α. By Lemma 27, this is impossible, hence z' ends in a move in component δ. Thus, z and z' have the same last move, and hence, by the induction hypothesis, $z = z'$.

The case of the last move in z being in component β is analogous. The same argument as above shows that the last move in z' must be in either β or γ, and Lemma 27 implies that it must be in β. Thus, z and z' have the same last move, and by the induction hypothesis $z = z'$. ■

Lemma 29 *Composition of strategies is associative.*

Proof: Let s, t and x be strategies; we are going to show that $(s; t); x \subseteq s; (t; x)$. Let $p \in (s; t); x$.

Suppose first that p arises from case (1) in Definition 17, and let u be the associated interaction sequence, and $q \in s; t$, $q' \in x$ the associated positions. There are two subcases to consider.

(i) Suppose first that q arises from case (1) in Definition 17, and let v be the associated interaction sequence and $r \in s$, $r' \in t$ be the associated positions. Now $u \upharpoonright \alpha, \beta = q = v \upharpoonright \alpha, \gamma$, hence by Lemma 28 there exists a double-interaction sequence z such that

$$z \upharpoonright \alpha, \beta, \gamma = v \text{ and } z \upharpoonright \alpha, \gamma, \delta = u.$$

Let $w = z \upharpoonright \beta, \gamma, \delta$. As $w \upharpoonright \alpha, \beta = r' \in t$ and $w \upharpoonright \beta, \gamma = q' \in x$, w is an interaction sequence for $t; x$, and hence $w \upharpoonright \alpha, \gamma \in t; x$. Let now $u' = z \upharpoonright \alpha, \beta, \delta$.

As $u' \restriction \beta, \gamma = w \restriction \alpha, \gamma$, u' is an interaction sequence for $s; (t; x)$, and hence $p \in s; (t; x)$.
(ii) Suppose now that q arises from case (2) in Definition 17, and let $(v_i)_{i \in \omega}$ be the associated sequence of interaction sequences. For all i, $u \restriction \alpha, \beta = q = v_i \restriction \alpha, \gamma$, hence by Lemma 28 there exists a double-interaction sequence z_i such that

$$z_i \restriction \alpha, \beta, \gamma = v_i \text{ and } z_i \restriction \alpha, \gamma, \delta = u.$$

Furthermore, the construction in Lemma 28 makes it clear that the z_i form an increasing sequence that is stationary in all components except β.

Let $w_i = z \restriction \beta, \gamma, \delta$. The w_i form an increasing sequence that is stationary in β, γ. Now $w_i \restriction \beta, \gamma \in x$, and $w_i \restriction \alpha, \beta = z \restriction \beta, \gamma \in t$. Hence, for all i, $w_i \restriction \alpha, \gamma \in t; x$.

Let $u'_i = z \restriction \alpha, \beta, \delta$. Then $u'_i \restriction \alpha, \gamma = p$, and it satisfies the hypotheses of case (2) of Definition 17, hence $p \in s; (t; x)$.

Suppose now that p arises from case (2) in Definition 17, and let $(u_i)_{i \in \omega}$ be the associated sequence of interaction sequences. Let $q_i = u_i \restriction \alpha, \beta$, $q'_i = u_i \restriction \beta, \gamma$. By Lemma 19, $q_i \in s; t$ is of even length, and hence arises from case (1) in Definition 17; let v_i be the associated interaction sequence. As $u_i \restriction \alpha, \beta = q_i = v_i \restriction \alpha, \gamma$, by Lemma 28 there exists a double-interaction sequence z_i such that

$$z_i \restriction \alpha, \beta, \gamma = v_i \text{ and } z_i \restriction \alpha, \gamma, \delta = u_i.$$

Now z_i is increasing and is stationary in all components except γ. Let $w_i = z \restriction \beta, \gamma, \delta$. Then $(w_i)_{i \in \omega}$ satisfies the hypotheses of case (2) of Definition 17, and hence $w_i \restriction \alpha, \gamma \in t; x$. Let $u' = z \restriction \alpha, \beta, \delta$. As $u' \restriction \beta, \gamma = w_i \restriction \alpha, \gamma \in t; x$, u' is an interaction sequence for $s; (t; x)$, and hence $p \in s; (t; x)$. ■

4.3.5 Simple strategies

A number of strategies are simple enough to be described directly. The simplest strategy, of course is **top** $= \{\epsilon\}$; a Player that plays according to **top** refuses any (initial) move. Dual¶ to this is the strategy Ω, which contains all positions of length 1 or less and only those. A Player that follows Ω accepts any initial move from Opponent but never replies to it.

Strategies for ground values are defined as follows. The strategy **tt** (resp. **ff**) is defined as consisting of all the even-length positions that can be built using the moves q and $a^{\mathbf{tt}}$ (resp. q and $a^{\mathbf{ff}}$): this corresponds to the fact that a Boolean

¶In a sense that will be made precise in Section 4.4.

with value **tt** can be computed an arbitrary number of times. In other words, **tt** consists of the positions ϵ, $q \cdot a^{\mathbf{tt}}$, $q \cdot a^{\mathbf{tt}} \cdot q \cdot a^{\mathbf{tt}}$, $q \cdot a^{\mathbf{tt}} \cdot q \cdot a^{\mathbf{tt}} \cdot q \cdot a^{\mathbf{tt}}$, etc.

The identity strategy $\mathbf{I}$ consists of all even-length positions of arrow shape that merely copy moves between components 0 and 1. More precisely, we say that a position p of arrow shape is an identity position if it is of even length, and for any even-length prefix q of p, $q \restriction 0 = q \restriction 1$. The strategy $\mathbf{I}$ is then defined as the set of all identity positions.

Lemma 30 *For any strategy of arrow shape s,*

$$\begin{aligned} \mathbf{I}; s &= s; \textit{ and} \\ s; \mathbf{I} &= s. \end{aligned}$$

Proof: we only prove the first equality, as the proof of the second one is very similar.
($\supseteq$) let p be a position in s, and let q be the position consisting of the fair interleaving of $\mathbf{I}_1(p \restriction 0)$ and $\mathbf{I}_0(p \restriction 0)$, *i.e.* if $p \restriction 0$ is of the form

$$p \restriction 0 = m^{(0)} \cdot m^{(1)} \cdots m^{(k)},$$

then q is of the form

$$q = m_1^{(0)} \cdot m_0^{(0)} \cdot m_1^{(1)} \cdot m_0^{(1)} \cdots m_1^{(k)} \cdot m_0^{(k)}.$$

Let u be the interaction sequence such that $u \restriction \alpha, \beta = q$ and $u \restriction \beta, \gamma = p$. Then u is an interaction sequence for $\mathbf{I}; s$ and gives rise to p.
($\subseteq$) let p be a position in $\mathbf{I}; s$. Suppose first that p arises from case (1) in Definition 17; let u be the associated interaction sequence, and q, q' the associated strategies. Because q is an identity position, p a position of arrow shape, and $q \restriction 1 = p \restriction 0$, $p = q'$, hence $p \in s$.

Suppose now that p arises from case (2) in Definition 17. Then for any i, $u_i \restriction \alpha, \beta \in \mathbf{I}$, hence, as the u_i form an unbounded sequence, Lemma 19 implies that there exists a position $q \in I$ ending in three subsequent moves in component 1, which is clearly not the case. ■

Clearly, composition does not have a neutral element over all strategies. However, by Lemma 30 the strategy $\mathbf{I}$) is a (left- and right-) identity for composition with strategies of arrow shape.Putting this together with the associativity of composition (Lemma 29, we get the following property:

Lemma 31 *The set of strategies of arrow shape, equipped with composition, is a category (actually a monoid).*

4.3.6 Products

Before we can introduce pairing of strategies, we need to introduce a strategy — the *diagonal strategy* Δ — which will be the main source of non-linearity in our constructions.

Consider the strategy $\mathbf{I}$ restricted to positions of product shape, *i.e.*

$$\mathbf{I}_\times = \{p \in \mathbf{I} \mid p \restriction 0 \text{ only has moves in } l \text{ and } r\}.$$

and rename all the moves in $\mathbf{I}_\times$ as follows:

$$\begin{aligned} 1 \cdot l &\mapsto 1 \cdot l, \\ 1 \cdot r &\mapsto 1 \cdot l, \\ 0 \cdot l &\mapsto 0, \text{ and} \\ 0 \cdot r &\mapsto 0. \end{aligned}$$

That this induces a strategy of arrow shape Δ is a routine verification.

Given strategies s and t, the pairing (s, t) consists of all positions p such that p is an interleaving of a position $q \in \mathbf{I}_l(s)$ and a position $q' \in \mathbf{I}_r(t)$. In order to define the *tensor product* $s \otimes t$, consider all positions in (s, t) that are entirely within components $l \cdot 1$, $l \cdot 0$, $r \cdot 0$, $r \cdot 1$, (*i.e.* all positions that are at a type $(A \to B) \times (C \to D)$), and rename them as follows:

$$\begin{aligned} l \cdot 1 &\mapsto 1 \cdot l, \\ l \cdot 0 &\mapsto 0 \cdot l, \\ r \cdot 1 &\mapsto 1 \cdot r, \text{ and} \\ r \cdot 0 &\mapsto 0 \cdot r. \end{aligned}$$

Finally, composing this with the strategy Δ, we obtain the functional pairing $\langle s, t\rangle = \Delta; (s \otimes t)$.

The projection strategies π_l and π_r are a simple modification of the identity. For π_l, simply take $\mathbf{I}$ and rename moves as follows:

$$\begin{aligned} 0 &\mapsto 0 \cdot l, \\ 1 &\mapsto 1. \end{aligned}$$

As to π_r, the renaming is

$$\begin{aligned} 0 &\mapsto 0 \cdot r, \\ 1 &\mapsto 1. \end{aligned}$$

Lemma 32 *For any strategies s, t,*

$$\langle s, t\rangle; \pi_l = s,$$

and

$$\langle s, t\rangle; \pi_r = t.$$

Proof: we only prove the first property as the proof of the second one is analogous. Note that $\pi_l \upharpoonright 0$ only contains moves in the l component; whence, for any strategy s,

$$s; \pi_l = s'; \pi_l,$$

where $s' = \{p \in s \mid p \upharpoonright 1 \text{ only has moves in the component } l\}$. In particular,

$$\begin{aligned}\langle s, t\rangle; \pi_l &= \langle s, \mathbf{top}\rangle; \pi_l \\ &= \Delta; s \otimes \mathbf{top}; \pi_l \\ &= \Delta; \mathbf{I}_l(s); \pi_l \\ &= s.\end{aligned}$$

The proof of the individual equalities is a routine verification similar in style to that of Lemma 30. ∎

4.3.7 Conditionals

Finally, we need to define the conditional strategy **ite**. Intuitively, **ite** takes a pair of a Boolean and a pair of values, and returns the first or the second value depending on the value of the Boolean. More precisely,

$$\begin{aligned}\mathbf{K}((\mathbf{tt}, (x, y))); \mathbf{ite} &= \mathbf{K}(x); \text{ and} \\ \mathbf{K}((\mathbf{ff}, (x, y))); \mathbf{ite} &= \mathbf{K}(y).\end{aligned}$$

Thus, **ite** consists of all the interleavings of positions of the form

$$q_1 \cdot q_{0l} \cdot a_{0l}^{\mathbf{tt}} \cdot p_{\mathbf{tt}}$$

and

$$q_1 \cdot q_{0l} \cdot a_{0l}^{\mathbf{ff}} \cdot p_{\mathbf{ff}}$$

where q is the initial question, $a^{\mathbf{tt}}$ and $a^{\mathbf{ff}}$ are the true and false answers respectively (see Section 4.2 on page 48). The position $p_{\mathbf{tt}}$ is an identity position renamed as follows:

$$\begin{aligned}&0 \mapsto 0 \cdot r \cdot l, \\ &1 \mapsto 1.\end{aligned}$$

As to $p_{\mathbf{ff}}$, it is an identity position renamed as follows:

$$0 \mapsto 0 \cdot r \cdot r,$$
$$1 \mapsto 1.$$

4.4 An ordering on strategies

The first step towards defining a sound model is to define an ordering $\preccurlyeq$ on strategies which, up to the interpretation, will be a refinement of the observational order on terms: whenever $M \lesssim N$, we will want to have $[\![M]\!] \preccurlyeq [\![N]\!]$.

The definition of the *liveness ordering* $\preccurlyeq$ is inspired by the *back-and-forth inclusion relation* introduced by Abramsky [Abr97, Prop. 5.2, 5.3]. Thinking of contexts as mappings from moves to moves, we will want strategies s and t to satisfy $s \preccurlyeq t$ if and only if s accepts more moves and produces less moves than t.

The definition is both complicated and simplified by the fact that we will want to use the liveness ordering not only with deterministic collections of positions such as strategies, but also, in the next chapter, with collections of positions that are only prefix-closed. We define $\preccurlyeq$ directly on prefix-closed positions, and deduce a suitable definition for strategies from that.

Definition 33 *Given prefix-closed sets of positions A and B, we say that B is* more live *than A, or A is* safer *than B, and write $A \preccurlyeq B$, if*

- *for every position of odd length $q \in B$, if $q_{-1} \in A$ then $q \in A$; and*
- *for every position of even length $p \in A$ $(p \neq \epsilon)$, if $p_{-1} \in B$, then $p \in B$.*

The definition of $\preccurlyeq$ may be paraphrased as follows. Given a prefix-closed collection of positions A, a position p is said to be *reachable* at A if $p_{-1} \in A$ (or $p = \epsilon$). In order to have $A \preccurlyeq B$, the set of odd-length positions (positions ending in an Opponent's move) in A that are reachable at A needs to be a superset of the set of odd-length positions in B; and, dually, the set of even-length positions in B that are reachable at B should be a superset of the even-length positions in A.

The intuition here is that A is a specification that is stricter that B. Unrolling the induction implicit in the definition, we first require that all positions of length 1 present in B be present in A; in other words, any initial Opponent's move that is specified to be accepted by B is also specified to be accepted by A. Moving on to positions of length 2, whenever Opponent played a move m, any Player's move

n that A allows must also be allowed by B — but only if a Player obeying B might actually have an opportunity to play n, *i.e.* if B allows playing the initial move m. Similarly, for positions of length 3, we require that any move required to be accepted by B is also required to be accepted by A, restricting this condition to Opponent's moves that might occur when playing against a Player obeying A.

In order to prove that $\preccurlyeq$ is indeed a partial order, we first need a property which is the main reason behind both transitivity and antireflexivity.

Lemma 34 *Let A, B and C be prefix-closed collections of positions such that $A \preccurlyeq B \preccurlyeq C$, and let p be a position that is in both A and C; then $p \in B$.*

Proof: by induction on the length of p. Let $A \preccurlyeq B \preccurlyeq C$, and p be a position that is both in A and C; suppose that any strict prefix of p is in B. Then $p_{-1} \in B$. If p has even length, then $p \in A$ and $A \preccurlyeq B$ ensure that $p \in B$; if, on the other hand, p has odd length, then $p \in C$ and $B \preccurlyeq C$ imply that $p \in B$. ∎

Theorem 35 *The relation $\preccurlyeq$ is a partial order on prefix-closed collections of positions.*

Proof: reflexivity is an immediate consequence of the definition. In order to prove transitivity, let A, B and C be prefix-closed sets of positions such that $A \preccurlyeq B \preccurlyeq C$. Let q be an odd-length position in C, and suppose that $q_{-1} \in A$. By Lemma 34, $q_{-1} \in B$, and $B \preccurlyeq C$ implies that $q \in B$. As $A \preccurlyeq B$, $q \in A$.

Conversely, let p be an even-length position in A, and suppose $p_{-1} \in C$. By Lemma 34, $p_{-1} \in B$, and therefore $p \in B$, which in turn implies $p \in C$. This completes the proof of transitivity.

Let now A and B be prefix-closed sets of positions such that $A \preccurlyeq B \preccurlyeq A$. Let p be a position in A such that for all strict prefixes p' of p are in B; then, by Lemma 34, p is in B, which completes the proof of $A \subseteq B$. The proof of $B \subseteq A$ being analogous, this completes the proof of antireflexivity. ∎

The definition of $\preccurlyeq$ above does not yield a transitive or antireflexive relation on arbitrary sets of positions. We may, however, extend $\preccurlyeq$ to arbitrary sets of positions by comparing their prefix closures. We recall that given a set of positions A, we write $\mathrm{Pref}(A)$ for the *prefix closure* of A, *i.e.* the set of all the prefixes of elements of A.

Definition 36 *Given arbitrary collections of positions A and B, write $A \preccurlyeq B$ if* $\mathrm{Pref}(A) \preccurlyeq \mathrm{Pref}(B)$.

While this definition only makes $\preccurlyeq$ into a preorder on arbitrary sets of positions, it does actually make it into a partial order on strategies.

Lemma 37 *If s and t are strategies, then* $\mathrm{Pref}(s) = \mathrm{Pref}(t)$ *implies $s = t$. Therefore, $\preccurlyeq$ is a partial order on strategies.*

Proof: suppose $\mathrm{Pref}(s) = \mathrm{Pref}(t)$, and let $p \in s$. Then there exists a position $q \in t$ such that p is a prefix of q.

If p is of even length, then by even-prefix-closedness of t, $p \in t$. If p is of odd length, then by Lemma 9 on page 52, no position of which p is a strict prefix is in s; hence, as $\mathrm{Pref}\, s = \mathrm{Pref}\, t$, $p \in t$, which completes the proof of $s \subseteq t$. A symmetric argument proves that $t \subseteq s$. ■

Note that this property fundamentally relies on the determinacy condition for strategies. In fact, there is a different characterisation, of the liveness ordering in the particular case of (deterministic) strategies. Intuitively, it expresses the fact that when $s \preccurlyeq t$, and t plays up to a given position, then in the same environment, s either plays up to the same position, or loops beforehand. We will use this property when we prove that composition of strategies is monotone (Lemma 41 on page 71).

Lemma 38 *Let s and t be strategies. Then $s \preccurlyeq t$ if and only if*

- *for any position $q \in \mathrm{Pref}\, t$ of odd length, either $q \in \mathrm{Pref}\, s$ or there exists an odd-length prefix q_0 of q such that $q_0 \in s$;*
- *for every position of even length $p \in s$ ($p \neq \epsilon$), if $p_{-1} \in \mathrm{Pref}\, t$, then $p \in t$.*

Proof: let us first prove that this condition is sufficient. First note that the second part of this lemma is a rephrasing of the second part of Definition 33, so we only need to concern ourselves with odd-length positions. Let s and t be strategies satisfying the condition; we are going to prove that $s \preccurlyeq t$. Suppose that q is an odd-length position in $\mathrm{Pref}\, t$. Then either $q \in \mathrm{Pref}\, s$, in which case we are done, or else there exists an odd-length prefix q_0 of q such that $q_0 \in \mathrm{Pref}\, s$. But then, by Lemma 9 on page 52, $q_{-1} \notin \mathrm{Pref}\, s$, which allows us to conclude.

Let us now prove that this condition is necessary. Let s and t be strategies such that $s \preccurlyeq t$. Let q be an odd-length position in $\mathrm{Pref}\, t$; we are going to prove by induction on the length of q that either $q \in \mathrm{Pref}\, s$ or there exists an odd-length prefix q_0 of q that is in s.

Suppose first that $|q| = 1$; then $q_{-1} = \epsilon \in s$, whence, as $s \preccurlyeq t$, $q \in \mathrm{Pref}\, s$ and we are done. Suppose now that $|q| \geq 3$; if $q \in \mathrm{Pref}\, s$, we are done. If, on the other hand, $q \notin \operatorname{Pref} s$, then, as $s \preccurlyeq t$, $q_{-1} \notin \operatorname{Pref} s$. Clearly, $q_{-2} \in \operatorname{Pref} t$, so by the induction hypothesis, either $q_{-2} \in \operatorname{Pref} s$, hence $q_{-2} \in s$, and we are done, or else there exists an odd-length prefix q_0 of q_{-2} in s; but then, q_0 is also a prefix of q, which proves the property. ∎

We now introduce the prefix-closed sets of positions $\top = \{\epsilon\}$ and $\bot$, which consists of ϵ and all positions of length 1. We show that $\top$ and $\bot$ are, respectively, the largest and smallest element of the set of (non-empty) prefix-closed collections of positions equipped with $\preccurlyeq$[‖].

Lemma 39 *The set of positions $\top$ is the greatest and the set $\bot$ the least element for $\preccurlyeq$.*

Proof: let A be a prefix-closed set of positions. As $\top$ does not contain any odd-length positions, any odd-length position in $\top$ is vacuously in A. Conversely, let p be an even-length position in A; as $\top$ contains no odd-length positions, $p_{-1} \notin \top$, which concludes the proof of $A \preccurlyeq \top$.

Let now p be an even-length position in $\bot$; as $\bot$ doesn't contain any non-empty even-length positions, $p = \epsilon$, and hence $p \in A$. Conversely, let p be an odd-length position in A. If p is of length 1, then $p \in \bot$; if, on the other hand p is of length at least 3, then $p_{-1} \notin \bot$, which concludes the proof of $\bot \preccurlyeq A$. ∎

Before we prove that composition is monotone with respect to the liveness ordering, we will need the following strengthening of Lemma 22 on page 58.

Lemma 40 *Let s, t, s', t' be strategies such that $s \preccurlyeq s'$ and $t \preccurlyeq t'$ (resp. $s' \preccurlyeq s$ and $t' \preccurlyeq t$). Let u be an interaction sequence for s, t and u' and interaction sequence for s', t' such that $u' \upharpoonright \alpha, \gamma$ is a prefix of $u \upharpoonright \alpha, \gamma$. Then u is a prefix of u'.*

Proof: by induction on the length of u. The base case, that of u consisting entirely of moves in γ, is immediate.

Suppose first that u ends in a visible Player's move. Then u is of the form

$$u = u_0 \cdot m \cdot m_1 \cdots m_k \cdot n$$

where m is a visible Opponent's move, the m_i are hidden moves, and n is a visible Player's move. By the induction hypothesis, u_0 is a prefix of u, and hence $u_0 \cdot m$ is a prefix of u. Now a simple inductive argument on i shows that $u_0 \cdot m \cdot m_1 \cdots m_i$

[‖] While $\top$ and $\bot$, taken as sets of positions, are equal to **top** and Ω respectively (Section 4.3.5 on page 63), we prefer to carefully distinguish between the latter, which are strategies, and the former, which are prefix-closed collections of positions, *i.e.* games (Definition 57 on page 86).

is a prefix of u. Suppose indeed that m is in component α (the dual case is analogous). If i is odd, then the fact that $s' \preccurlyeq s$ forces $u_0 \cdot m \cdot m_1 \cdots m_i$ to be a prefix of u; if i is even, then the fact that $t' \preccurlyeq t$ forces the same thing. Finally, the fact that $s' \preccurlyeq s$ if n is in component α, or the fact that $t' \preccurlyeq t$ if n is in component γ forces the fact that u is a prefix of u'.

The case of u ending in an Opponent's move is analogous. ∎

We are now ready to prove that composition (Definition 17 on page 56) is a monotone operation with respect to the liveness ordering.

Lemma 41 *Composition of strategies is a monotone operation w.r.t.* $\preccurlyeq$.

Proof: We only prove monotonicity on the left; monotonicity on the right is analogous. Let s, s' and t be strategies such that $s \preccurlyeq s'$; we are going to prove that for any position p,

- if p is of even length ($p \neq \epsilon$), $p \in s; t$ and $p_{-1} \in \mathrm{Pref}(s'; t)$ then $p \in s'; t$;
- if p is of odd length, $p \in \mathrm{Pref}(s'; t)$ and $p_{-1} \in s; t$ then $p \in \mathrm{Pref}(s; t)$.

Let $p \neq \epsilon$ be a position of even length, and suppose that $p \in s; t$. By Lemmata 18 and 19 on Page 56, p arises from case (1) in Definition 17 on page 56; let u be the associated interaction sequence, and $q \in s$, $q' \in t$ be the associated even-length positions. Suppose that $p_{-1} \in \mathrm{Pref}\, s'; t$; then either $p \in s'; t$, in which case we are done, or else $p_{-1} \in s'; t$. In that case suppose first that p_{-1} arises from case (1) in Definition 17 on page 56, and let u' be the associated interaction sequence, and $r \in s'$, $r' \in t$ be the associated positions. By Lemma 40, u' is a prefix of u; hence, r is a prefix of q and r' is a prefix of q'. By Lemma 9, r' is of even length; hence (by Lemma 18) r is of odd length, which is impossible by Lemma 9.

Suppose now that p_{-1} arises from case (2) in Definition 17, and let (u_i) be the associated sequence of interaction sequences. By Lemma 40, the u_i are prefixes of u, which is impossible as the u_i form an unbounded family.

Let now p be a position of odd length, and suppose that $p \in \mathrm{Pref}\, s'; t$ and $p_{-1} \in s; t$. As p_{-1} is of even length, it derives from case (1) in Definition 17; let u' be the associated interaction sequence, and r, r' the associated positions. Suppose first that $p \notin s'; t$; then there exists an even-length position p' such that $p'_{-1} \in s'; t$; let u be the associated interaction sequence, and q, q' the associated (even-length) positions. By Lemma 40, u' is a prefix of u, and hence r and r' prefixes of q and q' respectively.

Now u has the structure

$$u = u' \cdot m_1 \cdots m_k \cdot m \cdot n_1 \cdots n_l \cdot n$$

where the m_i and the n_i are hidden and m and n are visible. By an inductive argument similar to the one in the proof of Lemma 40, u is an interaction sequence for $s'; t$, $p' \in s'; t$ and hence $p \in \operatorname{Pref} s'; t$.

Suppose now that $p \in s'; t$. Suppose first that p arises from case (1) in Definition 17, and let u be the associated interaction sequence, q and q' the associated positions. By Lemma 40, u' is a prefix of u, and u is of the form

$$u = u' \cdot m_1 \cdots m_k$$

where all the m_i are hidden. An inductive argument similar to the one in the proof of Lemma 40 shows that u is an interaction sequence for $s'; t$.

Suppose now that p arises from case (2) in Definition 17, and let (u_i) be the associated sequence of interaction sequences. Now by Lemma 40, every u_i is of the form

$$u_i = u' \cdot m_1 \cdots m_{k_i}$$

where the m_l are hidden. An inductive argument shows that the u_i are interaction sequences for $s'; t$, hence that $p \in s'; t$. ■

Lemma 42 *Let s, t be two strategies, and suppose that $s \preccurlyeq t$. Then for any strategy u, $s; u = \top$ implies $t; u = \top$.*

Proof: By Lemma 41, $s; u \preccurlyeq t; u$. If $s; u = \top$, then $\top \preccurlyeq t; u$, and therefore, by Lemma 39, $t; u = \top$. ■

4.4.1 Finitude and approximations

In Denotational Semantics, many constructions and proofs are done on elements that are finite in a suitable sense and then extended by continuity to the whole domain. In this section, we show how similar techniques may be used with strategies.

It is customary, at this point of the exposition, to show that the poset under consideration enjoys desirable properties of completeness (directed-completeness, bounded-completeness, *etc.*). In Section 5.2.3, we prove that the set of strategies equipped with the liveness ordering is actually a complete lattice, using however concepts that will not be introduced until then. For now, we just construct *per pedes* all the limits that we need.

The obvious notion of finitude — that of being a finite set of positions — is not suitable, as our bottom strategy consists of an infinite set of positions. The relevant notion is, instead, defined by bounding the length of positions.

Definition 43 *A strategy s is said to have* finite depth *when the set of lengths of positions in s is bounded. The* depth *of a strategy of finite depth, a natural integer, is the length of the longest position it contains.*

Finite depth strategies are not *compact* in the usual sense.

Approximating a strategy s consists in arbitrarily chopping off branches in s at odd-length positions. Operationally, an approximant of s behaves just like s, except that it might loop at any time.

Definition 44 *Given a strategy s, a strategy t is said to be an* approximant *of s if $p \in t$ implies that either $p \in s$ or p is of odd length and there exists some position $p' \in s$ such that p is a prefix of p'.*

As one would expect, a strategy dominates all of its approximants.

Lemma 45 *Let s be a strategy. If t is an approximant of s, then $t \preccurlyeq s$. Therefore, any two approximants of s are compatible.*

Proof: clearly, $\mathrm{Pref}\, t \subseteq \mathrm{Pref}\, s$, so all even-length positions in t, whether reachable or not, are in s. Let now p be an odd-length position in $\mathrm{Pref}\, s$. Either $p \in \mathrm{Pref}\, t$, in which case we are done, or else there is an odd-length position $q \in t$ that is a strict prefix of p; but then, q is also a strict prefix of p_{-1}, which by Lemma 9, implies that $p_{-1} \notin t$. ■

Lemma 46 *Every strategy is the supremum of its finite-depth approximants.*

Proof: this proof uses the characterisation of upper bounds given in Section 5.2.1 (Definition 60 on page 90). Let s be a strategy, and F the set of finite approximants of s. As any approximant of s is less live than s, we only need to prove that $s \preccurlyeq \bigvee F$. We are going to show by induction on the length of a position p that if p is of odd length, $p \in \mathrm{Pref}\, s$ implies $p \in \bigvee F$, and that if p is of even length, $p \in \bigvee F$ implies $p \in \mathrm{Pref}\, s$. The base case, $p = \epsilon$, is immediate.

Let p be an odd-length position in $\mathrm{Pref}\, s$. By the induction hypothesis, $p_{-1} \in \bigvee F$. As p_{-1} is of even length, it is in all the prefix-closures of strategies in F. As p is in some finite-depth approximant of s, there exists an element s' of F such that $p \in \mathrm{Pref}\, s'$, and therefore $p \in \bigvee F$.

Conversely, let now p be an even-length position in $\bigvee F$, and assume that $p_{-1} \in s$. Then there exists an $s' \in F$ such that $p \in \mathrm{Pref}\, s'$. As $\mathrm{Pref}\, s' \subseteq \mathrm{Pref}\, s$, this implies that $p \in s$. ■

The following lemma will be used when we need to prove properties about composition. It says that in order to prove something about the composition

$s; t$, where t is a fixed strategy of finite depth, it is often enough to consider finite-depth strategies s.

We say that a strategy is *constant* if it consists entirely of positions consisting of moves in component 1; equivalently, s is constant if s is of arrow shape and for any strategy t, $t; s = s$.

Lemma 47 *Let s be a constant strategy, and t a strategy of finite depth $d > 0$. Then there exists an approximant s' of s of depth $d' < d$ such that*

$$s; t = s'; t.$$

For any s and t constant strategies, with t of finite depth $d > 0$, there exists an approximant s' of s of depth $d' < d$ such that

$$\langle t, s\rangle; \mathrm{eval} = \langle t, s'\rangle; \mathrm{eval}.$$

Proof: for the first statement, let d'' be the largest odd integer strictly less than d, and s' be the strategy consisting of all positions p such that

- either $|p| \leq d''$, and $p \in s$; or
- $|p| = d''$ and there exists a position $p' \in s$ of length $d'' + 1$ such that p is a prefix of p''.

For the second statement, simply note that for constant strategies s and t,

$$\langle t, s\rangle; \mathrm{eval} = s; (t \restriction 1),$$

that an approximant of a constant strategy is again a constant strategy, and that projection does not increase the depth. ∎

4.5 Interpretation of the untyped calculus

We interpret a pair $\Gamma \vdash M$, where Γ is an (ordered) list of variables, and M a term such that $\mathrm{FV}(M) \subseteq \Gamma$. The interpretation is defined as follows.

$$
\begin{aligned}
[\![x \vdash x]\!] &= \mathbf{I} \\
[\![\Gamma, x \vdash x]\!] &= \pi_r \\
[\![\Gamma, y \vdash x]\!] &= \pi_l; [\![\Gamma \vdash x]\!] \\
[\![\Gamma \vdash \lambda x.M]\!] &= \Lambda([\![\Gamma, x \vdash M]\!]) \\
[\![\Gamma \vdash (M\ N)]\!] &= \langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \rangle; \mathrm{eval} \\
[\![\Gamma \vdash (M, N)]\!] &= \langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \rangle \\
[\![\Gamma \vdash \pi_l(M)]\!] &= [\![\Gamma \vdash M]\!]; \pi_l \\
[\![\Gamma \vdash \pi_r(M)]\!] &= [\![\Gamma \vdash M]\!]; \pi_r \\
[\![\Gamma \vdash \mathbf{tt}]\!] &= \mathbf{K}(\mathbf{tt}) \\
[\![\Gamma \vdash \mathbf{ff}]\!] &= \mathbf{K}(\mathbf{ff}) \\
[\![\Gamma \vdash \mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ N'\ \mathbf{fi}]\!] &= \\
&= \langle [\![\Gamma \vdash M]\!], \langle [\![\Gamma \vdash N]\!], [\![\Gamma \vdash N']\!] \rangle \rangle; \mathbf{ite} \\
[\![\Gamma \vdash \mathbf{top}]\!] &= \mathbf{top}
\end{aligned}
$$

4.5.1 Soundness

In our calculus, reduction does not in general preserve the semantics of a term; rather, reduction at component c preserves the part of the semantics that is significant at component c.

Definition 48 *Given two collections of positions A and B, we say that A and B are* equal at component c, *and write $A =_c B$, when the set of positions in A starting with a move in component c is equal to the analogous set in B. More formally, when*

$$\{p \in A \mid p = m_{c \cdot d} \cdot p'\} = \{q \in A \mid q = n_{c \cdot d'} \cdot q'\}.$$

Clearly $=_c$ is an equvalence relation for any c.

We prove the equational soundness of our calculus using fairly standard techniques.

Lemma 49 (Equational Soundness) *If $M \Downarrow_c N$, then $[\![M]\!] =_{1 \cdot c} [\![N]\!]$.*

The proof of this lemma is delayed (it is on page 77), as we need an the following lemma first.

Lemma 50

$$[\![\Gamma \vdash M[x\backslash N]]\!] = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; [\![\Gamma, x \vdash M]\!],$$

and therefore

$$[\![\Gamma \vdash M[x\backslash N]]\!] = \langle \Lambda([\![\Gamma, x \vdash M]\!]), [\![\Gamma \vdash N]\!]\rangle ; \mathrm{eval}.$$

Proof: note first that the two properties are equivalent, as, indeed, for all strategies s and t of arrow shape, we have

$$\langle \Lambda(s), t\rangle ; \mathrm{eval} = \langle \mathbf{I}, t\rangle ; s.$$

The property is proved by induction on the syntax of M.

- if $M = \mathbf{tt}$,

$$\begin{aligned} &\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; \Lambda(\mathbf{K}(\mathbf{tt})) \\ &\quad = \mathbf{K}(\mathbf{tt}) \\ &\quad = [\![\Gamma \vdash \mathbf{tt}]\!]. \end{aligned}$$

  The cases of $M = \mathbf{ff}$ and $M = \mathbf{top}$ are analogous.

- if $M = x$,

$$\begin{aligned} &\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; [\![\Gamma, x \vdash x]\!] \\ &\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; \pi_r \\ &\quad = [\![\Gamma \vdash N]\!]. \end{aligned}$$

- if $M = y$, $y \neq x$,

$$\begin{aligned} &\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; [\![\Gamma, x \vdash y]\!] \\ &\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; \pi_l ; [\![\Gamma \vdash y]\!] \\ &\quad = [\![\Gamma \vdash y]\!]. \end{aligned}$$

- if $M = \lambda y.P$, and y is not free in N,

$$\begin{aligned} &\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; [\![\Gamma, x \vdash \lambda y.P]\!] \\ &\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle ; \Lambda([\![\Gamma, x, y \vdash P]\!]) \\ &\quad = \Lambda(\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle \otimes \mathbf{I} ; [\![\Gamma, x, y \vdash P]\!]) \\ &\quad = \Lambda(\langle I, \pi_l ; [\![\Gamma \vdash N]\!]\rangle ; [\![\Gamma, y, x \vdash P]\!]) \\ &\quad = \Lambda(\langle I, [\![\Gamma, y \vdash N]\!]\rangle ; [\![\Gamma, y, x \vdash P]\!]) \\ &\quad = \Lambda([\![\Gamma, y \vdash P[x\backslash N]]\!] \\ &\quad = [\![\Gamma \vdash \lambda y.P[x\backslash N]]\!] \end{aligned}$$

- if $M = (P\ Q)$,

$$\begin{aligned}
&\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash (P\ Q)]\!] \\
&\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; \langle [\![\Gamma, x \vdash P]\!], [\![\Gamma, x \vdash Q]\!]\rangle; \mathrm{eval} \\
&\quad = \langle \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash P]\!], \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash Q]\!]\rangle; \mathrm{eval} \\
&\quad = \langle [\![\Gamma \vdash P[x\backslash N]]\!], [\![\Gamma \vdash Q[x\backslash N]]\!]\rangle; \mathrm{eval} \\
&\quad = [\![(P\ Q)[x\backslash N]]\!].
\end{aligned}$$

- if $M = \langle P, Q\rangle$,

$$\begin{aligned}
&\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash \langle P, Q\rangle]\!] \\
&\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; \langle [\![\Gamma, x \vdash P]\!], [\![\Gamma, x \vdash Q]\!]\rangle \\
&\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; \langle [\![\Gamma, x \vdash P]\!], \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash Q]\!]\rangle \\
&\quad = \langle [\![P[x\backslash N]]\!], [\![Q[x\backslash N]]\!]\rangle \\
&\quad = [\![\langle P, Q\rangle[x\backslash N]]\!].
\end{aligned}$$

- if $M = \pi_l(P)$,

$$\begin{aligned}
&\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash \pi_l(P)]\!] \\
&\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash P]\!]; \pi_l \\
&\quad = [\![P[x\backslash N]]\!]; \pi_l \\
&\quad = [\![\pi_l(P)[x\backslash N]]\!].
\end{aligned}$$

  The case of $M = \pi_r(P)$ is analogous.

- if $M = \mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ Q'\ \mathbf{fi}$,

$$\begin{aligned}
&\langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash \mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ Q'\ \mathbf{fi}]\!] \\
&\quad = \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; \langle [\![\Gamma, x \vdash P]\!], \langle [\![\Gamma, x \vdash Q]\!], [\![\Gamma, x \vdash Q']\!]\rangle\rangle; \mathbf{ite} \\
&\quad = \langle \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash P]\!], \\
&\qquad\quad \langle \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash Q]\!], \langle \mathbf{I}, [\![\Gamma \vdash N]\!]\rangle; [\![\Gamma, x \vdash Q']\!]\rangle\rangle; \mathbf{ite} \\
&\quad = \langle [\![\Gamma \vdash P[x\backslash N]]\!], \langle [\![\Gamma \vdash Q[x\backslash N]]\!], [\![\Gamma \vdash Q'[x\backslash N]]\!]\rangle\rangle; \mathbf{ite} \\
&\quad = [\![\mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ Q'\ \mathbf{fi}[x\backslash N]]\!]. \quad \blacksquare
\end{aligned}$$

We are now ready to prove Lemma 49.

Proof of Lemma 49: by induction on the length of the derivation of $M \Downarrow_c N$. We proceed by cases on the last rule in the derivation.

- the base cases

$$\mathbf{tt} \Downarrow_\epsilon \mathbf{tt} \qquad \mathbf{ff} \Downarrow_\epsilon \mathbf{ff}$$

$$\mathbf{tt} \Downarrow_c \mathbf{top} \qquad \mathbf{ff} \Downarrow_c \mathbf{top}$$

$$\lambda x.M \Downarrow_{1\cdot c} \lambda x.M$$

$$\lambda x.M \Downarrow_c \mathbf{top}$$

$$\langle M, N\rangle \Downarrow_{l\cdot c} \langle M, N\rangle \qquad \langle M, N\rangle \Downarrow_{r\cdot c} \langle M, N\rangle$$

$$\langle M, N\rangle \Downarrow_c \mathbf{top}$$

  are immediate.

- suppose $[\![\Gamma \vdash M]\!] =_{1\cdot c} [\![\Gamma \vdash \lambda x.M']\!]$, and $[\![\Gamma \vdash M'[x\backslash N] =_c [\![\Gamma \vdash P]\!]]\!]$. Now,

$$[\![\Gamma \vdash \lambda x.M']\!] = \Lambda([\![\Gamma, x \vdash M']\!])$$

  and by Lemma 50,

$$[\![\Gamma \vdash M'[x\backslash N]]\!] = \langle \Lambda([\![\Gamma, x \vdash M']\!]), [\![\Gamma \vdash N]\!]\rangle; \mathrm{eval}$$

  But then

$$\Lambda([\![\Gamma, x \vdash M']\!]) =_{1\cdot c} [\![\Gamma \vdash M]\!],$$

  whence

$$[\![\Gamma \vdash M'[x\backslash N]]\!] =_c [\![\Gamma \vdash (M\ N)]\!].$$

  As $[\![\Gamma \vdash (M\ N)]\!] =_c [\![\Gamma \vdash P]\!]$,

$$[\![\Gamma \vdash M'[x\backslash N]]\!] =_c [\![P]\!].$$

- suppose $[\![M]\!] =_{l\cdot c} [\![\langle N, P\rangle]\!]$ and $[\![N]\!] =_c [\![N']\!]$. Then

$$[\![\pi_l(\Gamma \vdash M)]\!] = [\![\Gamma \vdash M]\!]; \pi_l = \langle [\![\Gamma \vdash N]\!], [\![\Gamma \vdash P]\!]\rangle; \pi_l = [\![\Gamma = vdash N]\!],$$

  and therefore

$$[\![\Gamma \vdash \pi_l(M)]\!] =_c [\![\Gamma \vdash N']\!].$$

  The case of π_r is analogous.

- suppose $[\![M]\!] =_\epsilon [\![\mathbf{tt}]\!]$ and $[\![N]\!] =_\epsilon [\![N']\!]$. Then $[\![\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P\ \mathbf{fi}]\!] =_\epsilon [\![N']\!]$. The case of $[\![M]\!] =_\epsilon \mathbf{ff}$ is analogous.

4.5.2 Computational Adequacy

The simplest way of proving computational adequacy, Plotkin's syntactic approximation method, is most convenient in the case where recursion and, more generally, any form of unbounded looping, is syntactically controlled. This is not the case of our calculus, which is currently untyped. It will also not be the case of the typed calculus presented in Chapter 6, which has recursive types with no explicit folding and unfolding.

The alternate method consists in constructing a set of *formal approximation relations* between strategies and types. The main difficulty consists in proving the existence of such relations, *i.e.* constructing them. The elegant method of doing so, using Freyd's and Pitts' formalism of admissible relations [Pit95, Pit94], does not directly apply to our case, as our domain of interpretation is not the principal invariant of the Arena construction that we use.

We choose therefore to construct approximation relations "by hand."

Definition 51 *An initial-component indexed family of relations $\lhd_c$ between constant strategies and closed terms is said to be* admissible *when*

- *for all initial components c and closed terms M, $\bot \lhd_c M$;*
- *given a family of compatible strategies $(t_i)_{i \in I}$, if for all $i \in I$, $t_i \lhd_c M$ and s is the supremum of the t_i, then $s \lhd_c M$.*

An admissible family is said to be a family of formal approximation relations *when it satisfies the following conditions:*

- $s \lhd_\epsilon M$ *if*
  - $s =_1 \bot$, *or*
  - $M \Downarrow_\epsilon \mathbf{tt}$ *and* $s \preccurlyeq \mathbf{K}(\mathbf{tt})$, *or*
  - $M \Downarrow_\epsilon \mathbf{ff}$ *and* $s \preccurlyeq \mathbf{K}(\mathbf{ff})$, *or*
  - $M \Downarrow_\epsilon \mathbf{top}$;
- $s \lhd_{l \cdot c} M$ *if*
  - *if* $s =_{1 \cdot l \cdot c} \bot$, *or*
  - $M \Downarrow_{l \cdot c} \langle N, P \rangle$ *and* $s; \pi_l \lhd_{1 \cdot c} N$, *or*
  - $M \Downarrow_{l \cdot c} \mathbf{top}$;
- $s \lhd_{4 \cdot c} M$ *if*

- *if* $s =_{1\cdot r\cdot c} \bot$*, or*
- $M \Downarrow_{r\cdot c} \langle N, P\rangle$ *and* $s;\pi_r \lhd_{1\cdot c} N$*, or*
- $M \Downarrow_{r\cdot c} \mathbf{top}$*;*

- $s \lhd_{1\cdot c} M$ *if*
  - $s =_{1\cdot 1\cdot c} \bot$*, or*
  - $M \Downarrow_{1\cdot 1\cdot c} \lambda x.M'$ *and for any strategy* t *and term* N*, if for all initial components* d*,* $t \lhd_d N$*, then* $\langle s, t\rangle; \mathrm{eval} \lhd_c M'[x\backslash N]$*, or*
  - $M \Downarrow_{1\cdot c} \mathbf{top}$.

Note that this definition is not by induction on the construction of the type A. Indeed, the last clause in the definition defines $\lhd_{\mu X.B[X]}$ in function of $\lhd_{B[\mu X.B[X]]}$. Thus, we need to show the existence of such a family of relations. We do that by constructing them by hand in the finite depth case, and extending them to all strategies by continuity using Lemma 46 on page 73.

Lemma 52 *There exists a family of approximation relations.*

Proof: we construct the relation $s \lhd_c M$, for s of finite depth, by simultaneous induction on the length of c and the depth of s. More precisely, given two pairs (s, c) and (s', c') of a finite-depth strategy and a component, we say that (s, c) is (strictly) beneath (s', c'), and write $(s, c) < (s', c')$, when s has strictly smaller depth than s' or when c is strictly shorter than c'. The set of all such pairs equipped with $<$ is clearly a well-founded strictly ordered set.

We note that all the rules in Definition 51 define $s \lhd_c$ depending only on the definition of $s' \lhd_{c'}$ for pairs (s', c') that are strictly beneath (s, c). Indeed, all but the last rule define $s \lhd_c$ in function of $s' \lhd_{c'}$ where c' is strictly shorter than c. In order to deal with the last rule, write $P(t, N)$ for the condition

$$\text{if for all initial components } d,\ t \lhd_d N, \text{ then } \langle s, t\rangle; \mathrm{eval} \lhd_c M'[x\backslash N]$$

and note that the condition

$$\text{for any strategy } t \text{ and term } N,\ P(t, N)$$

is, by Lemma 47 on page 74, equivalent to

$$\text{for any strategy } t \text{ of depth strictly smaller than } s \text{ and term } N,\ P(t, N)$$

so the rule does indeed only depend on pairs (s', c') strictly beneath (s, c).

The above defines a unique family of approximation relations over all finite-depth strategies. We extend this definition to all strategies by continuity: $s \lhd_c M$ if and only if for all finite-depth approximants t of s, $t \lhd_c M$. Because all the conditions in Definition 51 are closed with respect to $\bigvee$, Lemma 46 on page 46 allows us to conclude that there exists a family of approximation relations. ■

In order to show computational adequacy, we show that every term is approximated by its semantics; in the case of ground terms, this will immediately lead to the result we are seeking.

Lemma 53 *Let M be a term, $\Gamma = (x_1, x_2, \ldots x_n)$ an environment such that* $\mathrm{FV}(M) \subseteq \Gamma$. *Let t_i be strategies and N_i terms such that for all i from 1 to n, for all initial components d, $t_i \lhd_d N_i$. Then for any initial component c,*

$$\langle t_1, \ldots t_n \rangle; [\![\Gamma \vdash M]\!] \lhd_c M[x_i \backslash N_i].$$

Proof: by induction on the structure of M.

- $M = x_j$. By definition,

$$\langle t_1 \ldots t_n \rangle; [\![\Gamma \vdash x_j]\!] = t_j$$

  and as $t_i \lhd_c N_i$ by hypothesis,

$$\langle t_1 \ldots t_n \rangle; [\![\Gamma \vdash x_j]\!] \lhd_c x_j[x_i \backslash N_i].$$

- $M = \lambda x.M'$ (with no loss of generality, we assume that x is not one of the x_i). Either $M \Downarrow_c \mathbf{top}$, in which case we are done, or else $c = 1 \cdot c'$ for some c'. If $\langle t_1, \ldots t_n \rangle; [\![\Gamma \vdash M]\!] =_c \perp$,we are done, otherwise let t, N be such that $t \lhd_{c'} N$. Then

$$\langle t_1, \ldots t_n \rangle; \langle [\![\Gamma \vdash M]\!], t \rangle; \mathrm{eval} = \langle t, t_1 \ldots t_n \rangle; [\![\Gamma \vdash M']\!]$$

$$M'[x \backslash N][x_i \backslash N_i] = M'[x \backslash N, x_i \backslash N_i].$$

  By the induction hypothesis

$$\langle t_1, t_n, t \rangle; [\![\Gamma, x \vdash M']\!] \lhd_{c'} M'[x \backslash N, x_i \backslash N_i]$$

  and therefore

$$\langle [\![\Gamma \vdash M]\!], t \rangle; \mathrm{eval} \lhd_c M'[x \backslash N][x_i \backslash N_i].$$

  As M and $M'[x \backslash N]$ reduce to the same value at c, this yields the conclusion.

- $M = (N\ P)$. By the induction hypothesis,

$$\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash N]\!] \lhd_{1\cdot c} N[x_i \backslash N_i]$$

If $\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash N]\!] =_{1\cdot 1\cdot c} \perp$, then $\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash M]\!] =_{1\cdot c} \perp$, and we are done. If $N[x_i \backslash N_i] \Downarrow_{1\cdot c}$ **top**, then $M[x_i \backslash N_i] \Downarrow_c$ **top**, and we are done. Or else, $N[x_i \backslash N_i] \Downarrow \lambda x.N'$, and for any t and Q such that $t \lhd_c Q$, we have

$$\langle\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash N]\!], t\rangle; \mathrm{eval} \lhd_c N'[x \backslash Q].$$

But then, by the induction hypothesis,

$$\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash P]\!] \lhd_c P[x_i \backslash N_i]$$

whence,

$$\langle t_1, \ldots t_n\rangle; \langle [\![\Gamma \vdash N]\!], [\![\Gamma \vdash P]\!]\rangle; \mathrm{eval} \lhd_c N'[x \backslash (P[x_i \backslash N_i])$$

and as this term and M reduce to the same value w.r.t. c, we are done.

- $M = \langle N, P\rangle$. Either c is not of the form $l \cdot c'$ or $r \cdot c'$, in which case $M \Downarrow_c$ **top**, and we are done. Suppose $c = l \cdot c'$; either $s =_{1\cdot c} \perp$, and we are done. Or else, by the induction hypothesis,

$$\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash N]\!] \lhd_{c'} N[x_i \backslash N_i],$$

but then, $[\![\Gamma \vdash N]\!]; \pi_l = [\![\Gamma \vdash M]\!]$, and so

$$\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash M]\!]; \pi_l \lhd_{c'} N[x_i \backslash N_i],$$

and as $N[x_i \backslash N_i]$ and $\pi_l(M[x_i \backslash N_i])$ reduce to the same value at c',

$$\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash M]\!]; \pi_l \lhd_{c'} \pi_l(M[x_i \backslash N_i]).$$

The case of $c = r \cdot c'$ is analogous.

- $M = \pi_l(M')$. Either $[\![\Gamma \vdash M]\!] =_c \perp$, and the conclusion is immediate. Or else $M \Downarrow$ **top**, and again the conclusion is immediate. The remaining case is that $M'[x_i \backslash N_i] \Downarrow_{l\cdot c} \langle N, P\rangle$. By the induction hypothesis,

$$\langle t_1 \ldots t_n\rangle; [\![\Gamma \vdash M']\!] \lhd_{l\cdot c} M'[x_i \backslash N_i]$$

and therefore

$$[\![\Gamma \vdash M']\!]; \pi_l \lhd_c N$$

i.e.

$$\langle t_1, \ldots t_n\rangle; [\![\Gamma \vdash M]\!] \lhd_c N$$

and as $M[x_i \backslash N_i]$ and N reduce to the same term with respect to c, we are done. The case of $M = \pi_r(M')$ is analogous.

- $M = \textbf{if } N \textbf{ then } P \textbf{ else } P' \textbf{ fi}$. Either $M \Downarrow_c \textbf{top}$, in which case we are done. Otherwise, $c = \epsilon$, and either $[\![\Gamma \vdash M]\!] =_1 \bot$, in which case we are done, or else $N \Downarrow_\epsilon \textbf{tt}$ or $N \Downarrow_\epsilon \textbf{ff}$; assume the former.

  By the induction hypothesis,

  $$\langle t_1, \dots t_n \rangle [\![\Gamma \vdash P]\!] \lhd_\epsilon P[x_i \backslash N_i],$$

  whence

  $$\langle t_1, \dots t_n \rangle [\![\Gamma \vdash M]\!] \lhd_\epsilon P[x_i \backslash N_i],$$

  and as M and P reduce to the same value w.r.t. ϵ, we are done.

- The cases of $M = \textbf{tt}$, $M = \textbf{ff}$ and $M = \textbf{top}$ are immediate.

We are now ready to conclude that our interpretation is computationally adequate.

Corollary 54 (Computational Adequacy) *For any closed term M, if*

$$[\![\vdash M]\!] \neq_1 \bot,$$

then

$$M \Downarrow_\epsilon .$$

Proof: by Lemma 53, $[\![\vdash M]\!] \lhd_\epsilon M$. As $[\![\vdash M]\!] \neq_1 \bot$, there are three cases to consider:

- $M \Downarrow_\epsilon \textbf{tt}$;
- $M \Downarrow_\epsilon \textbf{ff}$; or
- $M \Downarrow_\epsilon \textbf{top}$.

In all cases, $M \Downarrow_\epsilon$. ∎

As usual, equational soundness and computational adequacy, taken together, imply inequational soundness.

Corollary 55 (Inequational Soundness) *For any two terms M and N, if $[\![M]\!] \preccurlyeq [\![N]\!]$ then $M \lesssim N$.*

Proof: we use the characterisation of the observational preorder given in Lemma 5 on page 33. Suppose that $[\![M]\!] \preccurlyeq [\![N]\!]$, and for some context $C[\cdot]$, $C[M] \Downarrow_\epsilon$ **top**. By soundness, this implies that $[\![C[M]]\!] =_1$ **top**. By compositionality of the semantics, $[\![C[M]]\!] \preccurlyeq [\![C[N]]\!]$, whence $[\![C[N]]\!] =_1$ **top**. By computational adequacy this means that $C[N] \Downarrow_\epsilon$. By soundness, we conclude that $C[N] \Downarrow_\epsilon$ **top**. ∎

Inequational soundness establishes a stronger relationship between the syntax and the semantics than equational soundness alone. It allows us to use the semantics to prove facts about the syntax that might be tedious or difficult to establish using the syntax alone. Here, for example, is a trivial proof of a property which we stated in Section 3.1.7.

Corollary 56 *The terms* **top** *and* $\lambda x.\mathbf{top}$ *are observationally equivalent.*

Proof: as $\mathbf{K}(\mathbf{top}) = \mathbf{top}$, we have

$$[\![\mathbf{top}]\!] = [\![\lambda x.\mathbf{top}]\!] = \mathbf{top},$$

which allows us to conclude by inequational soundness. ∎

Chapter 5

Interpreting types: Games

In the preceding chapters, we have put into place all the technology needed for interpreting untyped terms. Following our Type Assignment approach, types will be interpreted independently of terms. In this chapter, we introduce *games*, which are the structures that serve this purpose. Just like strategies, games are represented as sets of positions.

As we saw in Chapter 3, there are three important relations in the syntax: the observational preorder on terms, the typing relation between terms and types, and the subtyping preorder on types. All three of these orderings are interpreted by the liveness ordering. More precisely, for closed terms M and N and closed types A and B, we will have, up to some technical details,

- if $M \lesssim N$, then $[\![\vdash M]\!] \preccurlyeq [\![\vdash N]\!]$;
- if $\vdash M : A$, then $[\![\vdash M]\!] \preccurlyeq \top \to [\![A]\!]$; and
- if $\vdash A \leq B$ then $[\![A]\!] \preccurlyeq [\![B]\!]$.

5.1 Concrete structure of games

Types will therefore be interpreted as games. A game is a set of positions that can be seen as providing a specification that a strategy may or may not satisfy. The specifications expressed are strictly safety specifications, *i.e.* if a given strategy belongs to a game, then so does any safer strategy.

In fact, a game A provides not only a specification for Player but also a specification for Opponent. A strategy s belongs to the game A if its behaviour satisfies the constraints expressed by A, but only as long as Opponent behaves according to A. Indeed, in order to be of a type A, a term need only satisfy the specification corresponding to type A when in a context of type A; its behaviour is otherwise unrestricted (this is sometimes called the *assume-guarantee paradigm*).

Technically, this is expressed by the reachability condition in the definition of the liveness ordering (Section 4.4).

Definition 57 *A* game *is a prefix-closed set of positions. We write $\mathcal{G}$ for the set of games.*

5.1.1 Games and liveness: subtyping

The liveness ordering, as defined in Section 4.4, will also be used to interpret subtyping. This should, by now, be a fairly obvious idea: if $\preccurlyeq$ can be used to model typing of strategies, and is an ordering relation, then it should be also used for subtyping.

More precisely, let A and B be games, and suppose that $A \preccurlyeq B$. The transitivity of $\preccurlyeq$ (Theorem 35) ensures that any strategy that plays according to A also plays according to B; in other words, any strategy following the specification A also follows the specification B.

Theorem 35 ensures that

- $\preccurlyeq$ is a partial order over games;
- if s is strategy, A and B games, and $s \preccurlyeq A \preccurlyeq B$, then $s \preccurlyeq B$.

5.1.2 Simple games

We here define the games that have a direct construction; constructors on games will be defined later.

The top game $\top$ is defined by $\top = \{\epsilon\}$. The bottom game $\perp$ is defined by $\perp = \{q \mid q \text{ is an initial move}\}$; in other words, $\perp$ consists of all positions consisting of a single move.

The game of booleans **Bool** consists of all positions that can be built using only the moves q, $a^{\mathbf{tt}}$ and $a^{\mathbf{ff}}$. More precisely, it consists of the sequences, ϵ, q, $q \cdot a^{\mathbf{tt}}$, $q \cdot a^{\mathbf{ff}}$ and any concatenations thereof.

Given games A and B, the game $A \times B$ is the collection of all positions p that are an interleaving of an arbitrary number of positions in $\mathbf{I}_l(A)$ and an arbitrary number of positions in $\mathbf{I}_r(B)$.

Given games A and B, the function game $A \to B$ is defined as the set of all positions p of arrow shape such that $p \upharpoonright 1 \in B$ and $p \upharpoonright 0 \in A$.

We are now about to prove that composition is type-safe; in order to do that, we first need one technical lemma about interaction sequences.

Lemma 58 *Let A, B and C be games and s, t strategies such that $s \preccurlyeq A \to B$ and $t \preccurlyeq B \to C$. Let u be an interaction sequence for $s;t$. Then $u \restriction \alpha, \gamma \in A \to C$ if and only if $u \restriction \alpha, \beta \in A \to B$ and $u \restriction \beta, \gamma \in B \to C$.*

Proof: we are going to show a stronger property, namely that for any u which is a prefix of an interaction sequence for s and t, $u \restriction \alpha, \gamma \in A \to C$ if and only if $u \restriction \alpha, \beta \in A \to B$ and $u \restriction \beta, \gamma \in B \to C$. The proof proceeds by induction on the length of u. The base case (u of length 0) is immediate.

Write $u = u_0 \cdot m$ and suppose the property true for u_0. If $u_0 \restriction \alpha, \gamma \notin A \to C$, then the property is immediate for u by prefix-closure of all the games considered. Suppose therefore that $u_0 \restriction \alpha, \beta \in A \to B$, $u_0 \restriction \beta, \gamma \in B \to C$, and $u_0 \restriction \alpha, \gamma \in A \to C$.

The case of m being a visible move is clear. Indeed, if m is in component γ, then

$$u \restriction \beta, \gamma \in B \to C \text{ iff } u \restriction \gamma \in C \text{ iff } u \restriction \alpha, \gamma \in A \to C.$$

Similarly, if m is in component α,

$$u \restriction \alpha, \beta \in A \to B \text{ iff } u \restriction \alpha \in A \text{ iff } u \restriction \alpha, \gamma \in A \to C.$$

Suppose therefore that m is in component β. As $u \restriction \alpha, \gamma = u_0 \restriction \alpha, \gamma$, it is immediate that $u \restriction \alpha, \gamma \in A \to C$. Suppose first that m is an Opponent's move in β, γ; then $t \preccurlyeq B \to C$ implies that $u \restriction \beta, \gamma \in B \to C$; hence $u \restriction \beta \in B$, and hence $u \restriction \alpha, \beta \in A \to B$.

The case of m being an Opponent's move in α, β is dual. Indeed, in that case $s \preccurlyeq A \to B$ implies that $u \restriction \alpha, \beta \in A \to B$; hence $u \restriction \beta \in B$, and hence $u \restriction \beta, \gamma \in B \to C$. ∎

Lemma 59 *If $s \preccurlyeq A \to B$ and $t \preccurlyeq B \to C$, where s and t are strategies, and A, B and C are games, then $s;t \preccurlyeq A \to C$.*

Proof: let p be an even-length position in $s;t$, and suppose that $p_{-1} \in A \to C$. As p arises from case (1) in Definition 17, let u be the associated interaction sequence. Now u is of the form

$$u = u_0 \cdot m \cdot m_1 \cdots m_k \cdot n$$

where m is a visible Opponent's move, the m_i are hidden, and n is a visible Player's move.

As $u_0 \cdot m = p_{-1}$, $u_0 \cdot m \restriction \alpha, \gamma \in A \to C$, hence, by Lemma 58, $u_0 \cdot m \restriction \alpha, \beta \in A \to B$ and $u_0 \cdot m \restriction \beta, \gamma \in B \to C$. An inductive argument similar to the one in the proof of Lemma 58 proves that $u \cdot m \upharpoonright \alpha, \beta \in A \to B$ and $u \cdot m \upharpoonright \beta, \gamma \in B \to C$; hence, $u \cdot m \upharpoonright \alpha, \gamma \in A \to C$.

Let now $p = p_0 \cdot m$ be an odd-length position in $A \to C$, and suppose that $p_0 \in s; t$; as p_0 is of even length, it arises from a shortest interaction sequence u; let $q = u \upharpoonright \alpha, \beta$ and $q' = u \upharpoonright \beta, \gamma$.

Suppose that m is in component 0 (the case of m in component 1 is analogous). As $s \preccurlyeq A \to B$, either $q \cdot m \in s$, or else there exists a move m' such that $q \cdot m \cdot m_0 \in s$. In the first case, we are done, as $u \cdot m$ is an interaction sequence for $s; t$ that gives rise to p.

In the second case, we are going to build the longest possible sequence of pairs of positions $(q_i \in s, q'_i \in t)$ such that q is a prefix of every q_i, q' is a prefix of every q'_i, $q_i \upharpoonright \beta$ and $q'_i \upharpoonright \alpha$ are identical up to their last move. We initialise the construction by setting $q_0 = q \cdot m$ and $q'_0 = q'$.

If i is even, then $q_i \upharpoonright 1$ is longer than $q'_i \upharpoonright 0$. Now either there exists a position $q'_{i+1} \in s$ such that q'_i is a prefix of q'_{i+1} and $q_i \upharpoonright 1 = q'_{i+1} \upharpoonright 0$, in which case the construction terminates. Or else there is an even-length position q'_{i+1} such that q'_i is a prefix of q'_{i+1} and $q'_i \upharpoonright 1$ and $q'_{i+1} \upharpoonright 0$ are equal up to the last move of q'_{i+1}, in which case the construction continues.

If i is odd, the situation is dual: $q'_i \upharpoonright 1$ is longer than $q_i \upharpoonright 0$. Now either there exists a position $q_{i+1} \in s$ such that q_i is a prefix of q_{i+1} and $q'_i \upharpoonright 0 = q_{i+1} \upharpoonright 1$, in which case the construction terminates. Or else there is an even-length position q_{i+1} such that q_i is a prefix of q_{i+1} and $q_i \upharpoonright 0$ and $q_{i+1} \upharpoonright 1$ are equal up to the last move of q_{i+1}, in which case the construction continues.

If the construction never terminates, we are in the situation of case (2) Definition 17, and we have shown that $p \in s; t$. If the construction terminates, then we have built an interaction sequence u' such that $p \in \mathrm{Pref}(u \upharpoonright \alpha, \gamma)$. In either case we have shown that $p \in \mathrm{Pref}(s; t)$. ■

5.1.3 Aside: linear types

In a number of previous works on Game Semantics [AJ94, AJM00, McC96], the fundamental type structure is linear, or, to be more accurate, affine. Types are constructed using constructors derived from Linear Logic [Gir87] (see also Danos and DiCosmo [DD92]); an Intuitionistic type structure (similar to the one above) is then constructed out of those types.

This approach would work in our setting. Using, *mutatis mutandis*, the definitions given by McCusker [McC96], we may construct a tensor product of games $\otimes$, a product $\&$, a coproduct $\oplus$, a linear function space $\multimap$, and an "of course" modality "!". However, there is no multiplicative "par." Furthermore, there is no distinction between, 1 and $\top$ and, dually, 0 and $\bot$; this is a consequence of the affine, rather than linear, nature of the type structure.

As to subtyping, $\otimes$, & and $\oplus$ are covariant in both their arguments; $\multimap$ is covariant (resp. contravariant) in its second (resp. first) argument; "!" is covariant.

The main novelty comes from the "!," which satisfies the condition

$$!X \preccurlyeq X$$

which expresses the fact that any term can be considered as linear. This is intuitively obvious: a linear type expresses the constraint that a term will be used at most once, and this constraint can be imposed on an arbitrary term. Note that this is the opposite of what happens in the *Clean* programming language [PvE96], in which a term remains linear until it is first subsumed into a non-linear term (typically just before it is duplicated); thus, in Clean, linearity expresses a property of what has already been done to an object, rather than a constraint on what may be done to a term. Our notion of linearity also appears to be unrelated to the unshared call-by-value objects often used in "update in place" compilers for functional programming languages, and sometimes proposed as a programmer-visible feature [Bak95].

5.2 Quantification

As we saw in Chapter 3, only a few types are typically given in the syntax of a calculus; other types are built using a number of constructors. In order to achieve a compositional semantics, these constructors have to be mirrored by constructions over games. In the previous section, we interpreted the simpler constructors — product and function space. Here, we show how to interpret quantified and recursive types.

In order to do so, we will need to know more about the structure of the space of games. In this section, we further investigate the order-theoretic structure of the partially ordered set $(\mathcal{G}, \preccurlyeq)$, and show that it is a complete lattice; the existence of upper (resp. lower) bounds of arbitrary collections of games will be used to interpret universal (resp. existential) quantification. In Section 5.4, we temporarily disregard the order-theoretic structure and make $\mathcal{G}$ into a metric space; given enough contractive mappings, this will allow us to uniquely solve recursive equations on games, thus yielding an interpretation for recursive types.

5.2.1 The complete lattice of games

The definition of bounds of sets of games follows from the definition of $\preccurlyeq$ (see also Abramsky's definition [Abr97, Prop. 5.3]).

Definition 60 *Given a set of games $\mathcal{A} \subseteq \mathcal{G}$, define $\bigwedge \mathcal{A}$ to be the unique game such that:*

- *a position q of odd length is in $\bigwedge \mathcal{A}$ if and only if q_{-1} is in $\bigwedge \mathcal{A}$ and for some $A \in \mathcal{A}$, $q \in A$;*
- *a position p of even length is in $\bigwedge \mathcal{A}$ if and only if p_{-1} is in $\bigwedge \mathcal{A}$ (or $p = \epsilon$) and for all $A \in \mathcal{A}$ such that $p_{-1} \in A$, $p \in A$.*

Define $\bigvee \mathcal{A}$ dually:

- *a position q of odd length is in $\bigvee \mathcal{A}$ if and only if q_{-1} is in $\bigvee \mathcal{A}$ and for all $A \in \mathcal{A}$ such that $q_{-1} \in A$, $q \in A$;*
- *a position p of even length is in $\bigvee \mathcal{A}$ if and only if p_{-1} is in $\bigvee \mathcal{A}$ (or $p = \epsilon$) and for some $A \in \mathcal{A}$, $p \in A$.*

As with Definition 33, these definitions are inductive, as they define the set of positions of length n as a function of the set of positions of length $n-1$. To misquote Abramsky, if games are "relations extended in time," then $\bigwedge$ and $\bigvee$ are intersection and union extended in time. More precisely, $\bigwedge$ takes the union of odd-numbered (Opponent's) moves and the intersection of even-numbered (Player's) ones ($\bigvee$ does the opposite), but in doing this only considers moves that are reachable in the sense of Section 4.4 (on page 67).

These operations clearly preserve prefix-closedness (due to the reachability condition), so $\bigwedge \mathcal{A}$ and $\bigvee \mathcal{A}$ are games as soon as all elements of $\mathcal{A}$ are. We now need to prove that these operations do indeed produce bounds of games with respect to $\preccurlyeq$.

Theorem 61 *Given a set $\mathcal{A}$ of games, $\bigwedge \mathcal{A}$ (resp. $\bigvee \mathcal{A}$) is the infimum (resp. the supremum) of $\mathcal{A}$.*

Proof: We prove the property for infima; the proof for suprema is dual. Let $A \in \mathcal{A}$, be a game; we are first going to prove that $\bigwedge \mathcal{A} \preccurlyeq A$.

Let $q \in A$ be a position of odd length, and suppose that $q_{-1} \in \bigwedge \mathcal{A}$. As $A \in \mathcal{A}$, and $q \in A$, $q \in \bigwedge \mathcal{A}$.

Conversely, let $p \in \bigwedge \mathcal{A}$ ($p \neq \epsilon$) be a position of even length, and suppose that $p_{-1} \in A$; then $p \in A$, which completes the proof of $\bigwedge \mathcal{A} \preccurlyeq A$.

Suppose now that B is a game such that $\bigwedge \mathcal{A} \preccurlyeq B \preccurlyeq A$ for any $A \in \mathcal{A}$. We are going to prove that $B \preccurlyeq \bigwedge \mathcal{A}$, which, as $\preccurlyeq$ is antireflexive, will guarantee that $B = \bigwedge \mathcal{A}$.

Let p be a position of even length in B ($p \neq \epsilon$), and assume that $p_{-1} \in \bigwedge \mathcal{A}$. For any $A \in \mathcal{A}$ such that $p_{-1} \in A$, the hypotheses $B \preccurlyeq A$ and $p_{-1} \in A$, are enough to ensure that $p \in A$, which completes the proof of $p \in \bigwedge \mathcal{A}$.

Conversely, let q be a position of odd length in $\bigwedge \mathcal{A}$, and assume that $q_{-1} \in B$. As $q \in \bigwedge \mathcal{A}$, there is some $A \in \mathcal{A}$ such that $q \in A$, and as $q_{-1} \in B$ and $B \preccurlyeq A$, $q \in B$, which completes the proof of $B \preccurlyeq \bigwedge \mathcal{A}$. ■

We will use the usual infix notations $A \wedge B$ and $A \vee B$ for binary bounds. Of course, these notations represent operations that are associative and commutative.

We are now going to prove two technical properties that will be used Section 5.3.

Lemma 62 *For any games A, B and C,*

$$(A \wedge B) \vee (A \wedge C) \preccurlyeq A \wedge (B \vee C).$$

Proof: this is actually a general property of lattices, and not a specific property of $\mathcal{G}$. Indeed, it is clear that

$$\begin{aligned} A \wedge B &\preccurlyeq A \wedge (B \vee C), \\ A \wedge C &\preccurlyeq A \wedge (B \vee C), \end{aligned}$$

whence, taking suprema of both sides,

$$(A \wedge B) \vee (A \wedge C) \preccurlyeq A \wedge (B \vee C). \quad \blacksquare$$

The converse inequality, $A \wedge (B \vee C) \preccurlyeq (A \wedge B) \vee (A \wedge C)$, which would make $\mathcal{G}$ into a multiplicative lattice, does not hold in general. Indeed, let

$$\begin{aligned} A &= \{\epsilon, q_l\}, \\ B &= \{\epsilon, q_r\}, \\ C &= \{\epsilon, q_l, q_l \cdot a_l^{\mathbf{tt}}\}. \end{aligned}$$

The situation is depicted in Fig. 5.1: $B \vee C = \top$, whence $A \wedge (B \vee C) = A$, but $A \wedge B = \{\epsilon, q_l, q_r\} \preccurlyeq C$ and $A \wedge C = C$, whence $(A \wedge B) \vee (A \wedge C) = C$.

While games are naturally compared with the liveness ordering, they remain collections of positions; thus, there is another lattice structure on games — the one induced by inclusion. The two structures interact in mysterious and wonderful ways; the next lemma, shows that $\cap$ distributes over both $\vee$ and $\wedge$.

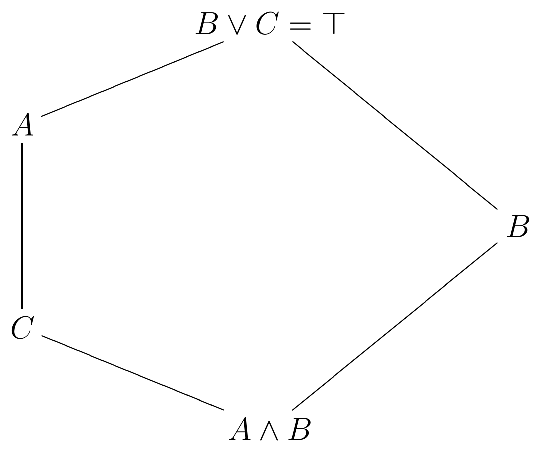


Figure 5.1: The lattice of Games is not multiplicative

Lemma 63 *For any games A, B and C,*

$$A \cap (B \vee C) = (A \cap B) \vee (A \cap C),$$
$$A \cap (B \wedge C) = (A \cap B) \wedge (A \cap C).$$

Proof: as the two properties have very similar proofs, we are only going to show the first one.

First, note that for any games D and E, $D \vee E \subseteq D \cup E$. Whence, $(A \cap B) \vee (A \cap C) \subseteq A$.

We are going to prove by induction on the length of a position p that if $p \in (A \cap B) \vee (A \cap C)$ then $p \in B \vee C$. Let $p \in (A \cap B) \vee (A \cap C)$. If p is of even (resp. odd) length, then $p \in A \cap B$ or $p \in A \cap C$ (resp. $p \in A \cap B$ and $p \in A \cap C$); whence, $p \in B$ or $p \in C$ (resp. $p \in B$ and $p \in C$). By the inductive hypothesis, $p_{-1} \in B \vee C$, and therefore $p \in B \vee C$. This completes the proof of $(A \cap B) \vee (A \cap C) \subseteq B \vee C$.

We are now going to prove by induction on the length of a position p that if $p \in A \cap (B \vee C)$ then $p \in (A \cap B) \vee (A \cap C)$. Let $p \in A$ and $p \in (B \vee C)$. If p is of even (resp. odd) length, then $p \in B$ or $p \in C$ (resp. $p \in B$ and $p \in C$). But as $p \in A$, this means that $p \in A \cap B$ or $p \in A \cap C$ (resp. $p \in A \cap B$ and $p \in A \cap C$); by the induction hypothesis, $p_{-1} \in (A \cap B) \vee (A \cap C)$, and therefore $p \in (A \cap B) \vee (A \cap C)$. This completes the proof of $A \cap (B \vee C) \subseteq (A \cap B) \vee (A \cap C)$.
∎

We will also need the following property, which will allow us to prove the soundness of the quantifier dualisation rules.

Lemma 64 *Let $(A_i)_{i\in I}$ be a collection of games, and B a game. Then,*

$$\bigwedge_{i\in I}(A_i \to B) = \left(\bigvee_{i\in I} A_i\right) \to B$$

and dually,

$$\bigvee_{i\in I}(A_i \to B) = \left(\bigwedge_{i\in I} A_i\right) \to B.$$

5.2.2 Aside: uniformity of quantifiers

In most modern programming languages, quantifiers are defined to be *uniform*; this corresponds to the operational intuition that a polymorphic function has to act on terms with no knowledge of their type. For example, it is not possible in ML [MTHM90, LRVD99] or Haskell [PJH99] to write a function of type $\forall\alpha.\alpha \to \alpha$ that would behave like

$$\lambda n.n + 1$$

when applied to an integer, but like the identity when applied to a value of any other type.

A model of quantification cannot claim to faithfully reflect the language it interprets unless it only includes uniform terms in its interpretation of the universal quantifier. In Chapter 6, we will show how to interpret the type above; up to some technical details, its interpretation will be

$$\bigwedge_{X\in\mathcal{G}} X \to X.$$

Unfortunately, this interpretation is not quite uniform. While it does accurately reject terms such as the one above (due to taking the quantification over all of the space of games, not only over those games that are interpretations of types), it does not reject, say, the term that loops when applied to integers, and behaves like the identity when applied to other values. It may be argued that we do not interpret quantification at all, but have provided a generalisation of union and intersection types instead [Rey88].

The reason for this failure will be made clear in Section 5.3: our model is isomorphic to a "types as ideals" or "inclusive sets" model, and thus we have no way of rejecting a term dominated by a uniform term. This situation is unavoidable in any approach that models types as safety specifications only.

5.2.3 Orthogonality and bounds of strategies

The above definition only provides bounds for games. As $\preccurlyeq$ can also be used to compare strategies, it is a natural question to ask whether we can define bounds for strategies in such a way that the Pref operation would commute with bounds. In general, the answer is no: the supremum of the strategies **tt** and **ff** is **top** and does not correspond to the supremum of their prefix closures taken as games: the supremum of the prefix closures is not deterministic. On the other hand, with infima everything works out.

We start by showing how the poset of strategies can be injected into the poset of games.

Definition 65 *A game A is a* singleton game *if there exists a strategy s such that $A = \mathrm{Pref}(s)$.*

By construction of $\preccurlyeq$ (Definition 36 on page 68) and by Lemma 37, Pref is a monomorphism of posets. This does not imply, of course, that Pref is a morphism of lattices.

Most games are not singletons (they cover too many positions to be made deterministic). It is not difficult to characterise the class of singleton games.

Lemma 66 *A game A is a singleton if and only if for any position $p \in A$ of odd length, there exists at most one position $q \in A$ of length $|p|+1$ such that $p = q_{-1}$.*

Proof: clearly, the prefix closure of a strategy satisfies the condition. Conversely, let A be a game satisfying the condition, and u be the collection of positions defined by the following properties:

- a position p of even length is in u if and only if p is in A;
- a position p of odd length is in u if and only if p is in A and no position $q \neq p$ such that p is a prefix of q is in A.

As A is prefix-closed and the set of even-length positions in u is the set of even-length positions in A, u is even-prefix closed. In order to prove the determinacy of u, suppose that q is an odd-length position, and $p = q \cdot m$ and $p' = q \cdot m'$ are two even-length positions in u; then p and p' are also in A, which contradicts the property.

Let now q be an even-length position, and $p = q \cdot m$ be an odd-length position in u. Then, by the second clause in the construction of u, q is not in u, which completes the proof of the determinacy of u.

The fact that $A = \mathrm{Pref}(u)$ is an immediate consequence of the construction of u. ■

Clearly, removing positions from a game that satisfies the condition in Lemma 66 doesn't violate the condition, whence the following property.

Lemma 67 *Let s be a strategy, and A a game such that $A \subseteq \mathrm{Pref}\, s$. Then there exists a unique strategy u such that $A = \mathrm{Pref}(u)$.*

We are now ready to show how to construct infima of families of strategies.

Lemma 68 *The infimum of a family of singleton games is a singleton game.*

Proof: let $(A_i)_{i \in I}$ be a collection of singleton games, and A their infimum. Let p be an odd-length position, and suppose that $p \cdot m \in A$ and $p \cdot n \in A$; then $p \cdot m \in A_i$ and $p \cdot n \in A_i$ for all $i \in I$. Picking any $j \in I$, Lemma 66 allows us to conclude that $m = n$. ■

This property implies that arbitrary families of strategies have infima and suprema.

Corollary 69 *The poset of strategies is a complete lattice.*

Proof: let $(s_i)_{i \in I}$ be a family of strategies. By Lemma 68, the associated games have an infimum that is a singleton game; let s be the associated strategy. By construction of $\preccurlyeq$, s is beneath all the s_i, and by Lemma 37 on page 69, it is in fact the infimum of the s_i.

The existence of arbitrary infima of strategies allows us to deduce the existence of arbitrary suprema using a standard order theory argument. Indeed, let S be the set of strategies that are above all the s_i; then $\bigwedge S$ is the supremum of the s_i. ■

Note, however, that while the poset of strategies is a sub-poset of $\mathcal{G}$ (up to Pref), it is not a sublattice of the lattice of games. Indeed, consider the strategies $\mathbf{tt}$ and $\mathbf{ff}$; their supremum in the poset of strategies is $\{\epsilon, q\}$, while the supremum of the associated games is $\mathrm{Pref}\, \mathbf{tt} \cup \mathrm{Pref}\, \mathbf{ff}$. On the other hand, as the previous lemmata show, everything works out in the case of infima.

There is one useful case, however, where taking suprema of strategies coincides with the analogous operation on the associated singleton games: it is the case of suprema of chains (completely ordered sets) of strategies.

Lemma 70 *Let $(A_i)_{i \in I}$ be a chain of singleton games. Then $\bigvee_{i \in I} A_i$ is a singleton.*

Proof: simply note that if all the A_i satisfy the condition in lemma 66, so does $\bigcup_{i\in I} A_i$, that $\bigvee_{i\in I} A_i \subseteq \bigcup_{i\in I} A_i$, and conclude by Lemma 67. ∎

Two strategies s and t are orthogonal if their respective sets of initial moves are disjoint; in other words, s and t never do anything interesting in the same context: their interests do not ever coincide. Two games A and B are orthogonal if any strategies $s \preccurlyeq A$ and $t \preccurlyeq B$ are orthogonal.

Definition 71 *Two collections of positions A and B are said to be* orthogonal *if* $\mathrm{Pref}(A) \vee \mathrm{Pref}(B) = \top$*, or, equivalently, if the collections of positions of length 1 in* $\mathrm{Pref}(A)$ *and* $\mathrm{Pref}(B)$ *are disjoint (if A and B are strategies, this means that there is no initial move that both A and B accept). The orthogonal A^o of a game A is defined as the least (w.r.t. $\preccurlyeq$) game that is orthogonal to A, or, equivalently, the set of all positions that do not have a non-empty prefix in* $\mathrm{Pref}(A)$*.*

This operation does not make $\mathcal{G}$ into a Boolean Algebra, as it is not in general the case that $A \wedge A^o = \bot$. Note furthermore that the notion of orthogonality is unrelated to classical negation, modelling which would probably involve exchanging the respective rôles of Player and Opponent [BDER97, AM99].

The supremum operation is rather simple in the case of pairwise orthogonal families (it yields $\top$). The following lemma shows that in the particular case of orthogonal families, the $\bigwedge$ operation may also be given a simpler construction than the one given in Definition 60.

Lemma 72 *Let $(A_i)_{i\in I}$ be a family of pairwise orthogonal prefix-closed collections of positions. Then $\bigwedge_{i\in I} A_i = \bigcup_{i\in I} A_i$.*

Proof: let p be a position in $\bigwedge_{i\in I} A_i$. Whether it is of odd or even length, there exists $i \in I$ such that $p \in A_i$, and therefore $\bigwedge_{i\in I} A_i \subseteq \bigcup_{i\in I} A_i$.

Conversely, let $p \neq \epsilon$ be a position in $\bigcup_{i\in I} A_i$. We are going to prove by induction on the length of p that $p \in \bigwedge_{i\in I} A_i$.

In the base case, $|p| = 1$, $p_{-1} = \epsilon \in \bigwedge_{i\in I} A_i$. As $p \in \bigcup_{i\in I} A_i$, there exists $j \in I$ such that $p \in A_j$, whence $p \in \bigwedge_{i\in I} A_i$.

In the inductive case, $|p| > 1$; by the induction hypothesis, $p_{-1} \in \bigwedge_{i\in I} A$. As above, there exists $j \in I$ such that $p \in A_j$. If p is of odd length, this is enough to guarantee that $p \in \bigwedge_{i\in I} A_i$. If p is of even length, we note that, as A_j is prefix-closed, $p_{-1} \in A_j$; and as the A_i are pairwise orthogonal, there is no $j' \neq j$ such that $p_{-1} \in A_{j'}$. From this we deduce that for any $i \in I$, if $p_{-1} \in A_i$, then $p \in A_i$, which leads to $p \in \bigwedge_{i\in I} A_i$. ∎

5.3 Abstract structure of games

While games are attractive due to their very concrete structure, it is sometimes useful to take a step backwards and look at the abstract structure of games as sets of strategies. In this section, we show how these sets can be seen as being ideals and partial equivalence relations — the very notions used in most models of subtyping [BL90].

Ideals and P.E.R.s are sets of terms that are ordered by inclusion; the intent is that inclusion models subtyping. In the following two sections, we show how to associate both a continuous ideal and a continuous P.E.R. to a game. The mapping between games and continuous ideals is an order isomorphism; on the other hand, the mapping from games into P.E.R.s is only injective. As any P.E.R. may be mapped to an ideal (by taking its range), this leads to the following commutative diagram:

$$\begin{tikzcd} G \arrow[r] \arrow[d] & I \\ \mathbf{PER} \arrow[ur] & \end{tikzcd}$$

5.3.1 Games as ideals

A set of strategies I is a *order ideal* if for any strategy s in I, any strategy t such that $t \preccurlyeq s$ is also in I (I is downwards closed), and whenever two strategies s and t are in I, and the games $\mathrm{Pref}(s)$ and $\mathrm{Pref}(t)$ have a supremum which is a singleton $\mathrm{Pref}(s \vee t)$, then $s \vee t$ is also in I (note that this is *not* the standard notion of order ideal). If, furthermore, for any chain of strategies $(s_i)_{i \in J}$ such that for all i, $s_i \in I$, the supremum of $(s_i)_{i \in J}$ is in I, then I is said to be a *continuous ideal*.

Lemma 73 *The map $[\cdot]$ from games to ideals defined by*

$$[A] = \{s \mid s \preccurlyeq A\}$$

is an order isomorphism between $(\mathcal{G}, \preccurlyeq)$ and the set of ideals of strategies ordered by inclusion.

Proof: It is immediate that $[A]$ is an order ideal and that $[\cdot]$ is a monotone map. In order to show that $[A]$ is continuous, let $(s_i)_{i \in J}$ be a chain of strategies such that for any $i \in J$, $s_i \preccurlyeq A$. By Lemma 70, the supremum of $(s_i)_{i \in J}$ is also dominated by A.

Let us now prove that to every continuous ideal I can be associated a game A such that $I = [A]$. Let I be an ideal of strategies, and let A be the supremum of I,

$$A = \bigvee \{\mathrm{Pref}(s) \mid s \in I\}.$$

We need to show that $[A] = I$.

If $s \in I$, then, by Definition 36, $s \preccurlyeq A$, whence $I \subseteq [A]$. Conversely, let $s \in [A]$; as $s \preccurlyeq A$,

$$s = s \wedge A = s \wedge \bigvee_{t \in I} t.$$

But then, By Lemma 62 on page 91,

$$s \wedge \bigvee_{t \in I} t \preccurlyeq \bigvee_{t \in I} s \wedge t,$$

and therefore, by continuity of I, $s \in I$, which completes the proof. ■

5.3.2 Games as partial equivalence relations

A relation R over a partially ordered set is a *partial equivalence relation* (P.E.R.) if it is symmetric and transitive (but not, in general, reflexive). Such a relation is *continuous* if for any two families $(s_I)_{i \in I}$ and $(t_i)_{i \in I}$ of strategies indexed over the same set I, having a supremum that is a strategy, and such that for all $i \in I$, $s_i R t_i$, we have $\bigvee_{i \in I} s_i R \bigvee_{i \in I} t_i$. It is *uniform* if for any strategies s, t and u, sRt implies $(s \wedge u) R (t \wedge u)$. A partial equivalence relation that is both continuous and uniform is a *continuous uniform partial equivalence relation* — a C.U.P.E.R. for short.

Given a game A and a strategy s, $s \cap A$ is the set of all positions that Player playing according to s can play when playing against Opponent obeying the constraints in A; it is the set of all positions in s that can actually "happen" at type A. Two strategies s and t are *equivalent at the game A* if $s \cap A = t \cap A$. Thus, every game induces an equivalence on strategies; by restricting ourselves to strategies that obey A, we get a P.E.R.

Lemma 74 *The map $\langle \cdot \rangle$ from games to P.E.R.s defined by*

$$\langle A \rangle = \{(s,t) \mid s \preccurlyeq A, t \preccurlyeq A, s \cap A = t \cap A\}$$

is an order monomorphism from $(\mathcal{G}, \preccurlyeq)$ into the set of C.U.P.E.R.s ordered by inclusion.

Proof: there is no doubt that $\langle A\rangle$ is a P.E.R. In order to prove that it is continuous, let $(s_i)_{i\in I}$ and $(t_i)_{i\in I}$ be two families of strategies having respective upper bounds s and t. Assume that for any $i \in I$, $s_i\langle A\rangle t_i$. Then for all i, $s_i \preccurlyeq A$, whence $s \preccurlyeq A$, and similarly for t. Furthermore, for all i, $s_i\langle A\rangle t_i$, *i.e.* $s_i \cap A = t_i \cap A$; whence,

$$\bigvee_{i\in I}(s_i \cap A) = \bigvee_{i\in I}(t_i \cap A),$$

and therefore, by Lemma 63

$$(\bigvee_{i_i n I} s_i) \cap A = (\bigvee_{i\in I} t_i) \cap A,$$

By definition, this means that $s\langle A\rangle t$, which completes the proof of the continuity of $\langle A\rangle$.

To prove that $\langle A\rangle$ is uniform, let s, t and u be strategies such that $s\langle A\rangle t$. Then $s \preccurlyeq A$ and $t \preccurlyeq A$, which implies that $s \wedge u \preccurlyeq A$ and $s \wedge u \preccurlyeq A$. Furthermore, $s\cap A = t\cap A$, which, applying Lemma 63 again, implies that $(s\wedge u)\cap A = (t\wedge u)\cap A$ and completes the proof of the uniformity of $\langle A\rangle$. ∎

Of course, $\langle\cdot\rangle$ is not surjective (onto). Given a P.E.R. R, we say that it *defines* the supremum of its range in the lattice of games. Clearly, this operation is a left inverse of $\langle\cdot\rangle$, but not a right inverse thereof. Consider indeed the discrete P.E.R. $\{(s, s) \mid s\}$ and the antidiscrete P.E.R. $\langle\top\rangle = \{(s, t) \mid s, t\}$. These are distinct C.U.P.E.R.s that both define the game $\top$.

5.4 Recursive games

To interpret recursive types, we use the familiar Banach theorem on metric spaces to show that enough mappings on games have a unique fixpoint.

As metric topology is possibly less familiar in Computer Science than order theory, we start by providing (often without proof) some elementary information about metric spaces.

Most of the material in the following sections can be found in any standard textbook on topology, such as Bourbaki [Bou74] or Engelking [Eng77, Ch. 4]; a gentler introduction is given by Kuratowski [Kur66, Part 2]. We show how many of the notions considered can be related to general topology; however, this section is entirely self contained, and an understanding of general topology is not necessary.

5.4.1 Metric spaces and metric topology

A metric is a formalisation of the intuitive notion of points being more or less far away from each other.

Definition 75 *Given a set X, a* metric *over X is a real-valued, positive two-argument function* d *over X satisfying the following conditions:*

(m1) $\mathrm{d}(x, y) = 0$ *if and only if* $x = y$.

(m2) $\mathrm{d}(x, y) = \mathrm{d}(y, x)$ *for all* $x, y \in X$.

(m3) $\mathrm{d}(x, z) \leq \mathrm{d}(x, y) + \mathrm{d}(y, z)$ *for all* $x, y, z \in X$.

If, furthermore, d *satisfies the condition*

(m3') $\mathrm{d}(x, z) \leq \max(\mathrm{d}(x, y), \mathrm{d}(y, z))$ *for all* $x, y, z \in X$,

then d *is said to be an* ultrametric *over X (and (X, d) is said to be an* ultrametric *space.*

A pair (X, d), where X is a set and d a metric (resp. an ultrametric) over X is called a *metric space* (resp. an *ultrametric space*), and elements of X are often referred to as *points*. Familiar notions of distance appear to be metrics.

Example 76 *The* standard metric on the reals *is defined by* $\mathrm{d}(x, y) = |x - y|$ *for arbitrary reals x and y. By restricting this metric to subsets of the reals, we define the standard metric on the rationals, the standard metric on the natural integers etc.*

Any set can be equipped with a metric, such that the distance between any two distinct elements is a non-zero constant.

Example 77 *The* discrete metric *on a set X is defined, for any points $x, y \in X$, by* $\mathrm{d}(x, y) = 1$ *if* $x \neq y$, $\mathrm{d}(x, x) = 0$.

As we will see in Section 5.4.2, metrics are interesting objects *per se*; however, they are often used as a handy tool for describing a class of topological spaces (known as *metrisable* topological spaces). Every metric defines a topology, constructed as follows. For any point x and positive real δ, *the open ball* of centre x and radius δ is defined as the set

$$\mathrm{B}(x, \delta) = \{y \in X \mid \mathrm{d}(x, y) < \delta\}.$$

The *topology induced by the metric* d is the topology over X defined by the basis

$$\{\mathrm{B}(x, \epsilon) \mid x \in X, \epsilon > 0\},$$

or, equivalently, by the locally countable basis*

$$\{\mathrm{B}(x, \frac{1}{n}) \mid x \in X, n \in \omega, n > 0\}.$$

It follows immediately from axiom (m1) that this topology is separated (Hausdorff, or T_2).

More directly, a subset $\mathcal{O} \subseteq X$ is open in the topology induced by d if and only if for any $x \in \mathcal{O}$ there exists a real number $\epsilon > 0$ such that $\mathrm{B}(x, \epsilon) \subseteq \mathcal{O}$. A set $\mathcal{F} \subseteq X$ is closed if its complementary $X \backslash \mathcal{F}$ is open.

The topological notions of continuity and convergence can be described directly in metric terms as follows. Given two metric spaces (X, d_X), (Y, d_Y), a mapping $f : X \to Y$ is said to be *continuous* when the distance between the images of two points can be controlled as soon as the distance between the points themselves can be; precisely,

$$\forall \epsilon > 0 \ \exists \delta > 0 \ \forall x, x' \in X \ \mathrm{d}_X(x, x') < \delta \Rightarrow \mathrm{d}_Y(f(x), f(x')) < \epsilon.$$

A sequence $(x_n)_{n\in\omega}$ of points converges to a point x if the real sequence $(\mathrm{d}(x_n, x))_{n\in\omega}$ converges to 0, *i.e.*

$$\forall \epsilon > 0 \ \exists N \in \omega \ \forall n \geq N \ \mathrm{d}(x_n, x) < \epsilon.$$

The following theorem provides a useful criterion of continuity of functions between metric spaces. Note that it is specific to metrisable spaces, and does not generalise to arbitrary topological spaces.

Theorem 78 *A map $f : X \to Y$ between two metrisable spaces is continuous if and only if it is* sequentially continuous, i.e. *if for any sequence $(x_n)_{n\in\omega}$ of points of X, $x_n \to_{n\to\infty} x$ implies that $f(x_n) \to_{n\to\infty} f(x)$.*

5.4.2 Non-topological properties of metric spaces

Many distinct metrics can induce the same topology over a given a set. For example, many interesting metrics can induce the discrete topology.

*This is, by the way, a characteristic feature of metrisable spaces: a topological space is metrisable if and only if it is a Hausdorff space and can be defined by a locally countable basis. As we will see later, the metric describing a given topology is not in general unique.

Example 79 *Consider the metric on the natural integers defined by*

$$\mathrm{d}(x, y) = \left| \frac{1}{x+1} - \frac{1}{y+1} \right|$$

This metric, the discrete metric, and the usual metric over the natural integers all induce the discrete topology over the natural integers.

From this, we may conclude that there exist non-topological properties of metric spaces (properties that are not invariant by homeomorphism).

5.4.2.1 Completeness: Cauchy sequences, Lipschitz mappings, and fixpoint equations

A very important notion of metric topology is that of *Cauchy sequence*[†]. Intuitively, a Cauchy sequence is a sequence the points of which become arbitrarily localised; in other words, it is a potentially convergent sequence.

Definition 80 *A sequence of points* $(x_n)_{n\in\omega}$ *in a metric space* (X, d_X) *is said to be a* Cauchy sequence *when for any* $\epsilon > 0$*, there exists an integer* N *such that for any* $n > N$ *and* $k > 0$*,* $d(x_n, x_{n+k}) < \epsilon$*.*

This notion is not topological: metrics defining the same topology may, in general, define different classes of Cauchy sequences.

Example 81 *Let* $(a_n)_{n\in\omega}$ *be the sequence of integers defined by* $a_n = n$*. This is a Cauchy sequence for the metric* d *of Example 79, but a Cauchy sequence neither for the standard metric on the integers nor the discrete metric on the integers (for either of which the Cauchy sequences are exactly the stationary sequences).*

Clearly, any convergent sequence is Cauchy, but the converse property is not true in general.

Example 82 *Let* $\mathbf{Q}$ *be the space of rational numbers equipped with the usual metric on the rationals, and let* $(a_n)_{n\in\omega}$ *be the sequence in* $\mathbf{Q}$ *defined by*

- $a_0 = 1$;
- $a_{n+1} = 1 + \frac{1}{1+a_n}$.

It is easily verified that $(a_n)_{n\in\omega}$ *is a Cauchy sequence (for the usual metric on the rationals). However, as the map* $x \mapsto 1 + \frac{1}{1+x}$ *is continuous, any limit* l *of* $(a_n)_{n\in\omega}$ *must satisfy* $l = 1 + \frac{1}{1+l}$*, which has no solution in* $\mathbf{Q}$*; therefore,* $(a_n)_{n\in\omega}$ *has no limit in* $\mathbf{Q}$*.*

[†] Or *Bolzano-Cauchy sequence.*

Intuitively, the sequence in this example fails to converge because it ends up in a "hole" of **Q**. This situation would be impossible to reproduce in **R** which does not have any holes; **R** is said to be *complete*.

Definition 83 *A metric space in which every Cauchy sequence converges is a* complete *metric space.*

A mapping f between metric spaces (X, d_X) and (Y, d_Y) is said to be a *Lipschitz* mapping if there exists a real constant $\lambda > 0$ such that $\mathrm{d}(f(x), f(x')) \leq \lambda \mathrm{d}(x, x')$; λ is then said to be a *Lipschitz constant* of f. A Lipschitz mapping is continuous, but the converse is not true in general (for example, the map $x \mapsto x^2$ over the reals equipped with the usual metric is continuous but not Lipschitz). A mapping with a Lipschitz constant of 1 is said to be *nonexpanding*, and a mapping with a Lipschitz constant $\lambda < 1$ is said to be *contractive*.

Contractive mappings are closely related to Cauchy sequences. Let f be a contractive mapping over a metric space (X, d), and let $x \in X$. Then the sequence $(x_n)_{n\in\omega}$ inductively defined by

- $x_0 = x$, and
- $x_{n+1} = f(x_n)$ for any $n \in \omega$

is a Cauchy sequence. This leads to Banach's theorem, which gives a very useful criterion for both the existence and unicity of a fixed point of a mapping.

Theorem 84 (Banach) *A contractive mapping from a non-empty complete metric space into itself has a unique fixpoint.*

Proof: let (X, d_X) be a non-empty complete metric space and $f : X \to X$ a contractive mapping over X. Choose a point $x \in X$ and define the sequence $(x_n)_{n\in\omega}$ as above by

- $x_0 = x$, and
- $x_{n+1} = f(x_n)$ for any $n \in \omega$.

The contractivity of f implies that $(a_n)_{n\in\omega}$ is a Cauchy sequence, and therefore has a limit l; the continuity of f implies that l is a fixpoint of f.

Let l and l' be two fixpoints of f. As f is contractive,

$$\mathrm{d}_X(l, l') = \mathrm{d}_X(f(l), f(l')) \leq \lambda \mathrm{d}_X(l, l'),$$

where $\lambda < 1$. This implies $\mathrm{d}_X(l, l') = 0$, whence $l = l'$. ∎

Given a contractive map f, we write $\operatorname{fix} f$ or $\operatorname{fix} x.f(x)$ for its unique fixpoint.

5.4.2.2 Products of ultrametric spaces and simultaneous recursion

Banach's theorem also allows us to solve simultaneous recursive equations by using a product space.

We first need to choose a notion of product of ultrametric spaces[‡]. Our choice is guided by the need to preserve contractivity when pairing mappings, and to make diagonal maps non-expanding.

Given metric spaces (X, d_X) and (Y, d_Y), we define the product space $X \times Y$ as

$$(X \times Y, \mathrm{d}_\infty),$$

where $X \times Y$ denotes the cartesian product of sets and

$$\mathrm{d}_\infty((x, y), (x', y')) = \max(d_X(x, x'), d_Y(y, y')).$$

Note that if X and Y are ultrametric spaces, then so is $X \times Y$.

If we restrict ourselves to uniformly bounded families of metric spaces, this definition generalises to infinite products of spaces. Let $(X_i)_{i \in I}$ be a uniformly bounded family of metric spaces, say with bound 1. Assume that $\mathbf{x} = (x_i)_{i \in I}$ and $\mathbf{y} = (y_i)_{i \in I}$ are points in the carrier $\mathbf{X} = \prod_{i \in I} X_i$, and define

$$\mathrm{d}_\infty(\mathbf{x}, \mathbf{y}) = \sup_{i \in I} \mathrm{d}_{X_i}(x_i, y_i)$$

Then $(\mathbf{X}, \mathrm{d}_\infty)$ is a metric space with the same bound 1. Furthermore, if the X_i are ultrametric spaces, then so is $\mathbf{X}$.

Contractivity of maps from product spaces is equivalent to contractivity of all the components. In order to formalise this, we need a notion of currying, and therefore a metric on maps. The simple *uniform metric* turns out to be sufficient for our needs.

Definition 85 *Let X be a metric space and Y a bounded metric space. Define the* uniform metric *on $X \to Y$ as follows. For maps $f, g : X \to Y$, let*

$$\mathrm{d}_{X \to Y}(f, g) = \sup_{x \in X} \mathrm{d}(f(x), g(x)).$$

[‡]The fact that there is a choice stems from the lack of a standard category of metric spaces. More precisely, there is no "best" choice for the definition of arrows in such a category; depending on the application, one may want to use the category of metric spaces and continuous mappings (equivalent to a full subcategory of the category of topological spaces), the category of metric spaces and uniformly continuous mappings (equivalent to a subcategory of the category of uniform spaces), the category of metric spaces and Lipschitz mappings, the category of metric spaces and pairs (f, λ), where f is a Lipschitz mapping and λ a Lipschitz constant of f, *etc.* More often than not, we implicitly manipulate more than one of these categories at the same time

The set of maps from X to Y equipped with $d_{X \to Y}$ is a metric space, and is an ultrametric space as soon as Y is.

Definition 86 *Given metric spaces X, Y and Z, and a nonexpanding map $f : X \times Y \to Z$, the* currying *of f is the map $\Lambda(f)$ from X to $Y \to Z$ defined by*

$$\Lambda(f)(x)(y) = f(x, y)$$

This definition generalises to infinite products as follows:

Definition 87 *Given an I-indexed family of metric spaces $\mathbf{X}$, a metric space Z and a nonexpanding map $\mathbf{X} \to Z$, the* currying *of f w.r.t. $j \in I$ is the map from X_j to $\mathbf{X} \to Z$ defined by*

$$\Lambda_j(f)(x)(\mathbf{x})) = f(\mathbf{x}')$$

where $\mathbf{x}' = (x'_i)_{i \in I}$, with $x'_j = x$ and $x'_i = x_i$ for $i \neq j$.

The simultaneous currying *of f w.r.t. $j_1, j_2 \in I$ is the map from $X_{j_1} \times X_{j_2}$ to $\mathbf{X} \to Z$ defined by*

$$\Lambda_{j_1, j_2}(f)(x_1, x_2)(\mathbf{x}) = f(\mathbf{x}')$$

where $\mathbf{x}' = (x'_i)_{i \in I}$, with $x'_{j_1} = x_1$, $x'_{j_2} = x_2$, and $x'_i = x_i$ for $i \notin \{j_1, j_2\}$.

Lemma 88 *Let (X, d), (Y, d) and (Z, d) be ultrametric spaces, and $f : X \times Y \to Z$. Then f is Lipschitz (resp. nonexpanding, resp. contractive) over $X \times Y$ if and only if it is Lipschitz (resp. nonexpanding, resp. contractive) in each of its arguments,* i.e. *$f(\cdot, y_0)$ and $f(x_0, \cdot)$ are Lipschitz (resp. nonexpanding, resp. contractive) for any $x_0 \in X$, $y_0 \in Y$.*

Proof: Suppose f is Lipschitz with constant λ over $X \times Y$, and fix $y_0 \in Y$. As

$$\mathrm{d}(f(x_1, y_0), f(x_2, y_0)) \leq \lambda \max(\mathrm{d}(x_1, x_2), \mathrm{d}(y_0, y_0)) = \lambda \mathrm{d}(x_1, x_2),$$

f is Lipschitz in its first argument, and similarly for the second.

Conversely, suppose f is Lipschitz in both its arguments, with respective constants λ_1, λ_2. Let $\lambda = \max(\lambda_1, \lambda_2)$; we then have

$$\begin{aligned} \mathrm{d}(f(x_1, y_1), f(x_2, y_2)) &\leq \max(\mathrm{d}(f(x_1, y_1), f(x_2, y_1)), \mathrm{d}(f(x_2, y_1), f(x_2, y_2))) \\ &\leq \max(\lambda_1 \mathrm{d}(x_1, x_2), \lambda_2 \mathrm{d}(y_1, y_2)) \\ &\leq \lambda \mathrm{d}((x_1, y_1), (x_2, y_2)) \quad \blacksquare \end{aligned}$$

This lemma generalises to infinite products as follows:

Lemma 89 *Let (X) be an I-indexed family of ultrametric spaces, and Y an ultrametric space. Let $f : \mathbf{X} \to Y$ be a mapping. Then f is a Lispchitz mapping for some constant $\lambda \in \mathbf{R}$ if and only if $\Lambda_i(f)$ is Lipschitz with constant λ for all $i \in I$.*

Let now (X, d_X) be a complete ultrametric space, and $f : X \times X \to X$ be a contractive map. For any fixed $y \in X$, $f(x, y)$ has a unique fixpoint $g(y)$; for any fixed $x \in X$, $f(x, y)$ has a unique fixpoint $h(x)$. We have

$$\mathrm{d}_X(g(y), g(y')) = \mathrm{d}(f(g(y), y), f(g(y'), y')) \leq \lambda \max(\mathrm{d}_X(g(y), g(y')), \mathrm{d}_X(y, y')).$$

As $\lambda < 1$, this implies that

$$\mathrm{d}_X(g(y), g(y')) \leq \lambda \mathrm{d}_X(y, y')$$

which implies that g has a unique fixpoint y_0. Similarly, let x_0 be the unique fixpoint of h.

Let now $\hat{f} : X \times X \to X \times X$ be defined by

$$\hat{f}(x, y) = (f(x, y), f(x, y)).$$

The map $\hat{f}$ is contractive over $X \times X$, and therefore has a unique fixpoint (x_1, y_1). As $\hat{f}(x_1, y_1) = (x_1, y_1)$, $x_1 = y_1$, and furthermore

$$\hat{f}(y_0, y_0) = \hat{f}(g(y_0), y_0) = (y_0, y_0)$$

and

$$\hat{f}(x_0, x_0) = \hat{f}(x_0, h(x_0)) = (x_0, x_0)$$

and therefore, by unicity of the fixpoint of $\hat{f}$,

$$x_0 = y_0 = x_1 = y_1.$$

The version on infinite products reads as follows:

Lemma 90 *Let $f : \mathbf{X} \to Y$ be a Lipschitz map with constant λ, $j_1, j_2 \in I$, and suppose that $\Lambda_{j_1}(f)$ and $\Lambda_{j_2}(f)$ are contractive. Then $\Lambda_{j_1,j_2}(f)$ is contractive, and*

$$\operatorname{fix} \Lambda_{j_1}(\operatorname{fix} \Lambda_{j_2}(f)) = \operatorname{fix} \Lambda_{j_2}(\operatorname{fix} \Lambda_{j_1}(f)) = \operatorname{fix} \Lambda_{j_1,j_2}(f)$$

and is a Lipschitz mapping with constant λ from $\mathbf{X}$ to Y that does not depend on the components j_1 and j_2 of its argument.

Finally, we will need another property.

Lemma 91 *If (X, d_X) is a metric space, then the diagonal map $\Delta : X \to X \times X$ defined by*

$$\Delta(x) = (x, x)$$

is nonexpanding.

Proof: $\mathrm{d}(\Delta_X(x_1), \Delta_X(x_2)) = \max(\mathrm{d}(x_1, x_2), \mathrm{d}(x_1, x_2)) = \mathrm{d}(x_1, x_2)$. ■

5.4.3 A complete metric space of games

Games, as we have seen, are families of sequences of moves; they may therefore be viewed as trees. Solving recurrence equations on trees is usually done using a metric known as the *tree metric*.

Consider two distinct trees A and B. In order to verify whether A and B are equal, it is possible to walk them in parallel using a breadth-first order. It is quite natural to quantify this difference by considering the depth at which the first mismatch is found, which also happens to be the length of the shortest path that is in only one of the trees; call l this value. This value defines a metric by setting the distance between A and B to be

$$\mathrm{d}(A, B) = 2^{-l}$$

The fact that this defines an ultrametric is a routine verification.

Suppose now that A and B are represented as prefix-closed collections of paths, and define the distance between two distinct paths to be

$$\mathrm{d}_0(a, b) = 2^{-|a \sqcap b|}$$

where $a \sqcap b$ is the longest common prefix of a and b. The definition above can now be rewritten as

$$\mathrm{d}_0(A, B) = \max(\sup_{a \in A} \inf_{b \in B} \mathrm{d}(a, b), \sup_{b \in B} \inf_{a \in A} \mathrm{d}(b, a))$$

which happens to be the formula defining the Hausdorff metric [Eng77, Problem 4.5.22]. It is to be noted, however, that the usual properties of the Hausdorff topology cannot be taken for granted as the space of paths is not, in general, complete or locally compact; in particular, we may not assume without proof that the Hausdorff space is complete.

Definition 92 *The* tree metric d_0 *on games is defined by*

$$\mathrm{d}_0(A,B) = \max(\sup_{p\in A}\inf_{q\in B} \mathrm{d}_0(p,q), \sup_{q\in B}\inf_{p\in A} \mathrm{d}_0(q,p))$$

with $\mathrm{d}_0(p,q) = 2^{-n}$ *where* n *is the length of the longest common prefix of* p *and* q.

Lemma 93 *The space of games equipped with the tree metric* $(\mathcal{G}, \mathrm{d}_0)$ *is a complete ultrametric space.*

Proof: first, note that the map d_0 on the set of positions is an ultrametric, and that it induces the discrete topology (as the ball of centre p and diameter $2^{-|p|-1}$ is equal to $\{p\}$, all the singletons are open). Therefore, games are closed subspaces of the space of positions, and by a standard property of the Hausdorff metric, $(\mathcal{G}, \mathrm{d}_0)$ is an ultrametric space.

However, as the space of positions is not complete, it is not immediate that the Hausdorff space is complete. Let therefore $(A_n)_{n\in\omega}$ be a Cauchy sequence of games. We are going to prove that

$$\limsup_{n\to\infty} A_n$$

exists and is the limit of $(A_n)_{n\in\omega}$. For all $n \in \omega$, let

$$B_n = \bigcup_{k\geq n} A_k$$

As prefix-closedness is preserved by union, B_n is a game for all n, and the sequence $(B_n)_{n\in\omega}$ is decreasing. As

$$\mathrm{d}_0(B_n, A_n) \leq \sup_{k\geq n} \mathrm{d}_0(A_n, A_k)$$

(B_n) is also a Cauchy sequence, and if either of (A_n) or (B_n) converge, then they both converge and their limits are equal.

Let now

$$A = \bigcap_{n\in\omega} B_n$$

As prefix-closedness is preserved by intersection, A is a game. As

$$\mathrm{d}_0(B_n, A) \leq \sup_{k\geq n} \mathrm{d}_0(B_n, B_k)$$

A is the limit of (B_n), and therefore also that of (A_n). ■

Unfortunately, the tree metric d_0 does not give us enough contractive maps for the needs of solving recursive equations with Banach's theorem. For example, neither of the maps $A \mapsto B \to A$, or $A \mapsto A \times B$, are contractive, (they are merely nonexpanding). A more convoluted approach is therefore needed, which consists in applying the above method twice, first to moves and then to positions.

Definition 94 *Let $p = m_0 \cdots m_{n-1}$ be a position, and M_p the set of moves m such that $p \cdot m$ is a legal position. For any move $m \in M_p$, the* weight *of m w.r.t. p, written $\mathrm{w}_p(m)$, is defined by*

$$\mathrm{w}_p(m) = 2^{-|c|}.$$

The ultrametric d_p on M_p is defined, for distinct moves m, m', by

$$\mathrm{d}_p(m, m') = 2^{-|c|}$$

where c is the longest common prefix of the components of m, m'.

We now define the metric d on positions. Given two distinct positions p and p', either one is the prefix of the other, in which case we will use the weight of the first differing move, or neither is a prefix of the other, in which case we use the distance between the first differing moves.

Definition 95 *The metric* d *on the set of positions is defined as follows. Given two distinct positions p, p', let $q = m_0 \cdots m_{n-1}$ be their longest common prefix. If*

$$\begin{aligned} p &= q \cdot m_n \cdot r \\ p' &= q \cdot m'_n \cdot r' \end{aligned}$$

then

$$\mathrm{d}(p, p') = 2^{-n} \mathrm{d}_q(m_n, m'_n).$$

On the other hand, if

$$\begin{aligned} p &= q \cdot m_n \cdot r \\ p' &= q \end{aligned}$$

then

$$\mathrm{d}(p, p') = 2^{-n} \mathrm{w}_q(m_n).$$

This metric does not induce the discrete topology on the set of positions. Indeed, let $(p_n)_{n\in\omega}$ be the sequence of positions defined by

$$p_n = m \cdot m_c \qquad\qquad \text{where } c = \underbrace{1 \cdots 1}_{n} \cdot 0,$$

where m is an initial move. Now, $\mathrm{d}(m, p_n) = 2^{-n-1}$, so $(p_n)_{n\in\omega}$ converges to $p = m$, but $(p_n)_n$ is not a stationary sequence. However, the following lemma shows that games are closed, which allows us to use the Hausdorff formula.

Lemma 96 *Games are closed subspaces of the space of positions equipped with the metric* d.

Proof: let A be a game, and p a position that is not in A. As the set of prefixes of p is finite, the set

$$\{2^{-|q|} \mid q \in A \text{ and } q \text{ is a prefix of } p\}$$

is a finite set of strictly positive reals; therefore, it has a minimal element $\delta_1 > 0$. Similarly, as p is of finite length, the set

$$\{2^{-|c|} \mid c \text{ is a prefix of the component of a move in } p\}$$

has a minimal element $\delta_2 > 0$. We are going to show that the distance between p and all elements of A is at least $\delta = \delta_1\delta_2$.

Let r be a position in A. If r is a prefix of p, then by definition of δ, $\mathrm{d}(p, r) \geq \delta$. If r is not a prefix of p, let r' be the longest common prefix of p and r, and let m_c, $m'_{c'}$ be moves such that $p = r' \cdot m \cdot p''$ and $r = r' \cdot m' \cdot r''$, and let c'' be the longest common prefix of c and c'.

$$\begin{aligned}\mathrm{d}(p, r) &= 2^{-|r'|}\, 2^{-|c''|} \\ &\geq 2^{-|r'|}\, 2^{-|c|} \\ &\geq \delta_1\delta_2 = \delta \quad \blacksquare\end{aligned}$$

Definition 97 *The metric* d *on the set of games is defined by the Hausdorff formula*

$$\mathrm{d}(A, B) = \max(\sup_{p\in A} \inf_{q\in B} \mathrm{d}(p, q), \sup_{p\in B} \inf_{q\in A} \mathrm{d}(p, q)).$$

Theorem 98 *The space of games* $(\mathcal{G}, \mathrm{d})$ *is a complete ultrametric space.*

Proof: by standard properties of the Hausdorff metric, Lemma 96 implies that $(\mathcal{G}, \mathrm{d})$ is an ultrametric space. The proof of the fact that it is complete is analogous to the proof of Lemma 93.

Indeed, let $(A_n)_{n\in\omega}$ be a Cauchy sequence of games. As above, take

$$B_n = \bigcup_{k\geq n} A_k, A = \bigcap_{n\in\omega} B_n.$$

As before, B_n is a game for all n, A is a game, and

$$\begin{aligned}\mathrm{d}(A_n, B_n) &\leq \sup_{k\geq n} \mathrm{d}(A_n, A_k), \\ \mathrm{d}(B_n, A) &\leq \sup_{k\geq n} \mathrm{d}(B_n, B_k),\end{aligned}$$

and therefore A is the limit of $(A_n)_{n\in\omega}$. $\blacksquare$

Finally, the following property relates the tree metric and the metric on games.

Lemma 99 *For any games A, B, $\mathrm{d}(A,B) \leq \mathrm{d}_0(A,B)$; therefore, any space open for d_0 will be open for d.*

Proof: clearly, for any positions p and q, $\mathrm{d}(p,q) \leq \mathrm{d}_0(p,q)$. This relation is preserved by the Hausdorff formula.

Let $\mathcal{A} \subseteq \mathcal{G}$ be open for d, and $A \in \mathcal{A}$. As $\mathcal{A}$ is open for d_0, there exists $\epsilon > 0$ such that the d_0-open ball of centre A and diameter ϵ is in $\mathcal{A}$. But as $\mathrm{d} \leq \mathrm{d}_0$, the d-open ball of centre A and diameter ϵ is also in $\mathcal{A}$, which concludes the proof. ∎

5.4.4 Solving recursive type equations

In order to be able to construct arbitrary recursive types, we will want to show that contractivity is preserved through all of our constructions.

5.4.4.1 Simple type constructors

The fact that the basic type constructors have contractive interpretations is immediate as our metric has been constructed *ad hoc* (see the discussion on page 108 and Definition 94).

Lemma 100 *The type constructors $\cdot \to \cdot$ and $\cdot \times \cdot$ are contractive in all of their arguments.*

Proof: For any games A, A', B, B', a routine verification shows that

- $\mathrm{d}(A \to B, A \to B') \leq \max(\frac{1}{4}\mathrm{d}(A,A'), \frac{1}{2}\mathrm{d}(B,B'))$, and
- $\mathrm{d}(A \times B, A' \times B') \leq \frac{1}{2}\max(\mathrm{d}(A,A'), \mathrm{d}(B,B'))$. ∎

5.4.4.2 Fixpoints of fixpoints

The second issue is that of fixpoints: in order to be able to take the fixpoint of a fixpoint, we need to know that contractivity is preserved through fixpoints. This is a general property of metric spaces, and was stated as Lemma 90 on page 106.

5.4.4.3 Fixpoints of bounds

Unfortunately, there is no obvious reason why contractivity should be preserved by taking order-theoretic bounds.

Consider indeed the set of non-empty closed bounded sets of the plane equipped with the Hausdorff metric and ordered by inclusion. Let A, A', B and B' be bounded closed subsets of the plane; then it is indeed true that $\mathrm{d}(A \cup B, A' \cup B') \leq \max(\mathrm{d}(A, A'), \mathrm{d}(B, B'))$. On the other hand, as shown in Fig. 5.2, an analogous property does not in general hold with respect to intersection.

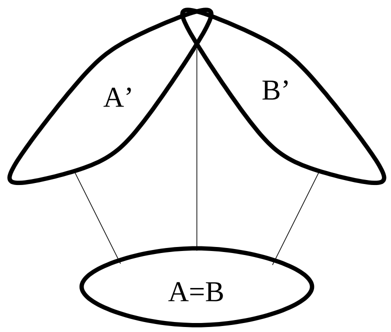


Figure 5.2: The Hausdorff distance is not preserved by intersection

.

As the bounds defined in Definition 60 on page 90 make use of both union and intersection, distances are not conserved by either the bound operations. Indeed, consider the following games

$$
\begin{aligned}
A = B &= \mathrm{Pref}(q \cdot a_l^{\mathbf{tt}}, q \cdot a_l^{\mathbf{ff}}), \\
A' &= \mathrm{Pref}(q \cdot a_l^{\mathbf{tt}}), \\
B' &= \mathrm{Pref}(q \cdot a_l^{\mathbf{ff}}).
\end{aligned}
$$

Then $\mathrm{d}(A, A') = \mathrm{d}(B, B') = 1/4$. As $A \wedge B = A = B$, and $A' \wedge B' = \mathrm{Pref}(q)$, we have

$$\mathrm{d}(A \wedge B, A' \wedge B') = \frac{1}{2} > \frac{1}{4} = \max(\mathrm{d}(A, A'), \mathrm{d}(B, B')).$$

These are artificial examples. We, however, are only interested in the particular maps over games that are the interpretation of types. As we were unable to find a realistic counterexample to the following property, we will assume it throughout the remainder of this thesis.

Conjecture 101 *Let $\mathcal{F} = \{F_i \mid i : A \to B\}$ be a family of maps over games where each F_i is the interpretation of a type (see page 116), and assume that each F_i is contractive (resp. nonexpanding). Then the bounds $\bigwedge \mathcal{F}$ and $\bigvee \mathcal{F}$ are themselves contractive (resp. nonexpanding).*

Why fixpoints of bounds are important The above property is only useful when proving the soundness of subtyping of the fixpoint of a quantified type, *i.e.* a type of the form

$$\mu X.\forall Y \leq A.B[X, Y]$$

or, more wickedly, of the form

$$\mu X.\forall Y \leq A[X].B[Y].$$

Thus, the failure of the proof does not put into doubt the correctness of our construction when restricted to either the fragment of the calculus that excludes recursive types, nor the fragment of our calculus that excludes quantification. While either calculus is already richer than most of those found in the literature, types of the form $\mu.\forall$ are useful for interpreting object calculi. This issue will be discussed further in Chapter 7.

5.4.4.4 Subtyping fixpoints

In order to be able to validate Amadio and Cardelli's subtyping rule, we will need a property that shows how the order structure carries over through fixpoints. We first show that the topology induced by the metric d is finer than the set of (principal) order ideals for $\preccurlyeq$.

Lemma 102 *If A is a game, then the order ideal*

$$A\downarrow = \{B \in \mathcal{G} \mid B \preccurlyeq A\}$$

and the order filter

$$A\uparrow = \{B \in \mathcal{G} \mid A \preccurlyeq B\}$$

are closed with respect to d.

Proof: let $C \not\preccurlyeq A$. There are two cases to consider: A contains an accessible odd-length position not in C, or C contains an accessible even-length position not in A.

Suppose first that there is an odd-length $p \in A$ such that $p \notin C$ but all strict prefixes of p are in C. Let n be the length of p, and write

$$p = m_1 \cdots m_n$$

and define the strictly positive real number δ as the minimum of the weights of the prefixes of p (see Def. 94 on page 109):

$$\delta = \min\{2^{-i}\mathrm{w}(m_i)) \mid 1 \leq i \leq n\}.$$

We are going to show that for any $B \preccurlyeq A$, $\mathrm{d}(B, C) \geq \delta$.

Indeed, let $B \preccurlyeq A$. Then either $p \in B$, in which case as $p \notin C$, $\mathrm{d}(B, C) \geq 2^{-1}\mathrm{w}(m_i) \geq \delta$. Or else $p \notin B$; as $B \preccurlyeq A$, $p_{-1} \notin B$, whence there exists a strict prefix $m_1 \cdots m_k$ of p that is not in B. Hence $\mathrm{d}(B, C) \geq 2^{-k}\mathrm{w}(m_k) \geq \delta$.

Conversely, suppose that there is an even-length position $p \in C$, $p \notin A$, but all strict prefixes of p are in A. As above, write

$$p = m_1 \cdots m_n,$$

and define the strictly positive real number δ as

$$\delta = \min\{2^{-i}\mathrm{w}(m_i)) \mid 1 \leq i \leq n\}.$$

Let $B \preccurlyeq A$; then either $p \notin B$, in which case $\mathrm{d}(B, C) \geq 2^{-1}\mathrm{w}(m_i) \geq \delta$. Or else there exists an even-length prefix $p' = m_1 \cdots m_k$ of p that is in B and not in A, whence $\mathrm{d}(B, C) \geq 2^{-k}\mathrm{w}(m_k) \geq \delta$.

The proof of the second property is dual to the proof above. ∎

From this property, we may conclude that fixpoints preserve monotonicity.

Lemma 103 *Let f, g be monotone, contractive maps over games such that for any game A, $f(A) \preccurlyeq g(A)$. Then* $\mathrm{fix}(f) \preccurlyeq \mathrm{fix}(g)$.

Proof: fix an integer n, and note that for any $k \geq n$, $g^n(\perp) \in g^k(\perp)\uparrow$, hence by the second property in Lemma 102, $\mathrm{fix}(g) \geq g^n(\perp)$. As $f^n(\perp) \preccurlyeq g^n(\perp)$, $f^n(\perp) \preccurlyeq \mathrm{fix}(g)$, *i.e.* $f^n(\perp) \in \mathrm{fix}(g)\downarrow$, and hence by the first property in Lemma 102, $\mathrm{fix}(f) \preccurlyeq \mathrm{fix}(g)$. ∎

Chapter 6

A Game Semantics for the Calculus

In this chapter, we show how to interpret the typed calculus. The interpretation of the untyped calculus was described in Section 4.5.

6.1 Interpretation of types

Consider an environment E: in general, it will contain information on the bounds of type variables, of the form $X \leq A$, and information on the types of variables, of the form $x : B$. A *type assignment* is a map from type variables to games; formally, it is a **TyVar**-indexed tuple of games $\eta \in \prod_{X \in \mathbf{TyVar}} \mathcal{G}$. We write $\eta(X)$ for the value of η at variable X, and $\eta[X \backslash \mathcal{X}]$ for the type assignment equal to η except at variable X, where it has value $\mathcal{X}$.

We will interpret a type A as a mapping $[\![A]\!] \in \prod_{X \in \mathbf{TyVar}} \mathcal{G} \to \mathcal{G}$ that maps a type assignment to a type. Given such a mapping F and a type variable X, we write $\Lambda_X(F)$ for the map defined by

$$\Lambda_X(F)(B)\eta = F(\eta[X \backslash B]).$$

The interpretation of types is defined as follows :

$$
\begin{aligned}
[\![\mathbf{Bool}]\!]\eta &= \mathbf{Bool} \\
[\![\top]\!]\eta &= \top \\
[\![\bot]\!]\eta &= \bot \\
[\![A \to B]\!]\eta &= [\![A]\!]\eta \to [\![B]\!]\eta \\
[\![A \times B]\!]\eta &= [\![A]\!]\eta \times [\![B]\!]\eta \\
[\![X]\!]\eta &= \eta(X) \\
[\![\forall X \leq A.B[X]]\!]\eta &= \bigwedge_{\mathcal{X} \leq [\![A]\!]\eta} [\![B[X]]\!](\eta[X \backslash \mathcal{X}]) \\
[\![\exists X \leq A.B[X]]\!]\eta &= \bigvee_{\mathcal{X} \leq [\![A]\!]\eta} [\![B[X]]\!](\eta[X \backslash \mathcal{X}]) \\
[\![\mu X.B[X]]\!](E') &= (\mathrm{fix}(\Lambda_X([\![B[X]]\!])))\eta
\end{aligned}
$$

Note that it is not immediate that the constructions above are well defined, as we have been taking fixpoints of maps without checking that they are contractive.

Lemma 104 *Let A be a type. Then*

(i) $[\![A]\!]$ *is well defined;*

(ii) $[\![A]\!]$ *is a nonexpanding mapping;*

(iii) if A is guarded in $X \in$ **TyVar**, *then $[\![A]\!]\eta$ is contractive in $\eta(X)$;*

(iv) if $X \in$ **TyVar** *does not appear free in A, then $[\![A]\!]\eta$ does not depend on the value of $\eta(X)$.*

Proof: by induction on the syntax of A. If A is **Bool**, $\top$ or $\bot$, (i) is immediate, $[\![A]\!]$ is a constant function which implies (ii); no type variable appears in A, guarded or otherwise, so (iii) and (iv) are vacuously true.

If A is a type variable X, then (i) is immediate and $[\![A]\!]$ is a projection, which clearly implies (ii). As A is not guarded in any type variable, (iii) is vacuously true; finally, the only projection of its argument that $[\![A]\!]$ depends on is the component X, and X does indeed appear free in A.

If A is of the form $B \times C$ or $B \to C$, then A is guarded in all of its free variables. By definition of $[\![A]\!]$, (i) and (iv) follow from the induction hypothesis on the B_i, B, and C. As to (iii) (and therefore (ii)), it follows from the induction hypothesis and Lemmata 100 on page 111 and 91 on page 107.

If A is of the form $\forall X \leq A.B[X]$ (or $\exists_{X\leq A}B[X]$), $[\![A]\!]$ is well defined by Theorem 61. Properties (ii), and (iii) follow from Lemma 103 and Conjecture 101 on page 114; (iv) is immediate.

Finally, if A is of the form $\mu X.B[X]$, where $B[X]$ is guarded in X, then, by induction hypothesis, $B[X]$ is contractive in X and only depends on the values of its free variables. Therefore, $\mathrm{fix}(\Lambda_X(B[X]))$ is well defined, and only depends on the free variables of A, which proves (i) and (iv). Properties (ii) and (iii) follow from Lemma 103 and Conjecture 101 on page 114. ∎

Lemma 105 *Let $A[X]$ be a type. If $A[X]$ is covariant in X, then $[\![A]\!]$ is a map that is monotone in the value of X; if $A[X]$ is contravariant in X, then $A[X]$ is a map that is antimonotone in the value of X.*

Proof: by induction on the syntax of $A[\cdot]$. The only nontrivial case is that of $\mu Y.A[X]$, where A is covariant in Y, but that is a consequence of lemma 103 on page 114. ∎

6.2 Soundness of typing

Let E be a typing judgement and η a type environment. We say that η *satisfies* E if for any type variable X, whenever a bounding hypothesis $X \leq A$ appears in E, then $\eta(X) \preccurlyeq [\![A]\!]\eta$. We now have the vocabulary necessary to state that the interpretation of types given above validates the subtyping rules given in Section 3.2.

Lemma 106 (Soundness of subtyping) *If $E \vdash A \leq B$ is derivable using the rules in Section 3.2, then for any type assignment $\eta \in \prod_{X\in\mathbf{TyVar}} \mathcal{G}_X$ satisfying E,*

$$[\![A]\!]\eta \preccurlyeq [\![B]\!]\eta.$$

Proof: by induction on the length of the derivation of $E \vdash A \leq B$. We proceed by cases on the form of the last rule in the derivation of $E \vdash A \leq B$.

- The soundness of Refl and Trans is a consequence of the reflexivity and transitivity of $\preccurlyeq$ (Theorem 35).
- The soundness of TopSub is due to the fact that $[\![\top]\!]\eta = \top$, which is the maximum element of $\mathcal{G}$.

- The soundness of TopArrowSub is due to the fact that

$$[\![A \to \top]\!]\eta = [\![A]\!]\eta \to \top = \top.$$

- For ArrowSub, let A, A', B and B' be games such that $[\![A']\!]\eta \preccurlyeq [\![A]\!]\eta$ and $[\![B]\!]\eta \preccurlyeq [\![B']\!]\eta$. Then,

$$[\![A]\!]\eta \to [\![B]\!]\eta \preccurlyeq [\![A']\!]\eta \to [\![B']\!]\eta$$

i.e.

$$[\![A \to B]\!]\eta \preccurlyeq [\![A' \to B']\!]\eta.$$

- In order to prove the soundness of ProdSub, let E be a type anvironment and A, A', B, B' types such that E proves $A \leq A'$ and $B \leq B'$. Then

$$\begin{aligned}
[\![A \times B]\!]\eta &= [\![A]\!]\eta \times [\![B]\!]\eta \\
&\preccurlyeq [\![A']\!]\eta \times [\![B']\!]\eta && \text{by Lemma 32} \\
&= [\![A' \times B']\!]\eta.
\end{aligned}$$

- ForallSub is an immediate consequence of the fact that games constitute a complete lattice (Theorem 61 on page 90). Indeed, assume that for all η satisfying E, $[\![C]\!]\eta \preccurlyeq [\![A]\!]\eta$. Then

$$\begin{aligned}
\bigwedge_{X \preccurlyeq [\![A]\!]\eta} [\![B[X]]\!](\eta[X \backslash Y]) &\preccurlyeq [\![B[X]]\!](\eta[X \backslash C]) \\
&= [\![B[C]]\!].
\end{aligned}$$

  The proof of the soundness of ExistsSub is similar.

- The soundness of ForallBodySub is as follows. Suppose that for an assignment η satisfying $E, X \leq A$, $[\![B[X]]\!]\eta \preccurlyeq [\![B'[X]]\!]\eta$; this is equivalent to saying that for any assignment η' satisfying E, and for any game $A' \preccurlyeq [\![A]\!]\eta'$,

$$[\![B[X]]\!](\eta'[X \backslash A']) \preccurlyeq [\![B'[X]]\!](\eta'[X \backslash A']),$$

  whence, taking infima pointwise,

$$\bigwedge_{Y \preccurlyeq [\![A]\!]\eta} [\![B[X]]\!](\eta'[X \backslash Y]) \preccurlyeq \bigwedge_{Y \preccurlyeq [\![A]\!]\eta} [\![B'[X]]\!](\eta'[X \backslash Y]),$$

  i.e.

$$[\![\forall X \leq A.B[X]]\!]\eta \preccurlyeq [\![\forall X \leq A.B'[X]]\!]\eta.$$

  The proof of the soundness of ExistsBodySub is similar.

- In order to prove the soundness of ForallBoundSub, let η be an environment satisfying E, and suppose that $[\![A]\!]\eta \preccurlyeq [\![A']\!]\eta$. Note then that

  $$\{Y \in \mathcal{G} \mid Y \preccurlyeq [\![A']\!]\eta\} \supseteq \{Y \in \mathcal{G} \mid Y \preccurlyeq [\![A]\!]\eta\},$$

  whence, for any map on games F,

  $$\{F(Y) \mid Y \preccurlyeq [\![A']\!]\eta\} \supseteq \{F(Y) \mid Y \preccurlyeq [\![A]\!]\eta\},$$

  whence,

  $$\bigwedge_{Y \preccurlyeq [\![A']\!]\eta} F(Y) \preccurlyeq \bigwedge_{Y \preccurlyeq [\![A]\!]\eta} F(Y)$$

  in particular, for $F(Y) = [\![B[X]]\!](\eta[X\backslash Y])$,

  $$\bigwedge_{Y \preccurlyeq [\![A']\!]\eta} [\![B[X]]\!](\eta[X\backslash Y]) \preccurlyeq \bigwedge_{Y \preccurlyeq [\![A]\!]\eta} [\![B[X]]\!](\eta[X\backslash Y])$$

  i.e.

  $$[\![\forall X \leq A'.B[X]]\!]\eta \preccurlyeq [\![\forall X \leq A.B[X]]\!]\eta.$$

  For the proof of the soundness of ExistsBoundSub, we proceed dually.

- The soundness of MuSub comes from the fact that a covariant type is interpreted as a monotone map (Lemma 105 on page 117). Suppose that for any environment E' satisfying E, and for any game C, $[\![A[C]]\!]\eta \preccurlyeq [\![B[C]]\!]\eta$; then, as the maps $X \to [\![A[X]]\!]\eta$ and $X \to [\![B[X]]\!]\eta$ are monotone, by Lemma 103 on page 114,

  $$\mathrm{fix}\, X.([\![A[X]]\!]\eta) \preccurlyeq \mathrm{fix}\, X.([\![B[X]]\!]\eta)$$

  or, equivalently,

  $$(\mathrm{fix}\, X.[\![A[X]]\!])\eta \preccurlyeq (\mathrm{fix}\, X.[\![B[X]]\!])\eta,$$

  i.e.

  $$[\![\mu X.A[X]]\!]\eta \preccurlyeq [\![\mu X.B[X]]\!]\eta.$$

- The soundness of the quantifier dualisation rules ForallDual and ExistDual is a consequence of the property proved in Lemma 64 on page 92. Indeed, let E be a typing environment, and E' be an environment satisfying E; then

$$\begin{aligned}
&[\![\forall X \leq A.(B[X] \to C)]\!]\eta = \\
&\qquad = \bigwedge_{Y \leq A} [\![B[X] \to C]\!](\eta[X\backslash Y]) \\
&\qquad = \bigwedge_{Y \leq A} ([\![B[X]]\!](\eta[X\backslash Y])) \to [\![C]\!](\eta[X\backslash Y])) \\
&\qquad = \bigwedge_{Y \leq A} ([\![B[X]]\!](\eta[X\backslash Y])) \to [\![C]\!]\eta) && \text{as } X \text{ is not free in } C \\
&\qquad = (\bigvee_{Y \leq A} ([\![B[X]]\!](\eta[X\backslash Y]))) \to [\![C]\!]\eta && \text{by Lemma 64} \\
&\qquad = [\![\exists X \leq A.B[X]]\!]\eta \to [\![C]\!]\eta \\
&\qquad = [\![(\exists X \leq A.B[X] \to C]\!]\eta,
\end{aligned}$$

  and similarly for ExistDual. ■

We will now need to use the typing assumptions on variables included in an environment. Let E be a typing environment, and η a type assignment satisfying E. Let E' be the sequence of typing assumptions contained within E; the type $E\eta$ is defined by the following induction on E':

- if $E' = \emptyset$ then $E\eta = \top$;
- if $E' = (x : A)$ then $E\eta = [\![A]\!]\eta$;
- if $E' = (E'_0, x : A)$ then $E\eta = E'_0\eta \times [\![A]\!]\eta$.

Theorem 107 (Soundness of typing) *If $E \vdash M : A$ is derivable using the rules in Section 3.2, then for any type assignment η satisfying E,*

$$[\![M]\!] \preccurlyeq E\eta \to [\![A]\!]\eta.$$

Proof: by induction on the length of the derivation of $E \vdash M : A$. We proceed by cases on the last rule in the proof of $E \vdash M : A$ (see Fig. 3.6 on page 38). In what follows, we write Γ for the sequence of variables present in E.

- In order to prove the soundness of Axiom, we need to distinguish between the cases $E = \emptyset$ and $E \neq \emptyset$. In the first case, $\Gamma = x$, $[\![x \vdash x]\!] = \mathbf{I}$. If η satisfies E, then $E\eta = [\![A]\!]\eta$, and as $\mathbf{I} \preccurlyeq B \to B$ for any game B,

$$\mathbf{I} \preccurlyeq [\![A]\!]\eta \to [\![A]\!]\eta$$

i.e.

$$[\![x \vdash x]\!] \preccurlyeq E\eta \to [\![A]\!]\eta.$$

If, on the other hand, $E \neq \emptyset$, $\Gamma = \Gamma', x$, $[\![\Gamma \vdash x = \pi_r]\!]$. If η satisfies E, then $E\eta = C \times [\![A]\!]\eta$ for some game C, and as $\pi_r \preccurlyeq C \times B \to B$ for any games C and B,

$$\pi_r \preccurlyeq C \times [\![A]\!]\eta \to [\![A]\!]\eta$$

i.e.

$$[\![\Gamma \vdash x]\!] \preccurlyeq E\eta \to [\![A]\!]\eta.$$

- The soundness of the variable abstraction rule Lambda is as follows. Suppose that $E, x : A \vdash M : B$, and let η be an assignment that satisfies $E, x : A$. Then

  $$[\![\Gamma, x \vdash M]\!] \preccurlyeq ((E, x : A)\eta) \to [\![B]\!]\eta,$$
  $$[\![\Gamma, x \vdash M]\!] \preccurlyeq E\eta \times [\![A]\!]\eta \to [\![B]\!]\eta,$$

  and therefore

  $$\Lambda[\![\Gamma, x \vdash M]\!] \preccurlyeq E\eta \to [\![A]\!]\eta \to [\![B]\!]\eta,$$

  i.e.

  $$[\![\Gamma \vdash \lambda x.M]\!] \preccurlyeq E\eta \to [\![A \to B]\!]\eta.$$

- The function application rule App is sound due to Lemma 59. Indeed, let E be an environment, and M and N terms such that E proves $M : A \to B$ and $N : A$. Let η be a type environment satisfying E; then, by the induction hypothesis,

  $$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to ([\![A]\!]\eta \to [\![B]\!]\eta),$$
  $$[\![\Gamma \vdash N]\!] \preccurlyeq E\eta \to [\![A]\!]\eta,$$

  whence,

  $$\langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \rangle \preccurlyeq E\eta \to \langle [\![A]\!]\eta \to [\![B]\!]\eta, [\![A]\!]\eta \rangle,$$

and as for any games C and D, $\mathrm{eval} \preccurlyeq \langle C \to D, C\rangle \to D$, by Lemma 59,

$$\langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!]\rangle; \mathrm{eval} \preccurlyeq E\eta \to [\![B]\!]\eta,$$

i.e.

$$[\![\Gamma \vdash (M\ N)]\!] \preccurlyeq E\eta \to [\![D]\!]\eta.$$

- Soundness of the Top rule is of course dependent on the fact that application is total, which is technically expressed by the fact that for any game A, $A \to \top = \top$. Indeed, let M be a term, E an environment, and η an assignment satisfying E. Then

  $$[\![\Gamma \vdash M]\!] \preccurlyeq \top$$

  as $\top$ is the maximal element of $(\mathcal{G}, \preccurlyeq)$, and therefore,

  $$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to \top$$

  so, as $[\![\top]\!]\eta = \top$,

  $$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to [\![\top]\!]\eta.$$

- The soundness of Bool is immediate, as

  $$\mathbf{K}(\mathbf{tt}) \preccurlyeq \top \to \mathbf{Bool},$$

  and

  $$\mathbf{K}(\mathbf{ff}) \preccurlyeq \top \to \mathbf{Bool}.$$

- The soundness of the conditional rule Cond is as follows. Let E be a type environment, and M, N and P terms such that E proves $M : \mathbf{Bool}$, $N : A$ and $P : A$ for some type A. By hypothesis,

  $$\langle [\![\Gamma \vdash M]\!], \langle [\![\Gamma \vdash N]\!], [\![\Gamma \vdash P]\!]\rangle\rangle \preccurlyeq E\eta \to \mathbf{Bool} \times ([\![A]\!]\eta \times [\![A]\!]\eta),$$

  and

  $$\mathbf{ite} \preccurlyeq \mathbf{Bool} \times ([\![A]\!]\eta \times [\![A]\!]\eta).$$

  Composing the two, we get,

  $$[\![\Gamma \vdash \mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ N'\ \mathbf{fi}]\!] \preccurlyeq E\eta \to [\![A]\!]\eta.$$

- The soundness of the product formation rule Prod is as follows. Let E be an environment that proves $M : A$ and $N : B$, and η a type assignment that satisfies E. Then

$$[\![\Gamma \vdash M]\!] \preccurlyeq [\![E]\!]\eta \to [\![A]\!]\eta,$$
$$[\![\Gamma \vdash N]\!] \preccurlyeq [\![E]\!]\eta \to [\![B]\!]\eta,$$

whence,

$$\langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \rangle \preccurlyeq \langle E\eta \to [\![A]\!]\eta, E\eta \to [\![B]\!]\eta \rangle,$$

i.e.

$$\langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \rangle \preccurlyeq E\eta \to [\![A]\!]\eta \times [\![B]\!]\eta,$$

or else

$$\langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \rangle \preccurlyeq [\![E]\!]\eta \to [\![A \times B]\!]\eta.$$

- The soundness of the projection rule Proj-l is as follows. Suppose that $E \vdash M : A \times B$, and η is an assignment that satisfies E. Then

$$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to [\![A \times B]\!]\eta,$$

as $[\![A \times B]\!]\eta = [\![A]\!]\eta \times [\![B]\!]\eta$ and $\pi_l \preccurlyeq C \times D \to C$ for any games C and D,

$$[\![\Gamma \vdash \pi_l(M)]\!] \preccurlyeq E\eta \to [\![A]\!]\eta.$$

Soundness of the projection rule Proj-r is analogous.

- Soundness of the ForAll typing rule is as follows. If $E, X \leq A$ proves $M : B[X]$, then for any environment η satisfying E and for any type $Y \preccurlyeq [\![A]\!]\eta$,

$$[\![\Gamma \vdash M]\!] \preccurlyeq E(\eta[X \backslash Y]) \to [\![B[X]]\!](\eta[X \backslash Y]).$$

But then, as X does not appear in E, the expression on the left hand side of the $\to$ does not depend on Y, whence

$$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to [\![B[X]]\!](\eta[X \backslash Y]).$$

Taking the infimum of the above over all $Y \preccurlyeq [\![A]\!]$,

$$[\![\Gamma \vdash M]\!] \preccurlyeq \bigwedge_{Y \preccurlyeq [\![A]\!]} E\eta \to [\![B[X]]\!](\eta[X \backslash Y]),$$

and, by continuity of $\to$,

$$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to \bigwedge_{Y \preccurlyeq [\![A]\!]} [\![B[X]]\!](\eta[X \backslash Y]),$$

i.e.

$$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to [\![\forall X \leq A.B[X]]\!]\eta.$$

- The soundness of the subsumption rule Sub is an immediate consequence of the soundness of subtyping proved above and the transitivity of $\preccurlyeq$ proved in Theorem 35. Indeed, suppose that E proves $A \leq B$, and that η satisfies E; then

  $$[\![A]\!]\eta \leq [\![B]\!]\eta.$$

  If, furthermore, E proves $M : A$, then, by the induction hypothesis,

  $$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to [\![A]\!]\eta,$$

  and therefore

  $$[\![\Gamma \vdash M]\!] \preccurlyeq E\eta \to [\![B]\!]\eta. \quad \blacksquare$$

This leads to the following property, which says that terms that can be typed to a reasonable (non-top) type cannot go wrong.

Corollary 108 (Type Safety) *If M is a closed term such that $\vdash M : \mathbf{Bool}$, then it is not the case that $M \Downarrow_\epsilon$ **top**.*

Proof: by the soundness of typing, $[\![M]\!] \preccurlyeq \top \to [\![\mathbf{Bool}]\!]$ therefore $[\![M]\!] \neq_1 \top$, whence, by Lemma 49, M does not reduce to $\top$. $\blacksquare$

Note that the proof of this property does not use either computational adequacy (Corollary 54 on page 83) or inequational soundness (Corollary 55).

Chapter 7

Conclusions and Further Work

In this thesis, we have shown how to model calculi with subtyping in a framework derived from Game Semantics. A model for a calculus with a simple dynamics but a rich type structure has been developed. This model has been proven to be sound, both with respect to the reduction relation and with respect to typing.

The semantic construction has been split into two clearly separated parts. In a first phase (Chapter 3), an untyped model has been described; this model makes no reference at all to any notions related to typing. In a second phase (Chapters 4 through 6), types were interpreted with no reference to terms.

The untyped calculus introduced in the first phase contains a major original feature: the explicit introduction in the syntax of typing errors (perhaps remindful, in spirit if not technically, of what was done by Cartwright *et al.* [CCF94]). In addition to the fact that these errors correspond to a pervasive intuition of programmers, they give a clear-cut criterion for soundness of typing. The semantics given for the untyped calculus explicitly takes errors into account.

'The introduction of errors in the calculus had the consequence of distinguishing $\lambda x.\Omega$ from Ω, in a construction that, fundamentally, is a call-by-name model. Thus, it can be argued that we have come up with a construction that realistically models lazy functional programming languages while preserving the simplicity of the call-by-name model; whether our semantics does indeed reflect that of programming languages in general use remains to be seen.

The second phase of the construction explores a space of games in search of a sufficient number of elements to interpret the types that we are interested in. The space of games is constructed with no reference to strategies, and is shown to have a very rich structure; this structure is not fully exploited by the syntax, and we have shown, for instance, the existence of arbitrary bounds of games (while we only use bounds of images of order ideals under certain maps) as well as that of singleton games. The notion of orthogonal of a game, that, again, was introduced but not reflected in the syntax, is intriguing and deserves further investigation.

The independence of the two constructions is, I believe, both the greatest drawback and the greatest strength of our approach. By not allowing oneself to refer to types while interpreting terms, many constructions are somewhat more complicated than they could otherwise be; for example, the constructions in Section 4.4.1 on page 72 would be much more natural if introduced after Chapter 5. However, this also means that dynamics can be discussed and studied without unrelated typing notions obscuring the matter; for example, all the constructions in Chapter 3 were implemented under the form of an interpreter in a purely untyped framework; had statics been intermingled with dynamics, the implementation would have required a typechecker*.

Similar advantages are gained from interpreting types with no reference to the dynamics. There is a clear conceptual advantage: it is possible to reason about types without being distracted by the dynamics. But there is also an advantage of a more technical nature: because the constructions relating to types are free-standing, it is possible to use our framework to reason about types without having settled for a particular kind of dynamics. For example, the rules for dualisation of quantifiers in Fig. 3.9 on page 39 are new, and were found by exploring the space of types. To give another example, a common issue is whether the types $\forall X \leq \bot\,.\,X \to X$ and $\forall X \leq \bot\,.\,\bot \to \bot$ can or should be identified [Pie97]; our model answers this question in the affirmative.

Another common issue is that of the interpretation of object types. The self-application interpretation of objects (see Section 2.2.4.4 on page 21) interprets an object as a record of functions that can be applied to the object itself; thus, a first approach would interpret an object type as

$$[m_i : B_i]^{i=1\cdots n} = \mu X.\langle m_i : X \to B_i\rangle^{i=1\cdots n}.$$

This kind of interpretation, however, yields types that are invariant in the B_i (due to the contravariant occurrence of X in the body of the μ); proper interpretation of objects requires covariant types. Thus, a finer approach is necessary:

$$[m_i : B_i]^{i=1\cdots n} = \mu X.\forall Y \sim X.\langle m_i : Y \to B_i\rangle^{i=1\cdots n}$$

for some suitably chosen relation $\sim$. The search for the proper definition of $\sim$ can be done with no reference to the dynamics.

*This implementation was used for typesetting the figures in Section 4.1 on page 42.

7.1 Limitations

The main limitation of this work is the lack of a proof of Conjecture 101 on page 112. What are the consequences of the truth or otherwise of this property?

This property has been used exactly once in this thesis, namely in the proof of Lemma 104 on page 116 in order to prove the existence of types of the form

$$\mu X.\forall Y \leq A.B$$
$$\mu X.\exists Y \leq A.B$$

Thus, the lack of a proof only casts a doubt on the suitability of our constructions for calculi that include both recursive types and quantification; calculi that include either one or the other of these features can be treated with our techniques as they stand.

One of the main motivations for the study of subtyping, however, is its importance for object-oriented languages. Common techniques for interpreting objects [AC96] use types that have the very form that we cannot interpret without reference to Conjecture 104; and, indeed, the types outlined at the end of the previous section have this form.

7.2 Changes to the dynamics

A number of extensions could be made to our untyped calculus. Most obviously, the reduction semantics could be changed to a call-by-value or lazy semantics. Perhaps more interesting would be the introduction of state, in the form of imperative variables [AM96], or of violations of the control discipline, in the form of non-local jumps [Lai97].

It is my belief that such enriched calculi can be studied just as we have done for the purely-functional, call-by-name calculus; this belief is justified by the fact that we have been careful never to assume well-bracketing or innocence of strategies. However, the introduction of every new feature enlarges the set of "interesting" strategies, and also makes the observational equivalence finer, thus providing opportunities for new types.

The set of well-bracketed positions, for example, is prefix-closed; there exists, therefore, a game B of (strategies consisting of) well-bracketed positions. To this game corresponds a type $\mathbf{B}$ of "jump-free" terms. With the addition of such a type, the identity term should be typable as

$$\mathbf{B} \wedge \forall A \leq \top.A \to A$$

which is a subtype of

$$\forall A \leq \top.(\mathbf{B} \wedge A) \to (\mathbf{B} \wedge A)$$

expressing, respectively, that the identity is jump-free and that it maps jump-free terms to jump-free terms.

The **call/cc** term of Scheme and some dialects of SML[†] can be typed as

$$\forall X.((X \to \bot) \to X) \to X$$

but of course not as

$$\mathbf{B} \wedge \forall X.((X \to \bot) \to X) \to X.$$

Whether such types should be exposed to the programmer and whether they can provide useful information to compiler optimisers remain, however, open questions.

The situation is less attractive with imperative variables: innocence [AM96] is not a game. Games represent conditions on positions, and a strategy belongs to a game if all its positions satisfy a given condition; innocence, however, is not a condition on positions, but a condition on strategies. We have no hope, therefore, of modelling a type of all "externally functional" terms, which is especially unfortunate as this is the sort of information that is useful both to the programmer and the compiler writer.

Another interesting extension would be the introduction of nondeterministic or concurrent features. Throughout our work, we have assumed that strategies are deterministic. While this assumption was not a hypothesis in most of our constructions (a notable exception being Lemma 37[‡]), it is not at all clear how to lift this limitation. Simply lifting the determinacy condition in Definition 8 would make our definition of composition correspond to the so-called *angelic* notion of composition, yielding a notion of reduction that guesses the future and "randomly" chooses the path that will avoid errors as much as possible. It is difficult to argue that such a model is realistic.

At any rate, determinacy is a property of a strategy, not of a position. A simple extension to our model is therefore not very likely to yield anything new.

7.3 More types!

As it stands, our language of types already allows expressing a number of constraints which are not, usually, seen as strictly belonging to the realm of program-

[†]Which is not quite the same as the $\mathcal{C}$ combinator of Felleisen [FFKD86].

[‡]Actually, we used this hypothesis throughout Section 4.3.3, but this assumption was only done to simplify the (already involved) technical treatment.

ming languages, but rather to that of static analysis, (say, Abstract Interpretation [CC92, BHA86], or type-based static analysis). For example, the information that an integer function is constant (in the sense that it doesn't use its argument, not that it returns constant results) can be expressed by the fact that it can be assigned the type

$$\top \to \mathbf{N}$$

in addition to

$$\mathbf{N} \to \mathbf{N}.$$

A number of extensions to the type language can be straightforwardly interpreted; this mostly stems from the richness of the space of games explored in Chapter 5. For example, the addition of an *intersection* operation $\cap$ on types (interpreted as intersection of the respective games) allows for the interpretation of "typeful bottom types" $\perp_T = T \cap \perp$ for any type T; with this extension, we can express the fact that an integer function is strict (in the sense that it loops when applied to a looping term) by typing it as

$$\perp \to \perp_{\mathbf{N}}.$$

In Section 5.2.3, we explored singleton games and orthogonals. These notions can be carried over to the syntax without much trouble, and it is not unlikely that a type system that includes such notions could allow us to type the self-application interpretation of object calculi.

A more challenging extension is the introduction of *dependent types*. Considering that the space of games has arbitrary infima, there should be no significant issues with interpreting these, and the subtyping rules validated should be fairly natural [AC97]. However, we have currently limited ourselves to non-normalising calculi; and, as far as syntax is concerned, non-normalising calculi with dependent types are something of a rarity [ML71].

Bibliography

[Abr90] Samson Abramsky. The lazy Lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

[Abr97] Samson Abramsky. Semantics of interaction. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, pages 1–31. Cambridge University Press, 1997. Based on lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge UK, September 1995.

[AC93] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proc. POPL'91.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[AC97] David Aspinall and Adriana Compagnoni. Subtyping dependent types. LFCS Report ECS–LFCS–97–370, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, October 1997.

[AHM98] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings of the thirteenth annual IEEE symposium on Logic in Computer Science*, pages 334–344, Indianapolis, Indiana, June 1998.

[AJ94] Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543–579, June 1994. Also appeared as Technical Report 92/24 of the Department of Computing, Imperial College of Science, Technology and Medicine.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). *Information and Computation*, 163(2):409–470, 2000.

[AM94] Samson Abramsky and Guy McCusker. Games for recursive types. In *Proceedings of the Second Workshop of the Theory and Formal Methods Section*. Department of Computing, Imperial College, 1994.

[AM95] Samson Abramsky and Guy McCusker. Games and full abstraction for the lazy λ-calculus. In *Proceedings of the 10th annual IEEE symposium on Logic in Computer Science (LICS'95)*, 1995.

[AM96] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions (extended abstract). In *Proceedings of the 1996 Workshop on Linear Logic*, volume 3 of *Electronic notes in Theoretical Computer Science*. Elsevier, 1996.

[AM99] Samson Abramsky and Paul-André Melliès. Concurrent games and full completeness. In *Proceedings of the Fourteenth International Symposium on Logic in Computer Science (LICS'99)*, pages 431–442. IEEE Computer Society Press, 1999.

[ANS94] Information technology: Programming language Common Lisp. ANSI standard X3.226–1994, 1994.

[Bak95] Henry G. Baker. 'Use-once' variables and linear objects — storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[BDER97] P. Baillot, V. Danos, T. Ehrhard, and L. Regnier. Believe it or not, AJM's games model is a model of classical linear logic. In *Proceedings of the Twelfth International Symposium on Logic in Computer Science (LICS'97)*. IEEE Computer Society Press, 1997.

[BDG+89] Daniel G. Bobrow, Linda G. DiMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification. *Lisp and Symbolic Computation*, 1(3/4):245–394, January 1989. Republished in revised form as Chapter 28 of [Ste90].

[BHA86] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis. *Science of Programming*, 7:249–278, 1986.

[BL90] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87:196–240, 1990.

[Bou74] Nicolas Bourbaki. *Topologie Générale, chapitres 5 à 10*. Éléments de Mathématiques. Hermann, 1974.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[Car97] Luca Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*. 1997.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CCF94] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract models of observably sequential languages. *Information and Computation*, 111(2):297–401, 1994.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report 52, Systems Research Center, Digital Equipment Corporation, Palo Alto, November 1989.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[Chr00] Juliusz Chroboczek. Game Semantics and subtyping. In *Proceedings of the fifteenth annual IEEE symposium on Logic in Computer Science*, Santa Barbara, California, June 2000.

[Chr01] Juliusz Chroboczek. Subtyping recursive games. In *Proceedings of the Fifth International Conference on Typed Lambda Calculi and Applications (TLCA'01).*, Kraków, Poland, May 2001.

[Chu41] Alonzo Church. *The calculi of lambda-conversion*. Number 6 in Annals of mathematics studies. Princeton University Press, 1941.

[Com95] Adriana Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, University of Nijmegen, The Netherlands, January 1995.

[DD92] Vincent Danos and Roberto Di Cosmo. Initiation to Linear Logic. Unpublished lecture notes, 1992.

[Eng77] Ryszard Engelking. *General Topology*, volume 60 of *Monografie Matematyczne*. Państwowe Wydawnictwo Naukowe, Warsaw, 1977.

[FFKD86] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 131–141, Cambridge, MA, June 1986. IEEE Computer Society.

[Gir87] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. Addison-Wesley, August 1996.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GTL89] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press, 1992.

[Har99] Russ Harmer. *Games and Full Abstraction for Nondeterministic Languages*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing, 1999.

[HJ89] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall International series in Computer Science. Prentice Hall International, 1989.

[HO00] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163:285–408, December 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, October 1969. Reprinted in [HJ89].

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computer science. Englewood Cliffs, N.J. London: Prentice-Hall International, 1985.

[Hof99] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth International Symposium on Logic in Computer Science (LICS'99)*, Trento, Italy, 1999.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2(4):335–355, 1973. Reprinted in [HJ89].

[ISO90] Information technology — programming language C. International standard ISO/IEC 9899:1990, 1990.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kur66] Kazimierz Kuratowski. *Introduction à la théorie des ensembles et à la topologie*. L'enseignement Mathématique. Institut de Mathématiques de l'Université de Genève, 1966.

[Lai97] James Laird. Full abstraction for functional languages with control. In *Proceedings of the Twelfth International Symposium on Logic in Computer Science (LICS'97)*, Warsaw, Poland, 1997.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[LRVD99] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml system, documentation and user's guide, Release 2.02*. Institut National de Recherche en Informatique et Automatique, Rocquencourt, 1999.

[McC62] John McCarthy. *Lisp 1.5 Programmer's Manual*. The MIT Press, 1962.

[McC96] Guy McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types.* PhD thesis, Imperial College, University of London, 1996.

[Mil77] Robin Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science*, 4(1), 1977.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[ML71] Per Martin-Löf. A theory of types. Unpublished typescript, October 1971.

[Mog90] Eugenio Moggi. An abstract view of programming languages. LFCS Report ECS–LFCS–90–113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, April 1990.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Types*, 10(3):470–502, July 1988.

[MTHM90] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[Nau63] Peter Naur. Revised report on the algorithmic language ALGOL 60. *Numerische Mathematik*, 4:420–453, 1963. Also published in the Communications of the ACM 6(1):1–17 (1963) and Computer Journal 5:349–.

[Pie97] Benjamin C. Pierce. Bounded quantification with bottom. CSCI Technical Report 492, Indiana University, November 1997.

[Pit93] Kent M. Pitman. What's in a name? *Lisp Pointers*, 6(1), January–March 1993.

[Pit94] A. M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *Mathematical Foundations of Programming Semantics, Proc. 9th Int. Conf., New Orleans, LA, USA*, volume 802 of *Lecture Notes in Computer Science*, pages 72–82. Springer-Verlag, Berlin, 1994.

[Pit95] Andrew M. Pitts. Relational properties of domains. *Information & Computation*, (127):66–90, 1995.

[PJH99] Simon Peyton-Jones and John Hughes. *Haskell 98: A Non-strict, Purely Functional Language*, February 1999.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975. Also published as Memorandum SAI–RM–6, School of Artificial Intelligence, University of Edinburgh, Edinburgh, 1973.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[PvE96] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean — a general purpose, higher order, pure and lazy functional programming language based on graph rewriting designed for the development of sequential, parallel and distributed real world applications.* University of Nijmegen, March 1996. Version 1.1.

[PWO97] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computer Science*, 9:49–67, 1997. Also available as BRICS Technical Report RC–95–33.

[RC92] Jonathan Rees and William Clinger. Revised^4 report on the algorithmic language Scheme. AI memo 848b, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, November 1992.

[Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North Holland, 1981.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Report CMU–CS–88–159, Carnegie Mellon University, June 21 1988.

[SP94] Martin Steffen and Benjamin Pierce. Higher-order subtyping. LFCS Report ECS–LFCS–94–280, Department of Computer Science, University of Edinburgh, 1994.

[Ste90] Guy L. Steele. *Common Lisp: The Language.* Digital Press, second edition, 1990.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, August 1991.

[Wij75] A. Van Wijngaarden. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5:1–236, 1975. Also published by Springer-Verlag (1976), and SIGPLAN Notices 12(5) (1977).

[WM81] D. L. Weinreb and D. A. Moon. *Lisp Machine Manual.* Artificial Intelligence Laboratory, MIT, Cambridge, MA, July 1981.

Principal notations

Index