T H E S I S

*presented at*

Paris 7 University – Denis Diderot

*to obtain the title of*

Doctor of Science

Computer Science specialty

# Theory, conception and realisation of a programming language adapted to XML

*presented by*

Alain Frisch

*on 13 December 2004, in front of a jury composed of:*

| | | |
|---|---|---|
| Mr. | Pierre-Louis Curien | *President* |
| Mr. | Giuseppe Castagna | *Thesis director* |
| Messrs. | Giorgio Ghelli | |
| | Martin Odersky | *Referees* |
| Mrs. | Mariangiola Dezani | *Examiner* |
| Mr. | Xavier Leroy | *Examiner* |

# Abstract

This thesis describes the theoretical foundations of a type-safe and higher-order functional language, adapted to the manipulation of XML documents. The first part presents the semantic foundations: a type algebra with recursive types, boolean combinations, arrow and product constructors; the definition of a semantic subtyping relation via a set-theoretic model notion for types; and a description of the functional core of the language, in particular its type system and its type-driven dynamic semantics. The second part focuses on the algorithmic aspects: computing the subtyping relation and compiling pattern matching with optimizations. The third part presents the ℂDuce language, built on top of the functional core, as well as some of the novel techniques used in its implementation.

# Acknowledgements

My thanks go, first of all, to Giuseppe Castagna who kindly supervised my thesis and my DEA internship. He left me free to choose the directions in which to orient my work, and the way to approach them; he always knew how to show me great confidence and he advised and encouraged me during decisive moments. Thanks to him, these years of thesis were a great enrichment for me, and I developed a taste for research. I also thank Véronique Benzaken who accompanied my thesis work.

Giorgio Ghelli and Martin Odersky have done me the honor of being referees for this thesis. I am very grateful to them for having accepted this heavy task, despite the language barrier. I would also like to thank Mariangiola Dezani, Pierre-Louis Curien, and Xavier Leroy for agreeing to participate in my jury.

This thesis could not have been completed without the flexibility of the General Council of Information Technologies and the research department of Télécom Paris, which authorized me to carry out my research work in parallel with a specially arranged engineering training. INRIA should also be thanked for allowing me to devote my first weeks within it to finishing the writing of the thesis.

The ℂDuce language, whose theoretical foundations are presented in this thesis, is a team effort. Giuseppe Castagna and Véronique Benzaken participated in defining the language and writing the documentation. Some parts of the implementation were carried out by other students: Cédric Miachon for the CQL query language integrated in ℂDuce, Stefano Zacchiroli for the XML Schema support, and Julien Demouth for the interface with the Objective Caml language. The researchers and students who contributed in one way or another to ℂDuce, as well as the small community of early users, helped create a positive dynamic around the project.

My office colleagues at the ENS deserve credit for putting up with me. I would like to thank them for the nice atmosphere of this office under the roofs where it is however so easy to bang your head. Francesco Zappa-Nardelli proposed the name ℂDuce; I haven't decided yet if I should thank him for it.

My parents are responsible for my existence, and therefore, in a way, for that of this thesis. I never had the opportunity to thank them "formally" for everything they have done for me, for the taste for knowledge and the effort they have transmitted to me, for their encouragement and their love. So I thank them for all of this. Likewise, I thank my friends, and Abbey in particular, for their moral support and everything.

# Foreword

This memoir presents the research work I did in the *Languages* team of the Computer Science Department of École Normale Supérieure de Paris under the guidance of Giuseppe Castagna (DEA intership: April-August 2001; thesis: October 2002 - September 2004). During that time, I also worked at the University of Turin (one week), at Microsoft Research Cambridge (three months), and at the PSD Laboratory in Tokyo (one week); the final writing for this memoir was achieved at project Cristal in INRIA Rocquencourt. The list of publications I contributed to is given below (some of the result obtained are not presented in this memoir).

**Convention** This memoir is written in the first-person plural "we", except for introduction and conclusion chapters, where the first-person singular is used to give a more personal tone.

### Publications

[BCF02] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: a white paper. In *Programming Languages Technologies for XML (PLAN-X)*, 2002.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: An XML-centric general-purpose language. In *ACM International Conference on Functional Programming (ICFP)*, 2003.

[CF04] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Second workshop on Programmable Structured Documents*, 2004.

[DCFGM02] Mariangiola Dezani-Ciancaglini, Alain Frisch, Elio Giovannetti, and Yoko Motohama. The relevance of semantic subtyping. In *Intersection Types and Related Systems (ITRS)*. Electronic Notes in Theoretical Computer Science, 2002.

[Fri01] Alain Frisch. Types récursifs, combinaisons booléennes et fonctions surchargées: application au typage de XML, 2001. Rapport de DEA (Université Paris 7).

[Fri04] Alain Frisch. Regular tree language recognition with static information. In *3rd IFIP International Conference on Theoretical Computer Science (TCS)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop.

[FC04] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 137–146. IEEE Computer Society Press, 2002.

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *Proceedings of the 32st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

# Contents

# List of Figures

# Notations

| Symbol | Description | Section |
|---|---|---|
| $:=$ | Definition of an object or a notation | |
| $\mathcal{P}(X)$ | Set of parts of $X$ | |
| $\mathcal{P}_f(X)$ | Set of finite parts of $X$ | |
| **Set** | Category of sets and total functions | |
| $X+Y$ | Disjoint union of two sets | |
| $\overline{X}^Y$ | Set-theoretic complement of $X$ with respect to $Y$ | |
| $\mathbb{N}$,$\mathbb{Z}$ | Non-negative integers (resp. relative) | |
| $i_{X,Y}$ | Canonical injection $X \subseteq Y$ | |
| $i = 1..n$ | $i \in \{1, \ldots, n\}$ | |
| $\mathbf{Alg}_F$ | Category of $F$ coalgebras | 2.2 |
| $\mathcal{B}$ | "Boolean combinations" functor | 3.1 |
| $\vee$,$\wedge$,$\setminus$,$\neg$ | Boolean operators on types | 3.1 |
| $\sqsupset$ | Socle | 3.3 |
| $\mathbb{T}$ | Type algebra | 3.2 |
| $T$ | Set of type nodes | 3.2 |
| $\widehat{T}$ | Set of type expressions | 3.2 |
| $\tau$ | Description of type nodes $\tau : T \to \widehat{T}$ | 3.2 |
| $T_A$,$T_u$ | Type atoms (resp. type atoms of kind $u$) | 3.2 |
| $\mathbb{B}$ | Base types | 3.3 |
| $\mathcal{C}$ | Constants | 4.1 |
| $\times$,$\to$ | Type constructors | 3.3 |
| $\mathbb{0}, \mathbb{1}$ | Empty type (resp. universal) | 3.1 |
| $\mathbb{1}_{\mathbf{prod}}$ | Universal product type $\mathbb{1}_{\mathbf{prod}} = \mathbb{1} \times \mathbb{1}$ | 6.5.2 |
| $[\![\_]\!]$ | Set-theoretic interpretation | 3.1 |
| $\mathbb{E}[\![\_]\!]$ | Extensional interpretation associated with $[\![\_]\!]$ | 4.2 |
| $\pi(t)$ | Decomposition into a finite union of products | 4.8 |
| $\pi_1[t], \pi_2[t]$ | Type projections | 5.8 |
| $\mathrm{Dom}(t), \rho(t)$ | Decomposition of functional types | 4.8 |
| $G$ | "Pattern" functor | 6.1 |
| $\mathbb{P}$ | Pattern algebra | 6.1 |
| $P$ | Set of pattern nodes | 6.1 |

| Symbol | Description | Section |
|---|---|---|
| $\widehat{P}$ | Set of pattern expressions | 6.1 |
| $\sigma$ | Description of pattern nodes $\sigma P \to \widehat{P}$ | 6.1 |
| $\mathbf{\vert}$,**&** | Pattern constructors (alternation and conjunction) | 6.1 |
| $\mathrm{Var}(p)$ | Capture variables of the pattern $p$ | 6.1 |
| $v/p$ | Result of applying the pattern $p$ to the value $v$ | 6.2 |
| $\Omega$ | Pattern matching failure, type error | 6.2 |
| $\vert, \oplus$ | Combination of pattern matching results | 6.2 |
| $(\!\vert p \vert\!), \lbag p \rbag$ | Set of values accepted by $p$ | 6.3 |
| $\lbag p \rbag$ | Types of values accepted by $p$ | 6.3 |
| $(t /\!\!/ p)(x)$ | Result of applying $p$ to the type $t$ | 6.3 |
| $(t / p)(x)$ | Type of the result of applying $p$ to the type $t$ | 6.3 |
| $\mathcal{E}$ | Set of expressions of the calculus | 5.1 |
| $\mathcal{O}$ | Set of operators | 5.1 |
| **app** | Functional application operator | 5.1 |
| $\mathcal{V}$ | Set of valeurs | 5.4 |
| $o : t_1 \to t_2$ | Axiomatic typing of the $o$ operator | 5.2 |
| $v \overset{o}{\rightsquigarrow} e$ | Semantics of the $o$ operator | 5.5 |
| $(o : t_1 \Longrightarrow t_2)$ | Semantic typing of the $o$ operator | 5.6 |
| $\circledcirc$,$\circledcirc$,$\otimes$ | Constructors and operators on schemas | 5.7 |
| $\{\!\vert \mathbb{t} \vert\!\}$ | Filter defined by the schema $\mathbb{t}$ | 5.7 |
| $\mathcal{F}$ | Boolean formulas | 7.1.1 |
| $p \xrightarrow{t} x$ | Factorization of a capture variable | 8.3 |
| $\mathcal{L}$ | Set of record labels | 9 |
| $\mathcal{L} \xrightarrow{c} Z$ | Quasi-constant functions from $\mathcal{L}$ to $Z$ | 9 |
| $\mathrm{Dom}(r)$ | Domain of a record | 9 |
| $\mathtt{def}(r)$ | Infinitely repeated field of a record | 9 |

# Introduction

# Chapter 1

# Introduction

## 1.1 Motivations, goals

The initial goal of the research work presented in this thesis was to propose a functional programming language adapted to the manipulation of XML [XML] documents, with some general-purpose traits. This work continues the research work started in Hosoya's thesis [Hos01]. It led to the definition of the ℂDuce language and to its implementation. This thesis presents the theoretical semantic and algorithmic bases on which this language is founded.

### 1.1.1 Programming languages

When developing a software application, one of the questions that arise is choice of the programming language. This choice can be guided, among others, by the following considerations:

— Expressive power of the language regarding the domain application: is it easy in this language to express the operations and computations needed by the application?

— Safety: is the language helping the programmer in avoiding programming errors (for example, by using automatic verification, or by encouraging a certain style of programming)?

— Efficiency: are the applications developed in the language fast enough to be used in the way they are intended to be?

These questions are generally not completely independent. Greater expressive power in a language usually comes from high-level constructions, which are very declarative. These constructions allow the programmer to express complex operations in a concise and idiomatic way, without worrying about low-level problems, and to focus on the architecture and correction of their program. These high-level constructions pose interesting compilation and efficiency challenges: their declarative aspect makes necessary the use of an efficient evaluation strategy. Finally, static analysis techniques used to check program safety can give precise information to the compiler to "understand" programs and compile them more efficiently.

This thesis starts from the idea that a domain-specific language, or at least a language built for a specific usage, can manipulate some specific concepts in a more adapted way than an all-purpose language can. A variant of this point

of view consists in saying that when studying what is specific to a domain, one can better comprehend this domain, which can inspire extensions to all-purpose languages, or encourage a certain development method for this domain.

The domain considered in this thesis is the one of applications manipulating XML documents.

### 1.1.2 XML

*Extensible Markup Language* [XML] is a file format designed to represent tree data in a textual form. We can consider XML as a concrete syntax for an abstract tree model. There is no consensus on the exact nature of that type of document, but in a simplified point of view, XML trees are trees of variable arity, whose nodes (called elements) are labelled and possess each a finite set of textual attributes, and whose leafs are characters.

Here is an example of XML document:

```
<animal kind="insect">
 <name>Bumble bee</name>
 <comment>There are about <strong>19</strong> different species
          of bumblebee.</comment>
</animal>
```

In this example, `animal`, `name`, `comment` and `strong` are labels (*tags*). The tags `<...>` and `</...>` signal respectively the beginning and end of an element. One of the conditions of well-formedness of XML documents is the well-parenthesizing of those tags, which allows us to see documents as trees. In the example above, the root is the element labelled `animal`, which possesses two children nodes `name` and `comment`, as well as space characters. The element `comment` possess as children characters and a `strong` element. The root also has an attribute `kind` whose value is the character string `"insect"`. In a well-formed XML document, the same element cannot define the same attribute twice.

The XML specification does not attribute any semantic meaning to the documents, to the labels of their elements, or to their attributes. It simply specifies which characters strings effectively represent an XML tree. By doing so, it encourages a specific abstract document model (trees). An application which is able to represent its data as an XML tree can use XML as an external storage format. Similarly, two applications which need to exchange tree-structured data can use XML as an exchange format. In both cases, applications do not work on any kind of XML documents, instead they both only work on a certain class of them, defined by the labels authorized for their elements, by the possibly nested structures required, by the attributes that need to be defined, by the form of their values, by the possible presence of textual data (characters) in certain places, ...

There are two levels of *correction* that can be distinguished for XML documents:

— a concept of syntactic well-formedness, defined by the XML specification, and which is the necessary and sufficient condition in order to interpret a document as an XML tree;

— a concept of validity regarding a certain amount of constraints on the structure and content of some documents, specific to an application or to a group of applications subjecting themselves to a standard.

A set of validity constraints is called a schema, or XML type. Validity presupposes a well-formedness, since the constraints apply to the abstract model (tree) and not on the textual representation of XML documents. Separating between those two correction concepts allows one to develop generic tools, which work on any type of XML documents (such as *parsers*, editors, request tools, transformations, page layout, ...), and which act at a sufficiently abstract level (trees) in order to express the constraints of schemas. Syntactic questions, such as the way to write comments, to include external fragments, to encode character sets, or to protect special characters, are set up once and for all by the XML specification, and applications that want to use XML can position themselves at the right conceptual level of the abstract model.

Although generic XML tools work, by nature, on arbitrary XML documents, applications which use XML as a storage format or in order to exchange information usually work with one or more XML types. Describing these types in a formal way (or a certain approximation of the constraints put on the documents) allows one to see them as non ambiguous specifications, and, for example, to use validation or edition tools that take XML types into account. There are a few languages that formally describe XML types : DTD (an integral part of the XML specification), XML-Schema [SCH], Relax-NG [REL], ... In a DTD, the labels denoting authorized elements are specified, and for each label, some optional or mandatory attributes are given, and the content is constrained by a regular expression (with a hard constraint of non-ambiguity on regular expressions). In an XML-Schema specification, the content model does not only depend on the labels of the elements, but also on their position (their context) in the tree; it allows one to specify more precise constraints. XML-Schema also possess a notion of "simple type", to specify the form of atomic values in documents (attributes, for example) and the way to interpret them (as an integer, as a date, ...).

### 1.1.3 Programming with XML

When developing an application that manipulates XML documents, there are different conceptual levels which can be used to assess the documents:

— Level 0: the application considers XML documents as text files. *Parsers* operate at this level, verifying the well-formedness constraints of XML documents, and exposing their structure to the rest of the application. It is impossible to do any other interesting treatment at that level without breaking the abstraction of the abstract document model. Hence, a simple text search in an XML document cannot be done at this level, because the same text could be represented in many different ways in an XML document.

— Level 1: the application considers XML documents that have gone through a *parsing* step, but without taking into account XML types. This level provides an interface to access documents and their structure. For example, DOM [DOM] allows us to see XML documents as graphs, to navigate these, and to modify them in-place. The SAX interface exposes to the application a flow of syntactic events encountered in the XML document

(like the beginning or the end of an XML element, or of a text box), that is to say a series of commands that would allow progressive reconstruction of the XML tree associated to the document. However, the application is free to treat this tree as a "stream", without explicitly reconstructing it.

— Level 2: the application handles XML documents while knowing their type. The programming language must be able to understand those types during the execution of the program, either by translating XML types into generalist types (for general-purpose languages), or by taking directly into account the XML types (domain specific languages). The application uses the type system of the language to ensure certain properties on the type of the handled documents.

Usually, an application developed in a classic language can operate on level 0 or 1. If it uses an XML validating *parser*, then it will be able to verify that the inputted documents have the expected type, but this information is then forgotten. Thus the application only sees an ordinary generic XML tree and it cannot take advantage of the fact that this document is well-typed. It is also possible to work on level 2 with a general-purpose language: one has to establish a correspondence between XML types and those of the programming language. It is the *data-binding* approach, which consists in using the constructions of a general-purpose language to represent XML data according to their type. An external tool can for instance take an XML type specification (DTD, XML-Schema, ...), and produce type declarations (or classes) in the programming language. This translation of the types usually introduces more structure in the data than in their XML representation, which limits the flexibility of XML types, and often imposes important approximations. For example, it is possible to specify in a regular expression that a sequence is non-empty (by replacing a Kleene star $*$ with $+$), but if in the language we want to use the same operations to manipulate some potentially empty sequences with some non-empty sequences, then these two categories of objects will have to be represented with the same data type, which means that we will forget one of the constraints specified by the XML type. It is sometimes possible to use one of the particularities of the host language (parametric polymorphism, overloading, ...) to encode more precisely the constraints arising from the XML types.

Another approach consists in defining a new language with a type system conceived from the beginning to represent XML type constraints. It is the approach we take in this thesis. Once the language traits and the type system adapted to XML are sufficiently well-understood, we will consider how to extend an existing language and its type-system to make it benefit from these new characteristics.

Consider a simple scenario for an XML application. This application takes an XML document as input, and does some computations before finally producing a new XML document. The specification of this application gives an XML type $t_1$ for the input documents, and an XML type $t_2$ for the output. It is a contract with the exterior world: the exterior world agrees to provide documents of type $t_1$, and if that is the case, the application commits to producing a document of type $t_2$. In practice, since it is never possible to trust the exterior world, the application will start by verifying that the document received really has type $t_1$, and will raise an error if that is not the case.

If the application is written on level 2, as seen above, it becomes possible to use the type system of the language to verify statically, when compiling the

application, that the transformation verifies that contract (which can be written as $t_1 \rightarrow t_2$). In fact, if the type system of the language is not adapted enough to XML in order to exactly represent the constraints of XML types $t_1$ and $t_2$, then it will be necessary to use an approximation, and to ensure a less binding contract, but it will give us a certain static guarantee anyway.

## 1.2 XDuce

### 1.2.1 Overview

This thesis follows the lead of the research initiated by Hosoya in his thesis [Hos01] and in the XDuce project [HP00, HVP00, HP03]. XDuce is a programming language dedicated to XML, built on a type algebra of regular expressions, close to the classic XML schema languages (DTD, XML-Schema, Relax-NG). It is based on:

— an internal encoding [HVP00] of XML documents that makes it possible to treat typing problems with methods derived from regular tree algorithms;
— a type-system similar to DTD, but in which the content of an element is not only restricted by its label, but also by its position, and that does away with the determinism restriction of DTD;
— an interpretation of types as sets of documents, leading to a natural definition of subtyping $\leq$ as the inclusion of such denoted sets (this is semantic – or set-theoretic – subtyping);
— a *pattern-matching* operation [HP01, HP02], with a pattern algebra adapted to the extraction of information from XML documents.

XDuce is a functional language, as programs are essentially *expressions* to evaluate. The results of the computations are *values* which, in this case, are XML trees.

In XDuce, types are interpreted as *sets* of values. This vision of types is not usual in the world of programming languages, where types are usually considered as a specification of the form of values, of their physical representation, and of their behavior. However it is very natural in the context of XML: indeed, types represent constraints on the structure of documents, but documents exist independently from their types, and some generic operations have a meaning on any XML type. Therefore, XML types are only there to specify, sometimes partially, the structure of XML documents. One has to be able to see the same document under different types, as it only consists in remembering more or less constraints about it. A consequence of this vision is a set-theoretic subtyping relation: a type $t_1$ is said to be a subtype of $t_2$ if the set of values of type $t_1$ is included in the set of values of type $t_2$, or, in other words, if the constraints expressed by $t_1$ are at least as strong as the constraints expressed by $t_2$. The type-system of the language is therefore built in such a way that the programmer can use an *expression* of type $t_1$ in any case where an expression of type $t_2$ is expected. This is done in a transparent way, without having to explicitly introduce a type coercion.

In particular, all the the equational theory of regular expressions can be used in a transparent way. A value of type `A+` can freely be seen as a value of type `A A*` or of type `A* A`, meaning that given a non-empty sequence, it is possible to isolate either its first element, or its last element. These three types

are equivalent, and a function defined on one of them can be applied on the two others.

The computation model of XDuce is functional: a program is a set of mutually recursive functions, each of them being defined with pattern-matching operations, inspired from ML. In the same way that in ML pattern-matching constructors correspond to the constructors of types and values, in XDuce, patterns are a generalization of types: they are regular expressions, with capture variables. This promotes a programming style in which applications are structured around the type of XML data that they must treat. Recursive functions correspond to the definitions of recursive XML types, and every pattern-match can inspect a sub-tree (its roots, its children, ...). Pattern-matching is an example of those declarative high-level operations, which I was mentioning in Section 1.1.1.

This thesis takes the ideas of XDuce, solves some of the open problems mentioned in Hosoya's thesis, and proposes a new language, called ℂDuce, which extends XDuce in a certain number of ways. I will not talk a lot about XML in this thesis. The link between XML and ℂDuce is the same that exists between XML and XDuce, and I invite the interested reader to review the work on XDuce.

### 1.2.2 Internal encoding

XDuce works with two representations for XML documents. The external representation represents documents as trees of arbitrary arity; the internal representation uses binary trees, which is more pleasant on a technical level (see also [MSV00]). The latter is defined by the following syntax (we consider a set of element labels, and we denote by $l$ a generic element of this set):

$$v ::= \epsilon \mid l(v, v)$$

These objects represent *sequences* of elements; the empty sequence is denoted by $\epsilon$, and $l(v_1, v_2)$ is the sequence which starts by an element of *tag* $l$, of content $v_1$, followed by the sequence $v_2$.

For example, the document:

```
<person>
 <name></name>
 <email></email>
</person>
```

is encoded in the internal representation by:

$$\texttt{person}(\texttt{name}(\epsilon, \texttt{email}(\epsilon, \epsilon)), \epsilon)$$

Types are expressions that represent sets of element sequences. Types also have two representations in XDuce. In the external form, the syntax of types is:

| | | |
|---|---|---|
| `T` | ::= | |
| | `()` | empty sequence |
| \| | `X` | declared type |
| \| | `l[T]` | XML element |
| \| | `T,T` | concatenation |
| \| | `T\|T` | union |

For every type name `X`, we assume that a declaration of the following form was given:

$$\texttt{type X} = \texttt{T}$$

The interpretation of these types as sets of sequences is clear, and it is possible to define the Kleene star as syntactic sugar in that representation.

For the internal representation, we use binary tree automata. We denote by $X, X_0, X_1, \ldots$ the states, and assume that for each of them we are given a description of its transitions, which is an *internal type*. The types of the internal algebra are defined by the following productions:

| | | | |
|---|---|---|---|
| $T$ | ::= | | |
| | | $\emptyset$ | empty type |
| | $\mid$ | $\epsilon$ | empty sequence singleton |
| | $\mid$ | $l(X, X)$ | XML element |
| | $\mid$ | $T \mid T$ | union |

For example, consider the automaton with states $X_0, X_1, X_2$, defined by:

$$\begin{array}{lcl} M(X_0) & = & \epsilon \\ M(X_1) & = & \texttt{name}(X_0, X_2) \\ M(X_2) & = & \texttt{email}(X_0, X_2) \mid \epsilon \end{array}$$

The type $\texttt{person}(X_1, X_0)$ represents sequences made up of a single element `person`, which has for children exactly one empty element `name`, followed by any number of empty elements `email`. In other words, this type corresponds to the `person` element defined by the DTD:

```
<!ELEMENT person (name,email*)>
<!ELEMENT name   EMPTY>
<!ELEMENT email  EMPTY>
```

The internal representation of the documents and types of XDuce is quite specific to XML, in which labels play a particular role. In fact, the internal representation only uses well-known concepts of type theory: singleton base types (for the labels and the empty sequence), product types (for the tree structure), recursive types, and union types (to encode regular expressions). We could reinterpret the underlying theory of XDuce in terms of product types, union types, and recursive types. Instead of writing $l(X_1, X_2)$, we would write $l \times X_1 \times X_2$, where $l$ is a singleton base type, and $\times$ is the type constructor of the cartesian product. The type example above would be written:

$$\texttt{person} \times (\texttt{name} \times \epsilon \times (\mu\alpha.(\texttt{email} \times \epsilon \times \alpha) \vee \epsilon)) \times \epsilon$$

where the notation $\mu\alpha.\ldots$ introduces a recursive type.

$\mathbb{C}$Duce works directly at the level of the internal representation, and uses general-purpose constructions, like the cartesian product type constructor, or the record type (to represent the XML attributes). XML notations are only seen as syntactic sugar. Section 10.4 presents the encoding of XML in $\mathbb{C}$Duce.

## 1.3 Quick overview of the thesis

### 1.3.1 Theoretical and semantic foundations

One of the major contributions of ℂDuce compared to XDuce is the addition of first-class functions, and of an arrow type constructor $\rightarrow$, without stratifying the types so that arrow types and XML types can freely be mixed. I have chosen to work directly on the level of the internal representation of types in XDuce, in which XML types are represented by union types, product types and recursive types.

This led me to study a $\lambda$-calculus with arrow types, product types, union types and recursive types. The main difficulty was to preserve the set-theoretic approach to define the subtyping. This set-theoretic approach, which consists in defining subtyping as set inclusion, allows us to carry out in a comfortable way the meta-theoretical study of the system. In order to do that, we adopt a purely set-theoretic model notion.

In order to add a pattern-matching operation inspired from the one of XDuce, which can perform type checks during execution, this calculus has a type-check operation. The semantics is therefore driven by types, and in order to express this at the type level, the calculus and the type algebra implement overloaded functions. Intuitively, if we interpret the type $t\rightarrow s$ set-theoretically as the set of functions whose result is of type $s$ as soon as their argument is of type $t$, then it is natural to define the overloading of functions by a simple intersection. A function of type $(t_1\rightarrow s_1)\wedge(t_2\rightarrow s_2)$ is simply an overloaded function, which verifies both the type constraint $(t_1\rightarrow s_1)$ and the type constraint $(t_2\rightarrow s_2)$. If we are going to reason set-theoretically, by introducing intersection for arrow types, then it seems natural to try to work uniformly with arbitrary boolean combinations (union, intersection, difference) of all type constructors. In XDuce, the intersection and difference operators are used internally in the type inference algorithm for pattern-matching; exposing them in the type algebra also simplifies the presentation of this algorithm.

The first four technical chapters of this thesis study this type algebra and the corresponding calculus. This study constitutes the theoretical foundations of the ℂDuce language.

Chapter 2 introduces some abstract formalism to manipulate recursive type algebras (or other kinds of recursive objects). This formalism allows us to get rid of implementation and optimization details (such as the more or less advanced sharing of cyclical structures) and cumbersome presentation specificities (syntactic representation with recursive binders, and the associated renaming problems; environments with global recursive definitions that have to be treated in every statement).

Chaper 3 defines the type algebra, using a non-syntactic representation. Recursion is handled with the formalism of Chapter 2, and boolean combinations are represented using a form that allows us to give a pleasant presentation of the algorithms. This representation is chosen so that every finite set of types can be saturated into a finite set stable by boolean combination and by decomposition of the constructors (arrow and product).

Chapter 4 tackles the definition of a set-theoretic subtyping relation for the type algebra. The challenge posed by higher-order functions comes from the circularity introduced between the type system, the subtyping relation, and the

set-theoretic definition of types. In XDuce, it is possible to directly give a set-theoretic semantic for types as value sets, because types are nothing more than tree automata. Subtyping then simply corresponds to inclusion between regular tree languages, and we can use it to type the calculus itself. In ℂDuce, the values and expressions of the calculus are closely linked; indeed, functions are values, but in order to know if a function is in the interpretation of a type, it is necessary to have the type system in its entirety (in order to be able to type the body of the function).

I remove that circularity by introducing a set-theoretic model notion for the type algebra, allowing us to consider set-theoretic interpretations before even introducing the calculus (and its type system). This model definition captures the intuition behind the behavior of functions (in fact, of arrow types), without specifying the concrete nature of the model elements. In order to be a model, a set-theoretic interpretation must capture the semantic properties of the calculus that are important for typing. All reasoning about the subtyping relation is done semantically, by working with models, and it allows us to separate the algorithmic aspects tied to the computation of this relation with its use in the type system. In particular, the meta-theoretical analysis of the subtyping relation and its associated operators can be done without considering the very complex rules of the subtyping algorithm.

Chapter 5 finally introduces the calculus itself, its type-system (which uses the subtyping relation), and its semantics (which is type-driven). Apart from the usual type safety results (type preservation through reduction, and no type errors during execution), the main theorem (Theorem 5.6) is that the set of values of the calculus is indeed a model, for the definition of Chapter 4, and that it induces the same subtyping relation as the one that was used to define the type-system. In other words, we have come full cycle, as the subtyping relation corresponds indeed *a posteriori* to inclusion between sets of values denoted by types.

Chapter 6 adds to the calculus a pattern-matching operation, which comes from the corresponding operation in XDuce. In fact, this operation generalizes the dynamic type-check present in the calculus, and does not fundamentally change the typing of the calculus or the properties of its semantics. The patterns are, like types, recursive objects, and the formalism of Chapter 2 is used again in order to define the pattern algebra. The pattern algebra of XDuce is extended with conjunction or intersection patterns (which do not change the expressive power, but allow us to write more compact patterns), and with constant patterns (which accept any value and output a constant binding for a specific variable). Furthermore, the linearity condition on capture variables is softened, which allows us to encode capture variables for sub-sequences (inside a regular expression) which can be repeated, meaning that they can appear multiple times or under a Kleene star; the semantics then consists in concatenating all the captured sub-sequences. I obtain an exact typing algorithm, including for sub-sequence capture variables in non-terminal position, which solves an open problem from Hosoya's thesis (which he has solved independently since then, with a similar technique, see Section 1.4.2).

### 1.3.2 Algorithmic aspects

One of the major contributions of project XDuce is the development of an efficient and practical subtyping algorithm, even though it corresponds to the algorithmically complex problem of inclusion between regular tree languages.

Chapter 7 studies the algorithmic aspect of the subtyping relation introduced in Chapter 4, for a certain class of universal models (those that induce the largest subtyping relation). This subtyping relation can be seen as a generalization of the one of XDuce. In fact, the subtyping algorithm has been prepared in Chapter 4, where the subtyping relation in universal models is characterized using coinduction (via a concept of simulation). The complementary of this relation is therefore simply an inductive predicate, and we present two algorithms that can autonomously compute an inductive predicate defined by logical formulas. The first uses the same principle used in the algorithm of XDuce, based on *memoization*, which is usual in subtyping algorithms with recursive types. I give another algorithm, which completely avoids any backtracking, by keeping trace of the dependencies between assumptions that are not yet surely verified, and deducted consequences. The subtyping algorithm itself is obtained by using either one of these algorithms, and logical formulas that are directly inspired from the simulation concept of Chapter 4. I also present alternative formulas, and optimizations. Having presented the computation of inductive predicates as a resolution engine separate from logical formulas allows us to decouple the problems and to avoid doing redundant proofs when considering those variants.

Chapter 8 considers the efficient evaluation of pattern-matching. Because of the recursivity of patterns and types, the pattern matching operation is more akin to the evaluation of a tree automaton than to ML-style pattern matching which only extracts subcomponents from values of finite depth. I work at the compilation level, where it is acceptable to carry out possibly costly calculations on the patterns to prepare for an efficient evaluation of these during execution. I present a fairly general framework to express several evaluation and optimization strategies. I am particularly interested in optimizations made possible by type information gained on pattern matched values, given by the static type system of the language.

### 1.3.3 The ℂDuce language

The ℂDuce language is defined on top of the calculus of Chapter 5. I do not give the concrete syntax of the language, or even an exhaustive description. I only indicate how the main features of the language can be expressed as part of the calculus.

Chapter 9 shows how to extend the calculus, the type algebra and the pattern algebra with extensible records. These records are used in ℂDuce to encode the attribute sets of an XML element. It is of course necessary to extend the set-theoretic approach for subtyping, and typing the concatenation operator gives rise to a non-trivial semantic development, in order to semantically and accurately characterize the approximation that is done in its typing.

Chapter 10 briefly presents the language, in particular how the types and regular expression patterns are encoded in the corresponding algebra of the calculus.

Chapter 11 presents some more practical aspects of the implementation of

ℂDuce, such as how to implement the type-checker to get localized error messages, techniques for representing values during execution in order to optimize some operations, variants for representing boolean combinations internally in the type-checker, or an interface system (which preserves the types) with the Objective Caml language.

## 1.4 Main contributions

The work on XDuce is the starting point of the research presented in this thesis. I have offered solutions to some open problems presented in Hosoya's thesis, as well as extensions of the theory.

### 1.4.1 Higher-order functions and set-theoretic subtyping

This thesis started with the observation of the absence of first-class functions in XDuce, even though the language looks roughly like a functional language of the ML family. Having first-class functions, which can be passed as arguments to other functions, returned as a result, or stored in data structures would make it possible, for example, to define XML transformations parameterized by elementary transformations on certain parts of the documents, or to factorize similar transformations. For example, if `AddrBook` is an XML schema used to represent a address book, made up of elements of type `Person` and elements of type `Company`, then we may want to write a generic layout function `(Person→Html)→(Company→Html)→AddrBook→Html`, where `Html` is the type of XHTML document fragments. This generic function takes as an argument two layout functions for basic entries, and outputs a layout function for address books. Therefore we can reuse this generic function to create multiple different presentations.

In addition to this use of higher-order functions within the framework of XML applications, it seems interesting, from a theoretical point of view, to understand how the set-theoretic type interpretation of XDuce and the type interpretation of the $\lambda$-calculus interact.

We have chosen to completely integrate arrow types into the type algebra, which makes it possible to use boolean combinations of arrow types. Other works have taken a lighter approach to integrate the regular expression types of XDuce into a language with higher-order functions or objects. The XHaskell language [LS04b] is an extension of Haskell with the regular expression types of XDuce. The type algebra is stratified: arrow types cannot appear inside of regular expressions, which makes it possible to define subtyping for these arrow types in a purely axiomatic way. The Xtatic language [GP03] extends the object-oriented language C$^{\#}$ with regular expression types. In Xtatic, regular expression types and object types (classes) are stratified, but mutually recursive. However, the definition of the subtyping relation remains easy as the typing of objects is done by nominal typing, with an explicitly declared subtyping relation (inheritance).

Damm [Dam94] studies a type algebra with union, intersection and recursive types, and product and arrow constructors. This algebra is very close to ours, but it has no set difference operator (which we need to precisely type the pattern matching operation). Damm also uses a semantic approach, by introducing two

type interpretations: a semantic interpretation of types in a model with ideals (ideals are stable by union and intersection, but not by complement), and an encoding of types as regular tree languages. By relating these two type interpretations, he reduces the subtyping problem (on the semantic side) to an inclusion between regular tree languages. This approach uses denotational semantics tools (domains, complete metric spaces), whereas the approach presented in this thesis is purely set-theoretic and fully compatible with an operational semantics of the calculus. Insofar as the semantics of ℂDuce is type-driven, it is not obvious to see how the denotational approach of Damm could be used in this context. Indeed, there are several different subtyping relations that can be deduced from models; the calculi built on these relations will have different semantics, and we can therefore clearly see that it is not possible to start, as Damm does, from an *a priori* semantics to define the subtyping relation.

In order to reduce the problem to regular languages, Damm encodes an arrow type as a set of sequences that represent all the possible graphs of finite approximations of functions of such a type. We avoid this encoding and these approximations by directly interpreting an arrow type as a set of graphs (Definition 4.2). This more direct interpretation makes it possible, using simple set-theoretic computations, to deduce the rule of the subtyping algorithm for the case of boolean combinations of arrow types (in a universal model). This allows us to extend the subtyping algorithm of XDuce, which has proven its effectiveness.

Vouillon and MelliÃ¨s [VM04] present a generalization of the ideal model for polymorphic recursive types. Their goal is to interpret types as sets of terms of their calculus (and not as sets of values), i.e. to classify terms according to their types. They are not interested in the effective study of the subtyping relation. The union type operator is an approximation (closure) with regard to set-theoretc union. A more in-depth study of the links between the system of Vouillon and MelliÃ¨s and the one presented in this thesis remains to be carried out.

### 1.4.2 Typing pattern matching

The type inference algorithm for pattern matching, presented in Hosoya's thesis [Hos01], does not give an exact type for sequence capture variables that are not in terminal position in a regular expression pattern. This comes from the translation of capture variables from the external syntax (regular expressions) to the internal pattern algebra (automata). In the translation of XDuce, one sequence capture variable gives rise to two variables in the internal representation, in order to represent the beginning and the end of the captured sub-sequence. The dependency between these two variables is lost, and it forces us to infer a type that is possibly too large for the capture variable. I solved this problem by adopting a different translation of sequence capture variables in regular expressions: they are now propagated on all the elements they capture (and no longer just a start and end mark) in the regular expression, by using the same variable name. This preserves the dependency between all these variables. The semantics of internal patterns is defined so that this translation provides the right capture semantics: when we apply a pattern $(q_1, q_2)$ to a value $(v_1, v_2)$, and each $q_i$ captures a value $v'_i$ for the same variable $x$, the result for $x$ in the pattern $(q_1, q_2)$ is $(v'_1, v'_2)$.

Hosoya has solved this problem independently. He proposes another semantics [Hos03] for patterns, which rejects ambiguous patterns (which simplifies things by avoiding to introduce differences in order to take into account the *first-match* policy of the alternation |). The inference algorithm he proposes in this case is exact (for non-ambiguous patterns). The technique consists in using a concept of automaton with transitions annotated by capture variables (that register elements activating these transitions). The translation of regular expression patterns to these automata propagate, as I do, the capture variables up to the transitions. Therefore the two solutions are very similar, and the one of Hosoya could be easily adapted to ambiguous patterns and to the *first-match* policy.

### 1.4.3 Efficient compilation of pattern-matching

Hosoya's thesis evokes an optimization technique for pattern evaluation, which makes it possible in some cases to only consider the label of an XML element instead of considering all the corresponding sub-trees, by using the information provided by the type system of the language. I present in this thesis some efficient techniques for pattern evaluation, that make it possible to express those kind of optimizations. The formalism introduced allows us to precisely keep track of type information (provided by the static system, or obtained during computation), and to express parallel computations, in order to avoid multiple passes on the matched value.

Levin and Pierce [Lev03, LP04] have studied in parallel some similar efficient evaluation methods for patterns in XDuce-like languages, by taking into account type information. They introduce a concept of *matching automata* that could serve as a target language for the compilation algorithm I present here. Their optimization algorithm, driven by types [LP04], is not described formally, hence it is difficult to compare it to the one I propose.

In previous work [Fri04], I presented an optimization algorithm for pattern matching without capture variables. The target language (non-uniform automata) imposes a sequential left-right traversal order for trees, and the algorithm corresponds to the left-right strategy presented in this thesis (Section 8.3.8). In addition to that, it has an invariance property when patterns (in fact, types, since there are no variables) are replaced with equivalent patterns; this comes from a canonical decomposition of types in the form of a minimal union of product types whose first components are disjoint. This ingredient could be added to the formalism in this thesis if we modified the normal form of patterns (Section 6.5.2) to use this canonical form, at least for types (to treat capture patterns, in general, we would need to study pattern equivalence, and I have not studied this problem; but we can settle with the detection of some equivalences).

### 1.4.4 The subtyping algorithm

Amadia and Cardelli [AC93] have proposed a subtyping algorithm in a system with arrow types and recursive types. This algorithm is a variant of the one without recursive types; it operates by keeping track (*memoization*) of type pairs encountered in the recursive descent. This method is used in most work on recursive types to ensure the termination of algorithms. The coinductive

aspect of Amadio and Cardelli's algorithm was highlighted by Brandt and Henglein [BH97]. Kozen, Palsbergy and Schwartzbachy [KPMI95] propose a method which uses finite automata to optimize the complexity of Amadio and Cardelli's algorithm. Their base ascertainment is that if we consider the product constructor $\times$ (covariant in both of his arguments) instead of the arrow constructor (contravariant in its first argument), then Amadio and Cardelli's algorithm is in fact a classic algorithm for deciding inclusion between finite automata. To take into account the contravariance of the arrow, this algorithm needs to be adapted to inverse the order of the arguments on the contravariant side. Gapeyev et al. [GLP00] do a review of these works.

XDuce's subtyping algorithm can be seen as an extension of Amadio and Cardelli's algorithm. It continues the coinductive approach, implemented by a set of assumptions being verified, that needs to be saturated in order to reveal possible contradictions. Because of the union type connector, the algorithm of XDuce cannot simply monotonically saturate this set of assumptions. Indeed, this algorithm, which we present with a set of rules, may have to check several different derivations in order to prove a given goal (it suffices that one of them succeeds). If the algorithm tries a branch that fails, it has to be able to do *backtracking* in order to put the set of assumptions back to its previous state, because some assumptions which were introduced in the failing branch have not been verified (in fact, at least one of them is false).

I developed an algorithm (Chapter 7) that avoids this backtracking. I chose to separate the presentation of the rules that define the subtyping relation itself, and, that of the resolution engine itself, which is independent from those rules. The algorithm without backtracking can be applied to any inductive relation (or coinductive, like subtyping, by considering the negation). For example, the capture factorization algorithm (Section 8.4) also fits into this frame. The algorithm without backtracking is in fact quite close in spirit to the algorithms proposed by Dowling and Gallier [DG84] to test the satisfiability of the Horn formula in linear time.

Tadahiro and Hosoya [SH04] have tackled the same problem in paralell (getting rid of backtracking in the subtyping algorithm of XDuce). Their solution, like mine, consists in keeping track of the dependencies between the assumptions being verified, and their consequences. A precise comparison between the two algorithms remains to be done.

### 1.4.5 XML attributes

When the work on this thesis started, XDuce did not handle XML attributes. I introduced extensible records in $\mathbb{C}$Duce (Chapter 9) in order to close this gap. The version presented in this thesis is slightly more expressive than the one implemented in $\mathbb{C}$Duce, as it makes it possible to constrain the type of all unspecified fields (the implemented version is only able to allow or forbid the presence of other fields than those specified). It differs also in that the absence of a field is encoded by a particular value, which can be manipulated and captured by pattern-matching. In the implementation, this value is hidden from the programmer.

In parallel, Hosoya and Murata [HM02, HM03] developed an alternative solution to add attributes to XDuce, which consisted in adding attribute specifications to the regular expressions. In that system, we can easily express

that an information is present exclusively either in a sub-element, or in an attribute, as well as many other similar element-attribute dependencies. These dependencies can be "developed", in order to separate the constraints on sub-elements from constraints on the attributes, by listing all the possible cases, at the cost of a growth exponential in the size of types. Hosoya and Murata developed specific algorithms in order to compute the subtyping relation and boolean combinations, which avoid this combinatorial explosion in practice. Thus, even though the expressive power of the two approaches is similar, the one of Hosoya and Murata has a clear algorithmic advantage. The one proposed in this thesis, based on records, does enjoy the advantage of simplicity; it is completely orthogonal to the rest of the system (records are in fact a generalization of cartesian products), and contrarily to the approach of Hosoya and Murata, it does not put into question, for example, the techniques for efficient compilation of pattern-matching, and it does not significantly complicate the subtyping or pattern-matching algorithms.

### 1.4.6 A calculus for overloaded functions

In XDuce and ℂDuce, computations are done essentially by pattern-matching: it is the only way to do computations on the form of objects. Each branch is given by a pattern and a body. The pattern verifies that the argument is of a certain type, and also extracts some sub-objects from the argument, which will be used in the body for computations. In fact, to each pattern $p$, corresponds a type $⌊p⌋$ which represents the set of all values accepted by the pattern: a value if accepted if and only if it is of type $⌊p⌋$. In other words, each branch handles a given type, and branch selection can be done using only the type of the pattern matched value. In case of ambiguity, the usual choice in programming languages is to chose the first branch that matches: this is what is done in ML-like languages, in XDuce, and in ℂDuce.

Pattern-matching is quite close, operationally, to calculi based on overloaded functions with late binding, which are also driven-by-type computations. In the $\lambda\&$-calculus [CGL95], an overloaded function is defined by several branches, which are "simple" functions. When we apply it to a value, a selection mechanism (*dispatch*) chooses the adequate branch. When several branches match, the most precise one, whose domain is the smallest, is chosen, and the typechecker ensures that such a branch exists.

The differences in the selection mechanism when several branches fit is not essential to our subject.

However, there is a fundamental difference between pattern-matching in XDuce, and overloading in a $\lambda\&$-calculus. With pattern-matching, all branches must give a result of the same type, whereas with overloading it is possible to have branches that operate on disjoint domains; in that case, the results of these branches can have different types, and this (more precise) type information can be exploited to type the rest of the program. In that sense, pattern-matching is less flexible.

In ℂDuce, overloaded functions keep this kind of information in the type of functions. There are no new type constructors, and the overloading is simply represented by an intersection type. The type $(t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n)$ denotes the functions that simultaneously have all the types $t_i \to s_i$. The type intersection can be interpreted as a conjunction of constraints (and therefore, as the

intersection of the corresponding sets).

Let us give a very simple example of overloaded functions. Suppose that a company receives from its customers XML documents that are sequences of orders, and that there are two types of orders $t_1$, $t_2$ (for example, for different types of products). The company sends back a receipt which acknowledges every order and adds information depending on the type. When responding to an order of type $t_i$, it responds with an XML fragment of type $s_i$. Without overloaded functions, the processing function would have the type $(t_1 \vee t_2)* \rightarrow (s_1 \vee s_2)*$ (here we freely use the Kleene star of regular expressions). It proceeds by simply iterating a function that treats individual orders, of type $(t_1 \vee t_2) \rightarrow (s_1 \vee s_2)$. In fact, it is an overloaded function, of type: $(t_1 \rightarrow s_1) \wedge (t_2 \rightarrow s_2)$, hence we can give to the general process function (without rewriting its code) the more precise type: $((t_1 \vee t_2)* \rightarrow (s_1 \vee s_2)*) \wedge (t_1* \rightarrow s_1*) \wedge (t_2* \rightarrow s_2*)$. Therefore, if a client only sends orders of type $t_1$, it implies that the order form is of type $t_1*$, and he can expect to receive an acknowledgement of type $s_1*$; and a client which sends orders of both types should receive answers of type $(s_1 \vee s_2)*$. If we add a new type of orders, it is enough to just modify the code of the order processing function so that it can process a new type (we also change the interface of functions in order to express the new types being handled).

We see that overloaded functions make it possible to work simultaneously on documents which have more or less different types, without losing information. This avoids having to duplicate code.

Let us go back for a moment to the $\lambda\&$-calculus. In this theory of overloaded functions, one of the basic well-formedness conditions off overloaded functions is the following. If an overloaded function has two branches of types $t_1 \rightarrow s_1$ and $t_2 \rightarrow s_2$, then: $t_1 \leq t_2 \Rightarrow s_1 \leq s_2$. To understand this condition, it must be recalled that the most precise type of a program decreases during evaluation. If, statically, we attribute to an expression the type $t_2$ and hence the result of the overloaded function applied to this expression has type $s_2$, then it is possible that at execution we realize that the expression has in fact the more precise type $t_1$, therefore it is branch $t_1 \rightarrow s_1$ which is being used, giving a result of type $s_1$. This requires, for type safety reasons, that $s_1 \leq s_2$.

In $\mathbb{C}$Duce, there are no conditions restricting the arrow types in the interface of a function. Consider the two following expressions:

$$\begin{aligned} e_0 &= \mu f(t_1 \rightarrow s_1; t_2 \rightarrow s_2).\lambda x.e \\ e_0' &= \mu f(t_1 \rightarrow (s_1 \wedge s_2); t_2 \rightarrow s_2).\lambda x.e \end{aligned}$$

Suppose $t_1 \leq t_2$. Function $e_0'$ verifies the $\lambda\&$-calculus condition, since $s_1 \wedge s_2 \leq s_2$, but it is not necessarily the case for $e_0$ (if $s_1 \not\leq s_2$). With the type system of Chapter 5, however, one of the expressions $e_0$,$e_0'$ is well-typed if and only if the other one is also well-typed, and then they have the exact same types. In general, it is always possible to rewrite the interface of an abstraction so that the covariance condition of the $\lambda\&$-calculus is verified. The theory we developed allows us to meet the condition posed *a priori* in the $\lambda\&$-calculus; in some sense, it gives a semantic justification to this condition.

# Part I

# Theoretic and semantic foundations

# Chapter 2

# A formal framework for recursive syntaxes

In this chapter, we will use elementary notions of category theory [AL91] which we do not recall here. We denote by **Set** the category of sets and total functions. The disjoint sum of two sets $X$ and $Y$ is written $X + Y$. If $X$ is a subset of $Y$, we write $i_{X,Y}$ for the canonical inclusion $X \hookrightarrow Y$. The identity is written $\mathrm{Id}_X$.

## 2.1 Motivation

**Inductive syntax** The set of types of a logical system or of a type system is usually introduced as an inductive term algebra. For example, the type algebra $T$ for a $\lambda$-calculus with a unique base type $\iota$ can be defined by the following syntax:

$$t \quad ::= \quad \iota \mid t \rightarrow t$$

This presentation makes it possible to define operators and relations by induction on the syntax of types. From a categorical point of view, this construction can be seen as follows. Let **Set** be the category of sets and $F$ the functor **Set** $\rightarrow$ **Set** which associates to a set $X$ the set of terms of the form $\iota$ or $x \rightarrow y$, with $x, y \in X$. An $F$-algebra is a pair $(X, f)$ where $X$ is a set and $f$ a function $FX \rightarrow X$. The term algebra $T$ can be seen as an $F$-algebra, its function $f$ being a bijection often left implicit. The class of $F$-algebras has a natural category structure, in which the type algebra $T$ is an initial object. It would therefore be possible to *define* the type algebra as the initial $F$-algebra (which characterizes it, up to isomorphism, in the category of $F$-algebras).

The functor $F$ represents the *signature* of the type algebra. From an arbitrary functor **Set** $\rightarrow$ **Set**, we can try to formally build an initial $F$-algebra. The classic technique consists in starting from the function $\emptyset \rightarrow F\emptyset$ and iterating the functor, thus giving rise to a diagram $\emptyset \rightarrow F\emptyset \rightarrow F^2\emptyset \rightarrow \ldots$, whose inductive limit can be considered. If the functor commutes with this limit (which is guaranteed if the functor is $\omega$-continuous), then the limit can be made into an initial $F$-algebra.

It is quite possible to take as signature a less "syntactic[1]" functor than the example given in introduction, like for example the $\omega$-continuous functor $\mathcal{P}_f$ : **Set** $\to$ **Set** which associates to a set the set of all its finite subsets.

**Recursive syntax** Let us consider now the case of recursive types. A classic presentation consists in introducing recursion variables $\alpha$ and binders in the syntax:

$$t \quad ::= \quad \iota \mid t \to t \mid \alpha \mid \mu\alpha.t$$

This syntax is quite arbitrary. We could indeed imagine using this other syntax which makes it possible to better share common sub-terms:

$$t \quad ::= \quad \iota \mid t \to t \mid \alpha \mid t \text{ where } \{\alpha_1 = t_1; \ldots; \alpha_n = t_n\}$$

In addition, we have to take into account:

— a syntactic restriction, in order to only authorize terms without free variables and without ill-formed recursions (that is, a variable must be separated from its binder by at least one constructor), and

— a contextual equivalence which stems from $\alpha$-conversion and *unfolding*: $\mu\alpha.t = t[\alpha \mapsto \mu\alpha.t]$ (which might be interpreted in a coinductive way).

Having a syntactic presentation of types also suggests describing the algorithms in inductive form. For example, subtyping algorithms with recursive types are presented through a system of rules, with a memoization environment to ensure their termination. Hence we have to be able to efficiently test type equality, and this can cause implementation problems.

These problems are further exacerbated if we want to introduce operators in the type algebra, such as a commutative and associative union operator. Indeed, equivalence tests can become very costly. A simple solution consists in not using any equivalence concept, and working with the syntax itself, by explicitly taking into account unfolding and other potential equivalences in the rules and in the algorithms. But we then lose the possibility of using these equivalences in the theoretical study of the system, as well as in the implementation (to increase the sharing of structures in memory).

This kind of syntactic approach tries to give an inductive nature (terms) to objects that are fundamentally cyclical. Algorithms that deal with recursive types usually work by *decomposing* types. Even in the case of an infinite unfolding, we have the guarantee of encountering only a finite number of different types, and this property ensures the termination of the algorithms via memoization techniques.

It seems more natural for us to use to try to formalize recursive types with coinduction, and to develop the theory of this formalization. The inductive syntax only has a superficial role, and can be treated at the *parser* level.

A first idea is to simply take the categorical dual of the initial algebra construction. For a given functor $F$, an $F$-coalgebra is a pair $(X, \delta)$ where $X$ is a set and $\delta : X \to FX$ is a function. While an $F$-algebra represents the way of constructing objects, an $F$-coalgebra is more about destructors, that is, the way of seeing an object from $X$ as an expression built on atoms from $X$. The class

[1] By syntactic functor, we mean a functor that associates to a set $X$ a set of "terms" built over $X$.

of $F$-coalgebras has a natural category structure, and we can define the natural concept of final $F$-coalgebra. We define it by iterating from a final object $\bullet$ in **Set** (i.e. a singleton) using the unique function $F\bullet \to \bullet$ and taking a projective limit.

Let us consider again the functor $F$ : **Set** $\to$ **Set** which associates to a set $X$ the terms of form $\iota$ and $x \to y$ with $x, y \in X$. The final $F$-coalgebra can be described as the set of potentially infinite binary trees, with leafs $\iota$ and internal nodes $\to$. These objects are not finitely representable. We would like to restrict our attention to regular trees, namely those that only have a finite number of different sub-trees, and that can be regarded as an infinite unfolding of recursive types. These regular trees can be concretely represented with graphs, but this is not a unique representation: several graphs represent the same regular tree. With the example functor $F$ above, it is easy to test for equality in this representation, but that would not be the case with more complex functors, for example ones that make use of the operator $\mathcal{P}_f(\_)$.

The idea is then to say that it is not necessary to have such a strong equality concept. It is not necessary to identify all the graphs that represent the same regular tree in order to ensure the termination of algorithms. We can be satisfied with a weaker concept of equality; it is enough indeed to identify enough types so that, starting from a type that we iteratively deconstruct, we only encounter a finite number of types that are different for this concept of equality. Our approach consists in axiomatizing this property, which makes it possible to develop the theory while leaving more flexibility to the implementation of the algebra.

## 2.2 The category of $F$-coalgebras

**Definition 2.1 ($F$-coalgebra)** *Let $F$ : **Set** $\to$ **Set** be a set-theoretic endofunctor. An **$F$-coalgebra** is a pair $\mathbb{T} = (T, \tau)$ where $T$ is a set and $\tau$ is a function $T \to FT$. We say that $T$ is the **support** of $\mathbb{T}$.*

To simplify, we will simply talk of coalgebras from now on. It is understood that all definitions relate to a certain functor $F$.

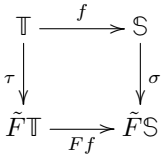
**Definition 2.2 (Category of coalgebras)** *Let $\mathbb{T} = (T, \tau)$ and $\mathbb{S} = (S, \sigma)$ be two coalgebras. A **morphism** $\mathbb{T} \to \mathbb{S}$ is given by a set-theoretic function $f : T \to S$ such that $Ff \circ \tau = \sigma \circ f$. We also write $f$ for the coalgebra morphism.*

$$\begin{array}{ccc} T & \xrightarrow{f} & S \\ {\scriptstyle\tau}\downarrow & & \downarrow{\scriptstyle\sigma} \\ FT & \xrightarrow[Ff]{} & FS \end{array}$$

*The composition of coalgebra morphisms inherits from the corresponding operation in* **Set**, *and the same goes for identities. Hence the class $\mathbf{Alg}_F$ of coalgebras is given a category structure.*

**Remark 2.3** *The functor $F$ naturally induces an endofunctor $\tilde{F}$ in $\mathbf{Alg}_F$ by $\tilde{F}\mathbb{T} = (FT, F\tau)$ if $\mathbb{T} = (T, \tau)$ and $\tilde{F}f = Ff$. We see that $\tau$ is a coalgebra*

*morphism $\mathbb{T} \to \tilde{F}\mathbb{T}$, and that the commutative diagram in* **Set** *in the definition of a coalgebra morphism can also be read in* $\mathbf{Alg}_F$ *as:*




*We will also write $F$ for this morphism $\tilde{F}$.*

**Remark 2.4** *The forgetful functor $\mathbf{Alg}_F \to$ **Set** that associates a coalgebra with its support is faithful. Therefore, to verify that a diagram in $\mathbf{Alg}_F$ is commutative, we can simply consider its projection in* **Set**.

The following lemma is interesting to "understand" the category $\mathbf{Alg}_F$, especially the way in which it inherits the existence of amalgamated sums from **Set**.

**Lemma 2.5** *The category $\mathbf{Alg}_F$ is cocomplete.*

*Proof*: We can easily show the existence of arbitrary sums and coequalizers, which is enough to establish the existence of arbitrary colimits. These constructions can be directly deduced from those of **Set**.

We will sketch another proof, that shows how colimits can be built directly from colimits obtained in **Set**. Let $I$ be a small category and $D$ an $I$-diagram in $\mathbf{Alg}_F$, that is, a functor $I \to \mathbf{Alg}_F$. We write $Di = \mathbb{T}_i = (T_i, \tau_i)$. Composition with the forgetful functor $\mathbf{Alg}_F \to$ **Set** gives an $I$-diagram $D'$ in **Set**. Since **Set** is cocomplete, we can consider a limiting $D'$-cocone $(T, (f_i : T_i \to T)_{i \in I})$. To make a coalgebra out of $T$, we have to define a function $\tau : T \to FT$. According to the universal property of colimits, this function is fully defined by the $D$-cocone $(FT, (g_i : T_i \to FT)_{i \in I})$. We take $g_i = Ff_i \circ \tau_i$. Let us show that it is indeed a cocone. Let $u : i \to j$. Since $Du : \mathbb{T}_i \to \mathbb{T}_j$ is a morphism in $\mathbf{Alg}_F$, we have: $\tau_j \circ Du = F(Du) \circ \tau_i$, hence $g_j \circ Du = Ff_j \circ \tau_j \circ Du = F(f_j \circ Du) \circ \tau_i = Ff_i \circ Du = g_i$. We deduce from that cone a function $\tau : T \to FT$, which gives a coalgebra $\mathbb{T} = (T, \tau)$. It can then be verified that the $f_i$ are coalgebra morphisms $\mathbb{T}_i \to \mathbb{T}$ and that the $D$-cocone $(\mathbb{T}, (f_i : \mathbb{T}_i \to \mathbb{T})_{i \in I})$ is limiting. □

**Finite coalgebras and recursive environments** It would be conceivable to stop there, and to start working with finite coalgebras. Such coalgebras can be seen as finite systems of equations $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$ where the $d_i$ are elements of $FX$ with $X = \{\alpha_1, \ldots, \alpha_n\}$. If we fix such a system, we can call "types" the elements of $FX$ and "recursion variables" the elements of $X$. The system itself is a "recursive environment" that defines recursion variables. The formalism now has to be developed with regard to a fixed recursive environment, and it is necessary to point out explicitly when it is necessary to extend it (in an algorithm or in the proof of a theorem). From a formal point of view, a type only

has meaning with regard to a given environment. To define an algebra of types, one would actually have to define a type as a pair consisting of an environment $(X, \sigma : X \to FX)$ and of an element of $FX$. We certainly want to identify two types that differ only by their environment, one being an extension of the other. But one may want to identify more types, for example modulo variable renaming, unfolding, or modulo a coinductive concept of equality. Making more types equal speeds up algorithms that perform by saturation, but detecting equivalences can come at a significant cost. So there is a trade-off to be made at the implementation level. In addition, taking into account equivalences in the theoretical development can be technically cumbersome.

It is legitimate to seek a formalism that is sufficiently abstract to allow for a proper theoretical development, but which nevertheless reflects the various possible implementation choices.

## 2.3 Congruences, quotients

**Definition 2.6** *Let $\mathbb{T} = (T, \tau)$ be a coalgebra and $\equiv$ be an equivalence relation on $T$. We denote by $T/\equiv$ the quotient set of this equivalence and by $\pi_\equiv : T \to T/\equiv$ the canonical projection. We say that $\equiv$ is a* **congruence** *(on $\mathbb{T}$) if there exists a function $\tau_\equiv$ which commutes the following diagram:*

$$\begin{array}{ccc} T & \xrightarrow{\tau} & FT \\ \downarrow{\scriptstyle \pi_\equiv} & & \downarrow{\scriptstyle F\pi_\equiv} \\ T/\equiv & \xrightarrow[\tau_\equiv]{} & F(T/\equiv) \end{array}$$

*Such a function is unique if it exists. If $\equiv$ is a congruence, we will speak of* the *quotient coalgebra $\mathbb{T}/\equiv \;=\; (T/\equiv, \tau_\equiv)$. The projection $\pi_\equiv$ is a coalgebra morphism.*

**Lemma 2.7** *Let $\mathbb{T}$ be a coalgebra and $(\equiv_i)_{i\in I}$ a family of congruences on $\mathbb{T}$. Then the equivalence relation $\equiv$ generated by the family $\equiv_i$ is yet another congruence on $\mathbb{T}$.*

> *Proof*: We need to prove that if $\alpha, \beta \in T$ and $\alpha \equiv \beta$, then $F\pi \circ \tau(\alpha) = F\pi \circ \tau(\beta)$. Since the equivalence relation $\equiv$ is generated by the $\equiv_i$, we can suppose that $\alpha \equiv_i \beta$ for a certain $i$. Let us write $\pi'_i$ the projection $T/\equiv_i \to T/\equiv$. We have: $F\pi \circ \tau = F(\pi'_i \circ \pi_{\equiv_i}) \circ \tau = F\pi'_i \circ F\pi_{\equiv_i} \circ \tau = F\pi'_i \circ \tau_{\equiv_i} \circ \pi_{\equiv_i}$. Since $\pi_{\equiv_i}(\alpha) = \pi_{\equiv_i}(\beta)$, we deduce that $F\pi \circ \tau(\alpha) = F\pi \circ \tau(\beta)$. □

**Corollary 2.8** *For all coalgebra $\mathbb{T}$, there exists an upper bound for congruences.*

**Example 2.9** *Let us give an example to explain the intuition (for the rest of this chapter). Let $\Sigma$ be an alphabet and $F$ the functor defined by $F(X) = \{0,1\} \times X^\Sigma$. We can interpret a finite coalgebra $\mathbb{T} = (T, \tau)$ as a complete finite deterministic automaton over the alphabet $\Sigma$. If $q \in T$ and $\tau(q) = (\epsilon, d)$, then $\epsilon = 1$ means that $q$ is a final state and $d : \Sigma \to T$ gives the outgoing transitions for $q$. The largest congruence corresponds to the classic Nerode equivalence, and the quotient of $\mathbb{T}$ by this congruence is the minimal deterministic automaton associated with $\mathbb{T}$.*

## 2.4 Regularity and recursivity

Instead of defining our type algebra by a universal property (initial algebra, final coalgebra), we will axiomatize the two properties that we need in order to develop the theory. This will give more freedom to the implementation. These two properties are the regularity and recursivity of the coalgebra. They respectively ensure:

— the termination of algorithms that proceed by iterative decomposition of types and memoization;

— the existence of solutions to systems of equations, with the purpose of defining recursive types.

To formalize these two conditions, it is convenient to make the following assumption about functor $F$.

**Definition 2.10 (Signature)** *A functor $F : \mathbf{Set} \to \mathbf{Set}$ preserves set-theoretic inclusions if, for sets $X, Y$ such that $X \subseteq Y$, then $FX \subseteq FY$ and the image of the canonical inclusion $i_{X,Y}$ is the canonical inclusion $i_{FX,FY}$. We also say that $F$ is a* **signature**.

**Example 2.11** *The functor $\mathcal{P}_f(\_)$ preserves set-theoretic inclusions, as well as any functor that associates to a set $X$ a set of terms with free variables in $X$ (and, as a particular case, constant functors). The composition of two functors that preserve set-theoretic inclusions still possesses this property, and the same goes for the sum and product of two functors (defined by $(F+G)(X) = FX + GX$, and $(F \times G)(X) = FX \times GX$). On this basis, we can obtain the functor from Example 2.9.*

Let us suppose from now on that the functor $F$ preserves set-theoretic inclusions.

**Lemma 2.12** *Let $f : X \to Y$ be a function. Then: $(Ff)(FX) \subseteq F(f(X))$.*

| *Proof*: Just write $f = i_{f(X),Y} \circ f$. □

**Lemma 2.13** *Let $f : X \to Y$ be a function and $A \subseteq X$. Then: $F(f_{|A}) = (Ff)_{|FA}$.*

| *Proof*: Just write $f_{|A} = f \circ i_{A,X}$. □

**Definition 2.14 (Sub-coalgebra, extension)** *Let $\mathbb{T} = (T, \tau)$ and $\mathbb{S} = (S, \sigma)$ be two coalgebras. We say that $\mathbb{T}$ is a* **sub-coalgebra** *of $\mathbb{S}$, or that $\mathbb{S}$ is an* **extension** *of $\mathbb{T}$ if $T \subseteq S$ and the canonical inclusion $T \hookrightarrow S$ is an algebra morphism $\mathbb{T} \to \mathbb{S}$. We then write $\mathbb{T} \subseteq \mathbb{S}$. A* **retraction** *of $\mathbb{S}$ over $\mathbb{T}$ is a morphism $f : \mathbb{S} \to \mathbb{T}$ such that $f \circ i_{\mathbb{T},\mathbb{S}} = Id_{\mathbb{T}}$.*

**Remark 2.15** *The sub-coalgebras of $\mathbb{S} = (S, \sigma)$ exactly correspond to the subsets $T \subseteq S$ such that $\sigma(T) \subseteq FT$. With this correspondence, an arbitrary union of sub-coalgebras is still a sub-coalgebra.*

**Remark 2.16** *The extensions of $\mathbb{T} = (T, \tau)$ exactly correspond to the functions $\sigma : X \to F(T + X)$ where $X$ is an arbitrary set.*

**Lemma 2.17** *Let $f : \mathbb{T} \to \mathbb{S}$ be a coalgebra morphism. Then the image of $f$ is a sub-coalgebra of $\mathbb{S}$.*

> *Proof*: The aim is to show that $\sigma(f(T)) \subseteq F(f(T))$. But $\sigma(f(T)) = Ff(\tau(T))$ since $f$ is a coalgebra morphism. We then write: $Ff(\tau(T)) \subseteq Ff(FT) \subseteq F(f(T))$ by Lemma 2.12. □

### 2.4.1 Regularity

**Definition 2.18 (Basis, regularity)** *Le $\mathbb{T}$ be a coalgebra. A* **basis** *of $\mathbb{T}$ is a sub-coalgebra with finite support. An element of the support of $\mathbb{T}$ is said* **regular** *if it belongs to a certain basis. The algebra $\mathbb{T}$ is said* **regular** *if all the elements of its support are regular.*

**Lemma 2.19** *A finite union of bases is a basis.*

**Lemma 2.20** *The direct image of a basis by a coalgebra morphism is a basis.*

**Lemma 2.21** *Let $f : \mathbb{T} \to \mathbb{S}$ be a coalgebra morphism. If $\mathbb{T}$ is regular, then its image by $f$ is regular too. In particular, any quotient of a regular coalgebra is regular.*

**Remark 2.22** *We have seen that the sub-coalgebras of $\mathbb{T} = (T, \tau)$ can be identified with their support, that is a subset of $T$. We will use this identification for bases.*

The regularity condition can be understood the following way. Let $\mathbb{T} = (T, \tau)$ be a regular coalgebra and $t$ an element of $T$. We can find a basis $S \subseteq T$ which contains it. The decomposition $\tau(t)$ is in $F$, meaning that it is an expression built immediately from other elements of $S$. We can then continue the decomposition, and all the elements obtained are still in $S$. The finiteness of $S$ ensures the termination of that saturation process.

**Example 2.23** *Let us take back the functor $F$ from Example 2.9. Consider a coalgebra $\mathbb{T} = (T, \tau)$, i.e. an automaton (that might be infinite). A state $q \in T$ is regular if it is part of a finite sub-automaton, that is, if its forward closure is finite.*

**Lemma 2.24** *Let $\mathbb{T}$ be an algebra. The sub-coalgebra of regular elements is a regular algebra.*

### 2.4.2 Recursivity

We now want to ensure that every system of equations $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$, with the $d_i$ in $F\{\alpha_1, \ldots, \alpha_n\}$ has a solution. In fact, we also want to be able to use already-known elements of the algebra on the right hand side of the equations.

**Definition 2.25 (Finite extension, regular extension)** *Let $\mathbb{T}$ be a coalgebra. An extension $\mathbb{S}$ of $\mathbb{T}$ is* **finite** *if the set-theoretic difference of the supports is finite.*

*An extension $\mathbb{S}$ of $\mathbb{T}$ is* **regular** *if for any element of $\mathbb{S}$, there exists a sub-algebra $\mathbb{T}'$ of $\mathbb{S}$ which contains this element and is a finite extension of $\mathbb{T}$ ($\mathbb{T} \subseteq \mathbb{T}' \subseteq \mathbb{S}$).*

**Remark 2.26** *The regular extensions of the empty coalgebra are precisely the regular coalgebras.*

A more concret way of seeing things is to say that a finite extension of $\mathbb{T}$ is given by a finite system $\{\alpha_1 = d_1; \ldots, \alpha_n = d_n\}$ where the $d_i$ are elements of $F(T+\{\alpha_1, \ldots, \alpha_n\})$, meaning they are expressions formed either from variables, or from elements of $T$. A retraction is a solution to this system, that is, a way of injecting the variables $\alpha_i$ in $T$ by validating the equations (this is the meaning of the commutative diagram that expresses the fact that retraction is a coalgebra morphism).

A more concrete way of looking at it is to say that a finite extension of $\mathbb{T}$ is given by a finite system $\{\alpha_1 = d_1; \ldots, \alpha_n = d_n\}$ where the $d_i$ are elements of $F(T+\{\alpha_1, \ldots, \alpha_n\})$, meaning they are expressions formed either from variables, or from elements of $T$

**Definition 2.27** *A coalgebra $\mathbb{T}$ is* **recursive** *if any finite extension admits a retraction.*

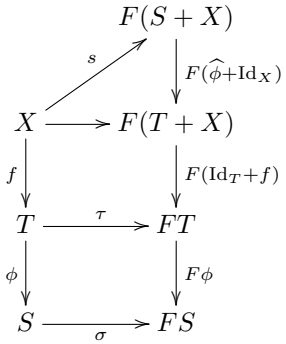
This definition does not impose unicity of the solutions. The same system can have multiple different solutions, and this frees the implementation from having to compute a unique representation of types.

**Example 2.28** *Let us continue with Example 2.9. If $\mathbb{T}$ is a recursive coalgebra, then any finite automaton can be injected into it (not necessarily in a unique way). Even better, if we give ourselves a finite automaton with transitions towards states of $\mathbb{T}$, then we can find an injection of this automaton into $\mathbb{T}$ that preserves the states in $\mathbb{T}$.*

**Lemma 2.29** *The image of a recursive coalgebra by a morphism is a recursive coalgebra. In particular,every quotient of a recursive coalgebra is recursive.*

*Proof*: Let $\phi : \mathbb{T} \to \mathbb{S}$ be a morphism, with $\mathbb{T} = (T, \tau)$ and $\mathbb{S} = (S, \sigma)$. We can assume that the morphsim is surjective and the aim is to show that if $\mathbb{T}$ is recursive, then so is $\mathbb{S}$. Let $\widehat{\phi}$ be a right inverse of the set-theoretic function $\phi$, that is a function $\widehat{\phi} : S \to T$ such that $\phi \circ \widehat{\phi} = \mathrm{Id}_S$.

We consider a finite extension of $\mathbb{S}$, say $\mathbb{S}' = (S+X, \sigma+s)$ with $s : X \to F(S+X)$. The recursivity of $\mathbb{T}$ gives the existence of a function $f : X \to T$ that commutes the diagram:




The function $F(S+X) \to FS$, obtained by composition on the

right side of of the diagram, is $F(\phi \circ \widehat{\phi} + \phi \circ f) = F(\mathrm{Id}_S + \phi \circ f)$, which gives the retraction $\mathbb{S}' \to \mathbb{S}$:

$$\begin{array}{ccc} X & \xrightarrow{s} & F(S+X) \\ {\scriptstyle \phi \circ f}\downarrow & & \downarrow{\scriptstyle F(\mathrm{Id}_S + \phi \circ f)} \\ S & \xrightarrow[\sigma]{} & FS \end{array}$$

□

**Lemma 2.30** *Let $\mathbb{T} = (T, \tau)$ be a recursive coalgebra. Then function $\tau : T \to FT$ is surjective.*

*Proof*: Let $t \in FT$. We consider the finite extension $\mathbb{S} = (T + \{\bullet\}, \sigma)$ of $\mathbb{T}$ defined by $\sigma(\bullet) = t$. A retraction of $\mathbb{S}$ over $\mathbb{T}$ gives indeed an element of $T$ whose image by $\tau$ is $t$. □

This show that the coalgebra $F\mathbb{T}$ is isomorphic to a quotient of $\mathbb{T}$ as long as $\mathbb{T}$ is recursive.

In addition, we can choose a right inverse of $\tau$, that is a function $f : FT \to T$ such that $\tau \circ f = \mathrm{Id}_{FT}$. This makes it possible to see $T$ as a $F$-algebra rather than as as a $F$-coalgebra. However, the choice of that inverse $f$ is not unique in general, therefore this point of view is not canonical.

**Lemma 2.31** *Let $\mathbb{T}$ be a recursive coalgebra. Then the coalgebra $F\mathbb{T} = (FT, F\tau)$ is recursive. And if $\mathbb{T}$ is regular, then so is $F\mathbb{T}$.*

*Proof*: Corollary of the Lemmas 2.21, 2.29 and 2.30. □

**Lemma 2.32 (Finite extension)** *Let $\mathbb{T}$ be a recursive coalgebra, $\mathbb{S}_1$ an arbitrary coalgebra, and $\mathbb{S}_2$ a finite extension of $\mathbb{S}_1$. Then every morphism $\mathbb{S}_1 \to \mathbb{T}$ can be extended to $\mathbb{S}_2$.*

*Proof*: Let $f : \mathbb{S}_1 \to \mathbb{T}$ be a morphism. Consider the amalgamated sum of the two morphisms $f : \mathbb{S}_1 \to \mathbb{T}$ and $i_{\mathbb{S}_1,\mathbb{S}_2} : \mathbb{S}_1 \to \mathbb{S}_2$ (which exists according to Lemma 2.5). We can build it so that we have a finite extension $\mathbb{S}_2'$ of $\mathbb{T}$:

$$\begin{array}{ccc} \mathbb{S}_1 & \hookrightarrow & \mathbb{S}_2 \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f'} \\ \mathbb{T} & \hookrightarrow & \mathbb{S}_2' \end{array}$$

The regularity of $\mathbb{T}$ gives a retraction $g : \mathbb{S}_2' \to \mathbb{T}$:

$$\begin{array}{ccc} \mathbb{S}_1 & \hookrightarrow & \mathbb{S}_2 \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f'} \\ \mathbb{T} & \hookrightarrow & \mathbb{S}_2' \\ & {\scriptstyle \mathrm{Id}_{\mathbb{T}}}\searrow & \downarrow{\scriptstyle g} \\ & & \mathbb{T} \end{array}$$

We deduce a morphism $g \circ f' : \mathbb{S}_2 \to \mathbb{T}$ which extends $f$. □

**Theorem 2.33 (Regular extension)** *Let $\mathbb{T}$ be a recursive coalgebra, $\mathbb{S}_1$ an arbitrary coalgebra and $\mathbb{S}_2$ a regular extension of $\mathbb{S}_1$. Then every morphism $\mathbb{S}_1 \to \mathbb{T}$ can be extended to $\mathbb{S}_2$.*

*Proof*: Let $f : \mathbb{S}_1 \to \mathbb{T}$ be a morphism. We consider the set of pairs $(\mathbb{S}', f')$ where $\mathbb{S}_1 \subseteq \mathbb{S}' \subseteq \mathbb{S}_2$ and $f'$ is a morphism $\mathbb{S}' \to \mathbb{T}$ that extends $f$. We define an order on this set by: $(\mathbb{S}'_1, f'_1) \leq (\mathbb{S}'_2, f'_2)$ iff $\mathbb{S}'_1 \subseteq \mathbb{S}'_2$ and $f'_2$ extends $f'_1$. This ordered set is inductive, hence the Zorn lemma ensures the existence of a maximal element $(\mathbb{S}_m, f_m)$. We now prove that $\mathbb{S}_m = \mathbb{S}_2$. It is enough to prove that for any sub-coalgebra $\mathbb{S}'$ of $\mathbb{S}_2$ which is a finite extension of $\mathbb{S}_1$, we have $\mathbb{S}' \subseteq \mathbb{S}_m$. For this coalgebra, we consider the sub-coalgebra $\mathbb{S}'' = \mathbb{S}' \cup \mathbb{S}_m$. It is a finite extension of $\mathbb{S}_m$, hence, by the previous lemma, the morphism $f_m$ can be extended to $\mathbb{S}''$. The maximality of the pair $(\mathbb{S}_m, f_m)$ gives $\mathbb{S}_m = \mathbb{S}''$. □

**Corollary 2.34 (Regular retraction)** *Let $\mathbb{T}$ be a recursive coalgebra and $\mathbb{S}$ be a regular extension. Then there exists a retraction $\mathbb{S} \to \mathbb{T}$.*

*Proof*: Just apply the previous theorem to extend the morphism $\mathrm{Id}_{\mathbb{T}}$ to $\mathbb{S}$. □

This corollary will be used in the next section.

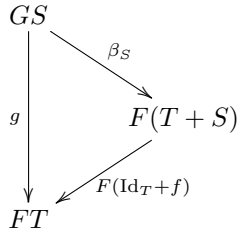
### 2.4.3 Application: transfer between signatures

In only this section, we take two different signatures, that is, two functors $F$ and $G$ that preserve set-theoretic inclusions. We give ourselves a recursive $F$-coalgebra $\mathbb{T} = (T, \tau)$ and a regular $G$-coalgebra $\mathbb{S} = (S, \sigma)$. It is natural to consider the pairs of functions $(f : S \to T, g : GS \to FT)$ that are compatible with the respective coalgebra structures, i.e. that make the following diagram commute:

$$\begin{array}{ccc} S & \xrightarrow{\sigma} & GS \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ T & \xrightarrow[\tau]{} & FT \end{array}$$

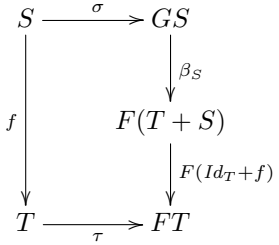
We want to constrain these functions by a system of equations that associate with every element of $GS$ an element of $F(T + S)$. To express the fact that this system does not depend from the construction of $\mathbb{S}$, we suppose that the function $GS \to F(T + S)$ is defined as $\beta_S$ where $\beta$ is a natural transformation $\beta : G \to F(T + \_)$. Formally, what we are asking is that the following diagram

commutes:




In other words, function $g$ can be deduced from $f$. The next theorem ensures the existence of such a function $f$.

**Theorem 2.35** *Given the objects introduced above, there exists a function $f : S \to T$ such that the following diagram commutes:*




*Proof*: We have the $F$-coalgebra $\mathbb{T}' = (T + S, \tau + \beta_S \circ \sigma)$ which is an extension of $\mathbb{T}$. A retraction of this $F$-coalgebra over $\mathbb{T}$ will give indeed a function $f : S \to T$ solution of the system, meaning that it verifies $\tau \circ f = F(\mathrm{Id}_T + f) \circ \beta_S \circ \sigma$. For every basis $B$ of $\mathbb{S}$, we have the sub-coalgebra of $\mathbb{T}'$ with support $T + B$, that is a finite extension of $\mathbb{T}$. This results from the naturality diagram of $\beta$ for the canonical injection $i_{B,S}$. Corollary 2.34 ensures indeed the existence of a retraction $\mathbb{T}' \to \mathbb{T}$. □

**An example** Now let us give a concrete application of the previous result. It is a simplified version of Theorem 6.9. We take a finite alphabet $\Sigma$. The functor $F$ associates to a set $X$ the set of finite terms generated by the following syntax:

$$t := \quad (t \vee t) \;\mid\; (x \times x) \;\mid\; a$$

where $x$ denotes a generic element from $X$, and $a$ denotes a generic element from $\Sigma$. We suppose as given a recursive $F$-coalgebra $\mathbb{T} = (T, \tau)$. We can see it as a type algebra (with product types and union types).

Similarly, the functor $G$ associates to a set $Y$ the set of finite terms generated by the following syntax:

$$p := \quad (p|p) \;\mid\; (y, y) \;\mid\; t$$

where $y$ denotes a generic element from $Y$ and $t$ denotes a generic element from $FT$. We suppose given a regular $F$-coalgebra $\mathbb{S} = (S, \sigma)$. We can see it as a pattern algebra.

We then want to associate to every pattern a type that represents the values accepted by $p$. Formally, we want to define two functions $f : S \to T$ and

$g : GS \to FT$ linked by the relation $\tau \circ f = g \circ \sigma$, and which verify the following equations:

$$\begin{cases} g(p_1|p_2) & = & g(p_1) \mathbin{\mathsf{V}} g(p_2) \\ g((y_1, y_2)) & = & f(y_1) \mathbin{\mathsf{\times}} f(y_2) \\ g(t) & = & t \end{cases}$$

The link with the theorem above comes when we define the natural transformation $\beta : G \to F(T + \_)$ by:

$$\begin{cases} \beta_Y(p_1|p_2) & = & \beta_Y(p_1) \mathbin{\mathsf{V}} \beta_Y(p_2) \\ \beta_Y((y_1, y_2)) & = & y_1 \mathbin{\mathsf{\times}} y_2 \\ \beta_Y(t) & = & t \end{cases}$$

We can then apply the theorem to define function $f : S \to T$, from which we deduce function $g$.

## 2.5 Constructions

In this section, we show how to formally build recursive regular coalgebras. We present three constructions. The first models an implementation which optimally shares recursive types (meaning that two types that have the same infinite unfolding will be equal in this coalgebra). The second, on the contrary, models an implementation that does not seek to introduce any sharing. The third is intermediate: it detects types defined by the same equations (modulo recursion variables renaming and suppression of useless equations), and shares them.

The constructions are built on an countably infinite set of variables $V$, which we fix for the rest of the study.

Similarly to the classical construction of an initial algebra, we need a continuity assumption on functor $F$. Since $F$ preserves set-theoretic inclusions, we can give a concrete version of this continuity hypothesis.

**Definition 2.36 (Continuity)** *A family of sets $(X_i)_{i \in I}$ is* **directed** *if:*

$$\forall i, j \in I.\ \exists k \in I.\ X_i \cup X_j \subseteq X_k$$

*The functor $F$ is* **continuous** *if for any family of directed sets $(X_i)_{i \in I}$:*

$$F(\bigcup_{i \in I} X_i) = \bigcup_{i \in I} FX_i$$

Throughout this chapter, we assume that functor $F$ is continuous. One consequence of this assumption is that the class of regular coalgebras is stable by finite extension.

**Lemma 2.37 (Stability of regular coalgebras by finite extension)** *Let $\mathbb{T} \subseteq \mathbb{S}$ be a finite extension. If $\mathbb{T}$ is regular, then $\mathbb{S}$ is regular.*

> *Proof*: Let us write $\mathbb{T} = (T, \tau)$ and $\mathbb{S} = (S, \sigma)$. We can write $S = T + X$ where $X$ is a finite set. By continuity of the functor $F$ and regularity of $\mathbb{T}$, we obtain: $F(S) = F(T + X) = F(\bigcup B + X) =$

$\bigcup F(B+X)$ where the unions are taken on the set of bases of $\mathbb{T}$. It is a directed family because the union of two bases is still a basis. The set $\sigma(X)$ being finite, it is possible to find a basis $\mathbb{B}$ such that $\sigma(X) \subseteq F(B+X)$. By definition of a basis, we also have: $\sigma(B) \subseteq FB \subseteq F(B+X)$. We obtain a basis $\mathbb{B}'$ of $\mathbb{S}$ whose support is $B+X$. We have showed that all the elements of $S \backslash T$ are in a basis of $\mathbb{S}$. The same is obviously true for elements of $T$. □

**Remark 2.38** *By a similar proof, we can show the transitivity of regular extensions: if $\mathbb{T}_1 \subseteq \mathbb{T}_2$ and $\mathbb{T}_2 \subseteq \mathbb{T}_3$ are two regular extensions, then so is $\mathbb{T}_1 \subseteq \mathbb{T}_3$.*

### 2.5.1 Optimal sharing

In this section, we are going to build a regular and recursive coalgebra with optimal sharing, that is, that has no quotients.

The idea is to define a type by a finite system of equations $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$ by distinguishing one of its variables $\alpha_i$. We consider the pairs $(\mathbb{T}, \alpha)$ where $\mathbb{T} = (T, \tau)$ is a finite coalgebra with support in $V$ ($T \in \mathcal{P}_f(V)$) and $\alpha \in T$. To introduce optimal sharing, we define $\simeq_D$ as the equivalence relation generated by the equations:

$$(\mathbb{T}, \alpha) \simeq_D (\mathbb{S}, f(\alpha))$$

for any morphism of finite coalgebras $f : \mathbb{T} \to \mathbb{S}$. Let $[\mathbb{T}, \alpha]$ denote the equivalence class of $(\mathbb{T}, \alpha)$ and $D$ the set of those classes. We are going to give a coalgebra structure to $D$, and show that it is regular and recursive.

**Remark 2.39** *Because of the equivalence $\simeq_D$, we will allow ourselves to write $(\mathbb{T}, \alpha)$ for any finite coalgebra $\mathbb{T}$. Indeed, the $\simeq_D$ equivalence allows us to reason up to variable renaming, which allows us to get back to the case where the support of $\mathbb{T}$ is a subset of $V$..*

First of all, for any finite coalgebra $\mathbb{T} = (T, \tau)$, we define function $\Phi_{\mathbb{T}} : T \to D$ by:

$$\Phi_{\mathbb{T}}(\alpha) = [\mathbb{T}, \alpha]$$

**Lemma 2.40** *If $f : \mathbb{T} \to \mathbb{S}$ is a morphism of finite coalgebras, then $\Phi_{\mathbb{S}} \circ f = \Phi_{\mathbb{T}}$.*

*Proof*: We compute: $\Phi_{\mathbb{S}}(f(\alpha)) = [\mathbb{S}, f(\alpha)] = [\mathbb{T}, \alpha] = \Phi_{\mathbb{T}}(\alpha)$. □

This allows us to define a function $\delta : D \to FD$ by:

$$\delta([\mathbb{T}, \alpha]) = F\Phi_{\mathbb{T}}(\tau(\alpha))$$

if $\mathbb{T} = (T, \tau)$. This definition is valid because if $f : \mathbb{T} \to \mathbb{S}$ then, by the previous fact, we have: $F\Phi_{\mathbb{S}}(\sigma(f(\alpha))) = F\Phi_{\mathbb{S}}(Ff \circ \tau(\alpha)) = F(\Phi_{\mathbb{S}} \circ f)(\tau(\alpha)) = F\Phi_{\mathbb{T}}(\tau(\alpha))$. Let $\mathbb{D} = (D, \delta)$ be the coalgebra thus defined. We will prove that it is regular and recursive.

**Lemma 2.41** *Function $\Phi_{\mathbb{T}} : T \to D$ is a coalgebra morphism $\mathbb{T} \to \mathbb{D}$.*

*Proof*: Immediate consequence of the definition of $\Phi_{\mathbb{T}}$ and of $\delta$. □

**Theorem 2.42 (Regularity)** *The coalgebra $\mathbb{D}$ is regular.*

*Proof*: Let $\mathbb{T} = (T, \tau)$ be a finite arbitrary coalgebra. By using Lemmas 2.41 and 2.17, we obtain that the set $\Phi_{\mathbb{T}}(T) = \{[\mathbb{T}, \alpha] \mid \alpha \in T\}$ is a sub-coalgebra of $\mathbb{D}$. Its finite character is obvious. So it is a basis, and any element of $D$ is in such a set. □

The following lemma connects two different visions of the bases of $\mathbb{D}$. Indeed, such a basis $\mathbb{B}$ can be seen either as a finite set of $D$ elements, or as a finite coalgebra that induces a morphism $\Phi_{\mathbb{B}} : \mathbb{B} \to \mathbb{D}$.

**Lemma 2.43 (Internalization of the bases)** *Let $\mathbb{B}$ be a basis of $\mathbb{D}$. Then $\Phi_{\mathbb{B}}$ is the canonical injection $\mathbb{B} \hookrightarrow \mathbb{D}$.*

*Proof*: We have to show that for any element $[\mathbb{T}, \alpha]$ of $\mathbb{B}$, we get: $[\mathbb{B}, [\mathbb{T}, \alpha]] = [\mathbb{T}, \alpha]$. To do so, let us introduce the basis $\mathbb{S} = \Phi_{\mathbb{T}}(\mathbb{T}) \cup \mathbb{B}$ of $\mathbb{D}$. We can see $\Phi_{\mathbb{T}}$ as a coalgebra morphism $\mathbb{T} \to \mathbb{S}$ that sends $\alpha$ over $[\mathbb{T}, \alpha]$ and this produces the equality: $[\mathbb{T}, \alpha] = [\mathbb{S}, [\mathbb{T}, \alpha]]$. On the other hand, we can consider the canonical injection $\mathbb{B} \to \mathbb{S}$, that sends $[\mathbb{T}, \alpha]$ (in $\mathbb{B}$) over itself (in $\mathbb{S}$), which gives: $[\mathbb{B}, [\mathbb{T}, \alpha]] = [\mathbb{S}, [\mathbb{T}, \alpha]]$. By transitivity, we obtain the desired equality. □

**Corollary 2.44** *Let $\mathbb{T}$ be a finite coalgebra. Then the only morphism $\mathbb{T} \to \mathbb{D}$ is $\Phi_{\mathbb{T}}$.*

*Proof*: Let $f : \mathbb{T} \to \mathbb{D}$ be such a morphism. Let $\mathbb{T}'$ be the image of $\mathbb{T}$ by $f$; this is still a finite coalgebra, hence it is a basis of $\mathbb{D}$. We can write $f = i_{\mathbb{T}',\mathbb{D}} \circ f'$ where $f'$ is a morphism $\mathbb{T} \to \mathbb{T}'$. Lemma 2.40 gives $\Phi_{\mathbb{T}} = \Phi_{\mathbb{T}'} \circ f'$ and the previous lemma gives $\Phi_{\mathbb{T}'} = i_{\mathbb{T}',\mathbb{D}}$. From which we conclude $f = \Phi_{\mathbb{T}}$. □

The corollary above characterizes $\mathbb{D}$ with a universal property in the full sub-coalgebra of $\mathbf{Alg}_F$ constituting of regular coalgebras.

**Corollary 2.45 (Finality)** *Let $\mathbb{T}$ be a regular coalgebra. Then there exists a unique morphism $\mathbb{T} \to \mathbb{D}$, that we write $\Phi_{\mathbb{T}}$.*

*Proof*: The restriction of such a morphism to each basis $\mathbb{B}$ of $\mathbb{T}$ must coincide with the unique morphism $\mathbb{B} \to \mathbb{D}$. This gives unicity, as well as the way to build the morphism $\mathbb{T} \to \mathbb{D}$. □

**Corollary 2.46** *The unique morphism $\mathbb{D} \to \mathbb{D}$ is the identity.*

**Corollary 2.47** *If $\mathbb{T}$ is a regular coalgebra, then any morphism $\mathbb{D} \to \mathbb{T}$ is injective.*

*Proof*: If $f$ is a morphism $\mathbb{D} \to \mathbb{T}$, then $\Phi_{\mathbb{T}} \circ f$ is the identity, according to the previous Corollary. □

**Theorem 2.48 (Recursivity)** *The coalgebra $\mathbb{D}$ is recursive.*

*Proof*: Let $\mathbb{T} = (T, \tau)$ be a finite extension of $\mathbb{D}$. According to Lemma 2.37, it is a regular coalgebra, and we obtain with Corollary 2.45 a morphism $\Phi_{\mathbb{T}} : \mathbb{T} \to \mathbb{D}$. It is a retraction according to Corollary 2.46. □

**Theorem 2.49 (Optimal sharing)** *The only congruence over $\mathbb{D}$ is the identity.*

*Proof*: Let $\equiv$ be a congruence over $\mathbb{D}$. Lemma 2.21 shows that the quotient $\mathbb{D}/\equiv$ is a regular coalgebra. Corollary 2.47 then indicates that the projection $\pi : \mathbb{D} \to \mathbb{D}/\equiv$ is injective. This congruence is therefore the identity. □

**Lemma 2.50** *The morphism $\delta : \mathbb{D} \to F\mathbb{D}$ is an isomorphism.*

*Proof*: We just need to check that this set-theoretic function is bijective. Surjectivity is a consequence of Lemma 2.30. According to Lemma 2.31, we know that $F\mathbb{D}$ is regular. Corollary 2.47 shows that the function $\delta$ is injective. □

**Implementation techniques** Efficient algorithms for optimal sharing have been developed by par Mauborgne [Mau99, Mau00] and Considine [Con00], in the case where the functor $F$ associates to a set $X$ a set of (free) terms built on $X$. These techniques are inspired by minimization methods for finite deterministic automatons [Wat94]. They cannot be directly applied, for example, to the case of functor $\mathcal{P}_f(\_)$, or to any functor built from $\mathcal{P}_f(\_)$, that is, when we add axioms such as commutativity or associativity on the terms.

### 2.5.2 Minimal sharing

We now describe another construction of a regular recursive coalgebra that does not rely on maximal sharing. In particular, two types defined by two identical systems of equations up to variable renaming will not be equal in the coalgebra. On the contrary, we are going to build a recursive, regular coalgebra $\mathbb{T}_\infty$ with minimal sharing, that is, such that any other regular recursive coalgebra is obtained as a quotient of $\mathbb{T}\infty$.

The idea is to work with terms of the form "$\alpha_1$ where $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$" where the $d_i$ are expressions built from the $\alpha_i$ and other terms of the same form.

Let $V$ denote an infinite set of variables. We define a sequence of sets $(T_n)_{n\in\mathbb{N}}$ by $T_0 = \emptyset$, $T_{n+1} = \{(X, \sigma, \alpha) \mid X \in \mathcal{P}_f(V), \sigma : X \to F(T_n + X), \alpha \in X\}$. We see that this suite is increasing (for inclusion). We then fix: $T_\infty = \bigcup T_n$. For any pair $(X, \sigma)$ with $X \in \mathcal{P}_f(V), \sigma : X \to F(T_n + X)$, we have the function $\Phi_\sigma : T_\infty + X \to T_\infty$ defined as the identity over $T_\infty$ and by $\Phi_\sigma(\alpha) = (X, \sigma, \alpha) \in T_{n+1} \subseteq T_\infty$ for $\alpha \in X$. We can the define a function $\tau_\infty : T_\infty \to FT_\infty$ by $\tau_\infty(X, \sigma, \alpha) = F\Phi_\sigma(\sigma(\alpha))$. Let $\mathbb{T}_\infty = (T_\infty, \tau_\infty)$ be the coalgebra thus defined.

**Theorem 2.51** *The coalgebra $\mathbb{T}_\infty$ is recursive.*

*Proof*: Let $\mathbb{S} = (S, \sigma)$ be a finite extension of $\mathbb{T}_\infty$. We can assume that $S = T_\infty + X$ for a certain set $X \in \mathcal{P}_f(V)$. We still write $\sigma$ for its restriction to $X$. The image of $X$ by $\sigma$ is contained in $F(T_\infty + X)$, which, by continuity of functor $F$, is the union $\bigcup F(T_n + X)$. However, the set $\sigma(X)$ is finite, hence for a large enough $n$, we can see $\sigma$ as a function $X \to F(T_n + X)$. We then consider function $\Phi_\sigma : S \to T_\infty$ defined above. It extends the identity over $T_\infty$. To conclude the proof, we then only have to show that it is a coalgebra morphism $\mathbb{S} \to \mathbb{T}_\infty$, that is, that:

$$\tau_\infty \circ \Phi_\sigma = F\Phi_\sigma \circ \sigma$$

We verify this equality separately over $T_\infty$ and $X$. Insofar as $\Phi_\sigma$ behaves as the identity over $T_\infty$, we deduce that $F\Phi_\sigma$ behaves as the identity over $FT_\infty$. In addition, $\sigma$ coincides with $\tau_\infty$ over $T$. Hence for an element $t \in T_\infty$, we indeed have $\tau_\infty(\Phi_\sigma(t)) = \tau_\infty(t) = \sigma(t) = F\Phi_\sigma(\sigma(t))$.

Now, let $\alpha \in X$. We have: $\tau_\infty(\Phi_\sigma(\alpha)) = \tau_\infty(X, \sigma, \alpha)$ by definition of $\Phi_\sigma$ and $\tau_\infty(X, \sigma, \alpha) = F\Phi_\sigma(\sigma(\alpha))$ by definition of $\tau_\infty$. □

**Theorem 2.52** *The coalgebra $T_\infty$ is regular.*

*Proof*: First of all, we find that the $T_n$ are sub-coalgebras of $T_\infty$. This comes from the fact that if $\sigma : X \to F(T_n + X)$, then the image of any $\alpha \in X$ by $\Phi_\sigma$ is in $T_{n+1}$; thus $\tau$ sends $T_{n+1}$ in $FT_{n+1}$.

Since $T_\infty = \bigcup T_n$, we only have to show that the sub-coalgebras $\mathbb{T}_n$ are regular. We do this by induction. The base case $n = 0$ is trivial. Assuming that $\mathbb{T}_n$ is regular, we show that $\mathbb{T}_{n+1}$ is regular too. Let $(X, \sigma, \alpha)$ be an element of $T_{n+1}$. We now prove it is regular. We define the coalgebra $\mathbb{S} = (T_n + X, \tau + \sigma)$. It is a finite extension of $\mathbb{T}_n$, so it is a regular algebra according to Lemma 2.37. As in the proof of Theorem 2.51, we can then see $\Phi_\sigma$ as a coalgebra morphism $\mathbb{S} \to \mathbb{T}_{n+1}$. According to Lemma 2.21, its image is regular. And since $\Phi_\sigma(\alpha) = (X, \sigma, \alpha)$ this element of $T_{n+1}$ is regular. □

Insofar as $\mathbb{T}_\infty$ is regular, we know there exists a morphism $\mathbb{T}_\infty \to \mathbb{T}$ for any recursive coalgebra $\mathbb{T}$ (Theorem 2.33). We will show that if we also assume that $\mathbb{T}$ is regular and countable, then we can build a *surjective* morphism $\mathbb{T}_\infty \to \mathbb{T}$. This shows that any coalgebra that verifies these conditions is isomorphic to a "quotient" of $\mathbb{T}_\infty$.

**Theorem 2.53** *Let $\mathbb{T} = (T, \tau)$ be a recursive, regular coalgebra with countable support. Then, there exists surjective morphisms $\mathbb{T}_1 \to \mathbb{T}$ and $\mathbb{T}_\infty \to \mathbb{T}$.*

*Proof*: For any pair $(X, \sigma)$ with $X \in \mathcal{P}_f(V)$ and $\sigma : X \to FX$, we suppose as given a coalgebra morphism $f_\sigma : (X, \sigma) \to \mathbb{T}$. We can then define a function $f : T_1 \to T$ by $f(X, \sigma, \alpha) = f_\sigma(\alpha)$. We then have $f_\sigma = f \circ \Phi_\sigma$. We also have $Ff_\sigma \circ \sigma = \tau \circ f_\sigma$, which expresses the fact that $f_\sigma$ is a morphism. This establishes that

$f$ is a coalgebra morphism $\mathbb{T}_1 \to \mathbb{T}$; for any triplet $(X, \sigma, \alpha)$, we compute: $Ff \circ \tau_\infty(X, \sigma, \alpha) = Ff \circ F\Phi_\sigma \circ \sigma(\alpha) = F(f \circ \Phi_\sigma) \circ \sigma(\alpha) = Ff_\sigma \circ \sigma(\alpha) = \tau \circ f_\sigma(\alpha) = \tau \circ f(X, \sigma, \alpha)$

The existence of the morphisms $f_\sigma$ for any $(X, \sigma)$ is a consequence of the recursivity of $\mathbb{T}$. We now show that a clever choice allows for making the morphism $f : \mathbb{T}_1 \to \mathbb{T}$ surjective.

The bases of $\mathbb{T}$ are countable; we denote them by $(B_n)_{n\in\mathbb{N}}$. For each of those bases, we choose a pair $(X_n, \sigma_n)$ that is isomorphic to it as a coalgebra. We write $f_n : X_n \to B_n$ for this morphism. It is possible to choose the pairs $(X_n, \sigma_n)$ so they are all different. This allows us to choose $f_{\sigma_n} = f_n$. For the other $f_\sigma$, we make an arbitrary choice.

With those choices, the morphism $f$ becomes surjective. Indeed, for any integer $n$, it induces a bijection between $\Phi_\sigma(X_n)$ and $B_n$. Corollary 2.33 allows us to extend $f$ into a surjective morphism $\mathbb{T}_\infty \to \mathbb{T}$. □

**Lemma 2.54** *The two coalgebras $\mathbb{T}_\infty$ and $\mathbb{D}$ are not isomorphic (except if $FX = \emptyset$ for any $X$).*

*Proof*: If $FX = \emptyset$ for any $X$, there is only one coalgebra (with empty support). Assume that is not the case. By continuity of functor $F$, there exists a *finite* set $X$ such that $FX \neq \emptyset$. We can consider, up to renaming, that $X \in \mathcal{P}_f(V)$ and $X \neq \emptyset$.

We know there exists a unique morphism $f : \mathbb{T}_\infty \to \mathbb{D}$. We now prove that it is not an isomorphism. We will show that it is not injective. Consider its restriction to the sub-coalgebra $\mathbb{T}_1$. It is the unique morphism $f_1 : \mathbb{T}_1 \to \mathbb{D}$. But we can define a function $T_1 \to D$ by $f_1(X, \sigma, \alpha) = [(X, \sigma), \alpha]$, hence it is easy to see that it is indeed a coalgebra morphism. Hence it is $f_1$. We now only have to prove that $f_1$ is not injective. We have $X \in \mathcal{P}_f(V)$ such that $FX \neq \emptyset$. Therefore there exists a function $\sigma : X \to FX$, and an element $\alpha \in X$. We deduce there exists a different triplet $(X', \sigma', \alpha')$, for example by renaming the elements of $X$ (or simply by taking a strict overset of $X$), so that there exists a morphism of finite coalgebras $(X, \sigma) \to (X', \sigma')$ that sends $\alpha$ over $\alpha'$. The two triplets $(X, \sigma, \alpha)$ and $(X', \sigma', \alpha')$ do not have the same image by $f_1$, therefore it is not injective. □

### 2.5.3 Sharing modulo renaming and extension

We give a third construction of a regular and recursive coalgebra, that is neither isomorphic to $\mathbb{T}_\infty$, or to $\mathbb{D}$. It means that it can obtained from $\mathbb{T}_\infty$ by quotienting over a congruence larger than the identity, but that is not maximal either. We give a direct construction for it.

The idea is to consider systems $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$ (where the $d_i$ are built over the $\alpha_i$) modulo variable renaming and extension (adding useless equations). Formally, this is like considering the pairs $(\mathbb{T}, \alpha)$ where $\mathbb{T} = (T, \tau)$ is a finite coalgebra whose support is included in $V$, and $\alpha \in T$. We consider

the equivalence induced by the relations:

$$(\mathbb{T}, \alpha) \simeq_{D'} (\mathbb{S}, f(\alpha))$$

for any *injective* morphism $f : \mathbb{T} \to \mathbb{S}$ (that represents renaming and adding equations). We denote by $< \mathbb{T}, \alpha >$ the equivalence class of the pair $(\mathbb{T}, \alpha)$ and by $D'$ the set of those classes. We then proceed as in the case of optimal sharing. For any finite coalgebra $\mathbb{T} = (T, \tau)$, we define function $\Phi'_{\mathbb{T}} : T \to D'$ by:

$$\Phi'_{\mathbb{T}}(\alpha) =< \mathbb{T}, \alpha >$$

**Lemma 2.55** *If $f : \mathbb{T} \to \mathbb{S}$ is an injective morphism of finite coalgebras, then $\Phi'_{\mathbb{S}} \circ f = \Phi'_{\mathbb{T}}$.*

We can then define function $\delta' : D' \to FD'$ by:

$$\delta'(< \mathbb{T}, \alpha >) = F\Phi'_{\mathbb{T}}(\tau(\alpha))$$

if $\mathbb{T} = (T, \tau)$. Let $\mathbb{D}' = (D', \delta')$ be the coalgebra thus defined.

**Lemma 2.56** *Function $\Phi'_{\mathbb{T}} : T \to D'$ is a coalgebra morphism $\mathbb{T} \to \mathbb{D}'$.*

**Theorem 2.57** *The coalgebra $\mathbb{D}'$ is regular.*

*Proof*: This proof is the same as the one for Theorem 2.42. □

**Theorem 2.58** *Coalgebra $\mathbb{D}'$ is recursive.*

*Proof*: Let $\mathbb{S} = (S, \sigma)$ be a finite extension of $\mathbb{D}'$. We can assume that $S = D' + X$ with $X = \{\alpha_1, \ldots, \alpha_n\} \in \mathcal{P}_f(V)$. According to Lemma 2.37, we can deduce the existence of a basis $B$ of $\mathbb{D}'$ that contains $X$. We write $B = \{< \mathbb{T}_1, \beta_1 >, \ldots, < \mathbb{T}_n, \beta_m >\} + X$. In fact, in view of the equivalence modulo renaming and extension, and even if we have to rename the $\beta_i$, we can assume that all the $\mathbb{T}_i$ are equal: $B = \{< \mathbb{T}, \beta_1 >, \ldots, < \mathbb{T}, \beta_m >\} + X$ with $\mathbb{T} = (T, \tau)$. Let us then set $T' = T + X$ and define function $f : B \to T'$ by $f(< \mathbb{T}, \beta_i >) = \beta_i$ and $f(\alpha_i) = \alpha_i$. We can then define $\tau' : T' \to FT'$, the extension of $\tau : T \to FT$, by: $\tau'(\alpha_i) = Ff(\sigma(\alpha_i))$ for $\alpha_i \in X$. Indeed, we know that $\mathbb{B}$ is a basis, hence $\sigma(X) \subseteq B$. This provides us with a finite coalgebra $\mathbb{T}' = (T', \tau')$.

We can now define a retraction $g : \mathbb{S} \to \mathbb{D}'$ as the identity of $D'$ and by $g(\alpha_i) = \Phi'_{\mathbb{T}'}(\alpha_i) =< \mathbb{T}', \alpha_i >$. We have to check that it is a coalgebra morphism. This is found with a simple calculation, by noticing that the restriction of $g$ to $B$ is the composition $\Phi'_{\mathbb{T}'} \circ f$: $\delta' \circ g(\alpha_i) = \delta'(< \mathbb{T}', \alpha_i >) = F\Phi'_{\mathbb{T}'}(\tau'(\alpha_i)) = F(\Phi'_{\mathbb{T}'} \circ f)(\sigma(\alpha_i)) = Fg \circ \sigma(\alpha_i)$. □

We will now prove that this construction gives in general a coalgebra $\mathbb{D}'$ that is isomorphic neither to $\mathbb{D}$ or to $\mathbb{T}_\infty$. We take as functor $F$ the identity. Coalgebras are then the (finite or infinite) graphs whose outdegree of each node is exactly 1.

We easily see that $\mathbb{D}$ is a singleton. We can also describe $\mathbb{D}'$ very precisely . For any coalgebra $\mathbb{T} = (T, \tau)$, and any element $\alpha \in \tau$, we define $p_{\mathbb{T}}(\alpha)$ as the smallest strictly positive integer $n$ such that $\tau^n(\alpha) = \alpha$, or $+\infty$ if that integer does not exist (which is only possible when $\mathbb{T}$ is infinite). We find that if $f : \mathbb{T} \to \mathbb{S}$ is an injective coalgebra morphism, then $p_{\mathbb{T}}(\alpha) = p_{\mathbb{S}}(f(\alpha))$ for every $\alpha$. It is therefore possible to define a function $p : D' \to \mathbb{N}^+$ by $p(< \mathbb{T}, \alpha >) = p_{\mathbb{T}}(\alpha)$. We will show that it is a bijection. For a fixed integer $n > 0$, we can consider the finite coalgebra $\mathbb{Z}_n = (\mathbb{Z}/n\mathbb{Z}, (x \mapsto x + 1))$. For all $x \in \mathbb{Z}$, we have: $p_{\mathbb{Z}_n}(x) = n$. We then find that for any coalgebra $\mathbb{T}$ and $\alpha \in T$, if $p_{\mathbb{T}}(\alpha) = n$, then there exists an injective morphism $\mathbb{Z}_n \to \mathbb{T}$ that sends an arbitrary element of $\mathbb{Z}_n$ over $\alpha$. The bijective nature of function $p$ is deduced from that, which ensures a countable cardinal for $\mathbb{D}'$ and we immediately see that $\mathbb{D}$ and $\mathbb{D}'$ are not isomorphic. However we also see that for every element $x$ of $\mathbb{D}'$, we have $p_{\mathbb{D}'}(x) = 1$ (in other words: $\delta'$ is the identity). Indeed, such an element $x$ can always be written as $< \mathbb{Z}_n, 0 >$ for a certain $n$, and $\delta'(< \mathbb{Z}_n, 0 > ) =< \mathbb{Z}_n, 1 >=< \mathbb{Z}_n, 0 >$. Clearly, the coalgebra $\mathbb{T}_\infty$ does not have the same structure. For example, if we take $X = \{\alpha_1, \alpha_2\}$, and $\sigma(\alpha_1) = \alpha_2$, $\sigma(\alpha_2) = \alpha_1$, then for the element $x = (X, \sigma, \alpha_1)$ of $\mathbb{T}_\infty$, we have: $p_{\mathbb{T}_\infty}(x) = 2$.

# Chapter 3

# Type algebra

In the previous chapter we introduced the formalism used to construct (co)algebras of recursive terms on an arbitrary signature given by a set-theoretic endofunctor $F$. In this chapter, we instantiate this framework to define the type algebra of a calculus which will serve as the core on which we will build the $\mathbb{C}$Duce language.

## 3.1 Boolean combinations

### 3.1.1 Motivation

We want to introduce boolean combinations (union, intersection, complement) in the type algebra. An immediate solution would be to consider these boolean operations as type constructors, in the same way as arrow or product types. With this approach, we would define the functor $F$ as:

$$t \in FX ::= x \to x \mid x \times x \mid x \vee x \mid x \wedge x \mid \neg x \mid \dots$$

where the meta-variable $x$ represents arbitrary $X$ elements.

This approach is well suited to the construction of an inductive type algebra. However, in the formalism we use for recursive algebras, this would allow for a type $t$ to be defined by an equation like:

$$t = \neg t$$

Obviously, it is not possible to give a set-theoretic semantic to such a type. Similarly, an equation such as:

$$t = t \vee t$$

does not give any information about the content of the type $t$. Generally, we want to avoid ill-formed recursions, which "do not traverse an actual constructor". One solution is to stratify the construction, by using an inductive construction for boolean operators. For example, we can define functor $F$ by:

$$t \in FX ::= x \to x \mid x \times x \mid t \vee t \mid t \wedge t \mid \neg t \mid \dots$$

For a fixed set $X$, the set $FX$ represents finite terms built with boolean operators, whose atoms are the "real" constructors of the type algebra. Hence we avoid being able to define ill-formed types.

However, the problem with this construction is that it prevents that identification of types that are equivalent modulo boolean tautologies (for example $t$ and $t \vee t$). In particular, even if the set $X$ is finite, the functor gives an infinite set $FX$. This can be a problem with algorithms that proceed by saturation of a set of types (for example, subtyping algorithms). To ensure termination, we have to find a representation that makes it possible to share tautologically equivalent boolean combinations. It is not necessary to detect all these equivalences, because it can have a high cost, but identifying more equivalent combinations allows to reduce the number of iterations in saturation algorithms.

We will now study an example of a representation that fits these requirements. In section 11.3, we point out some possible variants.

### 3.1.2 Finite boolean combinations in normal disjunctive form

A finite boolean combination in disjunctive normal form on a set of atoms $X$ is a finite union of finite intersections of literals, where a literal is either an atom, or the complement of an atom. We can define this set of objects by the following stratified syntax:

$$\begin{array}{lcl} C & ::= & \mathbb{0} \mid C \vee C \mid L \\ L & ::= & \mathbb{1} \mid L \wedge L \mid x \mid \neg x \end{array}$$

"Production" $L$ represents the "lines" of a disjunctive normal form. By taking into account the associativity and commutativity of intersection, we can define it in a less-syntactic way by gathering on the one hand the set of positive literals $x$ and on the other hand the set of negative literals $\neg x$. We then consider the set $\mathcal{P}_f(X) \times \mathcal{P}_f(X)$. For example, the line $x_1 \wedge \neg x_2 \wedge x_3$ can be represented with the pair $(\{x_1, x_3\}, \{x_2\})$. The neutral element for intersection $\mathbb{1}$ corresponds to the pair $(\emptyset, \emptyset)$.

We can do the same for production $C$, by seeing a combination as a finite set of lines. Overall, a finite boolean combination in disjunctive normal form can be represented by an element of the set $\mathcal{P}_f(\mathcal{P}_f(X) \times \mathcal{P}_f(X))$ which we denote as $\mathcal{B}X$ ($\mathcal{B}$ for boolean combinations). In this representation, the neutral element for union $\mathbb{0}$ corresponds to the empty set.

We can now define boolean combinators as operators on this set $\mathcal{B}X$. We first write $@x$ for the element $\{(\{x\}, \emptyset)\}$ of $\mathcal{B}X$ which correspond to the atom $x$, and similarly we write $@\neg x$ for the negative literal $\{(\emptyset, \{x\})\}$. We define union and intersection by:

$$t_1 \vee t_2 := t_1 \cup t_2$$

$$t_1 \wedge t_2 := \{(P_1 \cup P_2, N_1 \cup N_2) \mid (P_1, N_1) \in t_1, (P_2, N_2) \in t_2\}$$

It is clear that operators $\vee$ and $\wedge$ are commutative and associative, and that they admit respectively $\mathbb{0} := \emptyset$ and $\mathbb{1} := \{(\emptyset, \emptyset)\}$ as neutral elements. For any *finite* family $(t_i)_{i \in I}$ of elements of $\mathcal{B}X$, we can then define naturally $\bigvee_{i \in I} t_i$ and $\bigwedge_{i \in I} t_i$ by induction on the cardinal of $I$. Equivalently, they can be defined by the formulas:

$$\bigvee_{i \in I} t_i := \bigcup_{i \in I} t_i$$

$$\bigwedge_{i \in I} t_i := \{(\bigcup_{i \in I} P_i, \bigcup_{i \in I} N_i \mid \forall i \in I.\ (P_i, N_i) \in t_i\}$$

which show the associative character of operators $\vee$ and $\wedge$.

**Lemma 3.1** *For any element $t \in \mathcal{B}X$, we have:*

$$t = \bigvee_{(P,N)\in t} \left( \bigwedge_{x\in P} @x \wedge \bigwedge_{x\in N} @\neg x \right)$$

This suggests the following definition for complement and difference:

$$\neg t := \bigwedge_{(P,N)\in t} \left( \bigvee_{x\in P} @\neg x \vee \bigvee_{x\in N} @x \right)$$

$$t_1 \setminus t_2 := t_1 \wedge \neg t_2$$

**Lemma 3.2** *For any atom $x \in X$, we have $\neg @x = @\neg x$.*

In order to define a set-theoretic interpretation of types, it is natural to introduce the following definition.

**Definition 3.3** *A* **set-theoretic interpretation** *of $\mathcal{B}X$ in a set $D$ is a function $[\![\_]\!] : \mathcal{B}X \to \mathcal{P}(D)$ such that, for all $t, t_1, t_2 \in \mathcal{B}X$:*

- $[\![\mathbb{0}]\!] = \emptyset$
- $[\![\mathbb{1}]\!] = D$
- $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$
- $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$
- $[\![\neg t]\!] = D \setminus [\![t]\!]$

**Lemma 3.4** *A function $[\![\_]\!] : \mathcal{B}X \to \mathcal{P}(D)$ is a set-theoretic interpretation if and only if for all $t \in \mathcal{B}X$:*

$$[\![t]\!] = \bigcup_{(P,N)\in t} \left( \bigcap_{x\in P} [\![@x]\!] \setminus \bigcup_{x\in N} [\![@x]\!] \right)$$

*(with the convention that an intersection indexed on the empty set equals $D$)*

*Proof*: Suppose that $[\![\_]\!]$ is a set-theoretic interpretation and consider an element $t \in \mathcal{B}X$. By using Lemma 3.1, we obtain:

$$[\![t]\!] = \bigcup_{(P,N)\in t} \bigcap_{x\in P} [\![@x]\!] \cap \bigcap_{x\in N} [\![@\neg x]\!]$$

Lemma 3.2 gives $[\![@\neg x]\!] = [\![\neg @x]\!] = D \setminus [\![@x]\!]$, which allows us to conclude:

$$[\![t]\!] = \bigcup_{(P,N)\in t} \bigcap_{x\in P} [\![@x]\!] \setminus \bigcup_{x\in N} [\![@x]\!]$$

Conversely, if we assume that this formula is valid for all $t \in \mathcal{B}X$, we easily check that $[\![\_]\!]$ is a set interpretation. Consider for example the case $t_1 \wedge t_2$. We have:

$$\begin{aligned}
[\![t_1]\!] \cap [\![t_2]\!] &= \left( \bigcup_{(P_1,N_1)\in t_1} \bigcap_{x\in P_1} [\![@x]\!] \setminus \bigcup_{x\in N_1} [\![@x]\!] \right) \\
&\quad \cap \left( \bigcup_{(P_2,N_2)\in t_2} \bigcap_{x\in P_2} [\![@x]\!] \setminus \bigcup_{x\in N_2} [\![@x]\!] \right) \\
&= \bigcup_{\substack{(P_1,N_1)\in t_1 \\ (P_2,N_2)\in t_2}} \bigcap_{x\in P_1\cup P_2} [\![@x]\!] \setminus \bigcup_{x\in N_1\cup N_2} [\![@x]\!] \\
&= [\![t_1 \wedge t_2]\!]
\end{aligned}$$

□

**Corollary 3.5** *For every function $[\![\_]\!] : X \to \mathcal{P}(D)$, there exists a unique set-theoretic interpretation of $\mathcal{B}X$ into $D$ that sends $@x$ on $[\![x]\!]$ for every atom $x \in X$. We denote it again with $[\![\_]\!]$.*

An important property is that we can see $\mathcal{B} = \mathcal{P}_f(\mathcal{P}_f(\_) \times \mathcal{P}_f(\_))$ as a functor that preserves set-theoretic inclusions. It is legitimate not to specify the set $X$ when manipulating the operators $\vee, \wedge, \neg, \setminus$ and the elements $\mathbb{0}$ and $\mathbb{1}$. For example, if $t_1, t_2$ are both in $\mathcal{B}X$ and in $\mathcal{B}Y$, then we can compute $t_1 \vee t_2$ indifferently in $\mathcal{B}X$ and in $\mathcal{B}Y$, and obtain the same result. This can be interpreted categorically by saying that the operators are "natural" with respect to functor $\mathcal{B}$ (for example, the operator $\vee$ can be seen as a natural transformation $\mathcal{B}\_ \times \mathcal{B}\_ \to \mathcal{B}\_$).

For any element $t \in \mathcal{B}X$, we can define the "set of atoms of $t$" as the smallest set $Y \subseteq X$ such that $t \in \mathcal{B}Y$, which is $\bigcup_{(P,N)\in t} P \cup N$.

## 3.2 Algebras with recursive types and boolean combinations

We now have all the elements we need to define a type algebra with the following characteristics:

— recursive types;
— arbitrary constructors;
— well-formed boolean combinations.

If we write $F$ the signature of constructors (that is to say a functor **Set** $\to$ **Set** which preserves set-theoretic inclusions), we we can define a type algebra as being a recursive and regular $(\mathcal{B} \circ F)$-coalgebra $\mathbb{T} = (T, \tau)$. We write $\widehat{T} = \mathcal{B}FT$ and $T_A = FT$.

**Definition 3.6** *The elements of $T$ (resp. $\widehat{T}$) (resp. $T_A$) are called* **type nodes** *(resp.* **type expressions***, or simply* **types***) (resp.* **atoms***).*

The function $\tau : T \to \widehat{T}$ associates a node to an expression, and an expression is a boolean combination (in disjunctive normal form) of atoms. Atoms are objects built with type nodes, via the signature $F$. We denote by $\theta, \theta_1, \ldots$ the elements of $T$, and $t, t_1, \ldots$ the elements of $\widehat{T}$.

To ensure the existence of a type algebra, it is enough to assume that the functor $F$ is continuous. If that is the case, then functor $\mathcal{B} \circ F$ is continuous too, and we can apply the constructions of the previous chapter.

We now need to define the signature $F$, that is, the way of building atoms from type nodes.

## 3.3 The minimal algebra

We start working with a minimal algebra that only contains three types of constructors: product types, arrow types, and base types. It will be enough to present our approach. We will then extend the algebra with other constructors

(records, XML elements, ...) by explaining the changes to be made to the minimal system.

We fix a set of base types $\mathbb{B}$ (for example `Int`, `Char`) and we define the signature $F$ in the following way:

$$a \in FX ::= x \rightarrow x \;\mid\; x \times x \;\mid\; b$$

where $b$ denotes a generic element from $\mathbb{B}$ and $x$ a generic element from $X$.

Let us summarize the properties of the type algebra thus constructed. The following functions are available:

— The binary constructors $\rightarrow$ and $\times$: $T \times T \to T_A$ and the 0-ary constructors $b \in T_A$.

— The injection of atoms in type expressions $@ : T_A \to \widehat{T}$. We will make it implicit, by omitting $@$, which amounts to acting as if $T_A \subseteq \widehat{T}$.

— The boolean operators on type expressions $\vee, \wedge, \setminus : \widehat{T} \times \widehat{T} \to \widehat{T}$ and $\neg : \widehat{T} \to \widehat{T}$, as well as the two type expressions $\mathbb{0}, \mathbb{1} \in \widehat{T}$.

— The surjective function $\tau : T \to \widehat{T}$ which associates a type node with its description (an expression).

There are three atom "universes": arrow types, product types, base types. Let us write $T_u$ the atoms in universe $u$, where $u \in \{\mathbf{fun}, \mathbf{prod}, \mathbf{basic}\}$:

$$\begin{array}{lcl} T_{\mathbf{basic}} & = & \mathbb{B} \\ T_{\mathbf{fun}} & = & \{\theta_1 \rightarrow \theta_2 \mid (\theta_1, \theta_2) \in T\} \\ T_{\mathbf{prod}} & = & \{\theta_1 \times \theta_2 \mid (\theta_1, \theta_2) \in T\} \end{array}$$

We have:

$$T_A = \bigcup_{u \in \{\mathbf{fun}, \mathbf{prod}, \mathbf{basic}\}} T_u$$

Furthermore, the $(\mathcal{B} \circ F)$-coalgebra is regular and recursive. Recursivity implies that the function $\tau$ is surjective: for any type expression $t$, it is possible to find (in practice, to create) a node whose description is $t$. This allows us to define types inductively. Hence there exists two nodes $\theta_{\mathbb{0}}$ and $\theta_{\mathbb{1}}$ such that: $\tau(\theta_{\mathbb{0}}) = \mathbb{0}$ and $\tau(\theta_{\mathbb{1}}) = \mathbb{1}$. To simplify the notations, we will also denote them by $\mathbb{0}$ and $\mathbb{1}$.

We can also form recursive types by considering finite systems:

$$\left\{ \begin{array}{lcl} \alpha_1 & = & d_1 \\ \cdots & & \\ \alpha_n & = & d_n \end{array} \right.$$

where the $d_i$ are elements of $\mathcal{B}F(T + \{\alpha_1, \ldots, \alpha_n\})$, i.e. type expressions where variables are allowed, in addition to nodes, as constitutive elements of atoms. Here is an example:

$$\left\{ \begin{array}{lcl} \alpha_1 & = & \alpha_1 \times \alpha_2 \\ \alpha_2 & = & (\alpha_1 \rightarrow \alpha_2) \vee (\alpha_2 \rightarrow \alpha_1) \end{array} \right.$$

Recursivity ensures the existence of two nodes $\theta_1$ and $\theta_2$ such that:

$$\left\{ \begin{array}{lcl} \tau(\theta_1) & = & \theta_1 \times \theta_2 \\ \tau(\theta_2) & = & (\theta_1 \rightarrow \theta_2) \vee (\theta_2 \rightarrow \theta_1) \end{array} \right.$$

We can then reuse these nodes in a new system:

$$\left\{ \begin{array}{rcl} \alpha_3 & = & (\theta_1 \times \alpha_3) \vee (\theta_2 \to \theta_2) \end{array} \right.$$

which allows us to build a node $\theta_3$ such that:

$$\tau(\theta_3) = (\theta_1 \times \theta_3) \vee (\theta_2 \to \theta_2)$$

This is the way to *build* recursive types. We will now see how to deconstruct them.

**Definition 3.7** *A* **socle** *is a finite set of types $\beth \subseteq \widehat{T}$ that verifies the following conditions:*

— *$\beth$ is finite;*
— *$\beth$ contains $\mathbb{0}$, $\mathbb{1}$ and is stable by the boolean operators ($\vee$,$\wedge$,$\neg$);*
— *for every type $t \in \beth$ and every $(P, N) \in t$:*

$$(\theta_1 \to \theta_2 \in P \cup N) \vee (\theta_1 \times \theta_2 \in P \cup N) \Rightarrow \tau(\theta_1) \in \beth \wedge \tau(\theta_2) \in \beth$$

**Theorem 3.8** *Any finite set of types is included in a socle.*

*Proof*: We just have to prove it for a set of the form $\tau(X)$ where $X$ is a basis. Indeed, any finite set of types is included in such a $\tau(X)$.

Let $A$ be the set of atoms that appear in an element of $\tau(X)$:

$$A = \bigcup_{t \in \tau(X)} \bigcup_{(P,N) \in t} P \cup N$$

It is clear that it is the smallest set such that $\tau(X) \subseteq \mathcal{B}A$. Since $X$ is a basis, we have $\tau(X) \subseteq \mathcal{B}FX$, hence $A \subseteq FX$.

We now show that the set $\beth = \mathcal{B}A$ is a socle. Since the set $X$ is finite, then so is $\tau(X)$, as well as $A$ and finally $\beth$. The stability of $\beth$ by boolean operators is trivial (as for any set of the form $\mathcal{B}\_$).

Let us take an atom $\theta_1 \to \theta_2$ that appears in one of the types of $\beth$. It is an element of $A$, hence of $FX$. We deduce that $\theta_1$ and $\theta_2$ are in $X$, therefore $\tau(\theta_i) \in \tau(X) \subseteq \mathcal{B}A = \beth$.

The case of $\theta_1 \times \theta_2$ is similar. □

This theorem ensures that if we inductively unfold a type $t$ by considering the descriptions of the type nodes that appear in its atoms, then we obtain a finite number of different types. Even better, we can saturate the resulting set by applying the boolean operators, and we still obtain a finite set. This finiteness property will be used to ensure the termination of algorithms.

# Chapter 4

# Semantic subtyping

In this chapter, we will show how to define a subtyping relation on the minimal type algebra $\mathbb{T}$ introduced in the previous chapter. We will follow a new approach based on a set-theoretic interpretation of types.

**Naive set-theoretic approaches** We want to *define* the subtyping relation $\leq$ by using a set-theoretic interpretation (in the sense of Definition 3.3), that is to say a function $\widehat{T} \to \mathcal{P}(D)$, for a certain set $D$, which interprets boolean operators in a set-theoretic way.

**Definition 4.1 (Subtyping)** *Let $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation. It defines a binary relation $\leq_{[\![\_]\!]}$ on $\widehat{T}$ by:*

$$t_1 \leq_{[\![\_]\!]} t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!]$$

*This relation is* **the subtyping relation induced by $[\![\_]\!]$**.

We clearly have the equivalence: $t_1 \leq_{[\![\_]\!]} t_2 \iff [\![t_1 \backslash t_2]\!] = \emptyset$. In other words, to study this subtyping relation, it is enough to focus on the unary predicate "$[\![\_]\!] = \emptyset$".

In order to define this subtyping relation, we have to choose the set $D$, and the way to interpret the atoms $T_A$.

A natural choice for $D$ would be to take the set of values of the calculus we want to define (see Chapter 5), to interpret the type $t$ as the set of values of that type, and to check that we have indeed defined a set-theoretic interpretation We would write: $[\![t]\!] = \{v \mid \vdash v : t\}$. It implies that we already have the type system of the language, but it is precisely in order to define it that we need the subtyping relation. In fact, we only need the type of values, and not of arbitrary expressions. We need to know, given an atom $a \in T_A$ and a value $v$, if $v$ is of type $a$ or not. The problem comes from functions. If $v$ is a $\lambda$-abstraction $\lambda x.e$ and $a$ an arrow type $\theta_1 \to \theta_2$, we would like to say that $v$ has type $a$ if and only if $x : \tau(\theta_1) \vdash e : \tau(\theta_2)$, but then we would need the type system of the whole language. Moreover, this definition poses a type safety issue. In a calculus with subtyping, the type of an expression becomes more precise during execution. If the abstraction above does not have type $a$, it only means that the type system is not powerful enough to prove that $e$ has type $\tau(\theta_2)$ under the assumption

$x : \tau(\theta_1)$; in that case, the value $v$ must have type $\neg a$ (to have a set-theoretic interpretation of types). Now assume that we can reduce the body of $v$ [1], in order to obtain a new values $v' = \lambda x.e'$. It may be then that the system is able to prove that $e'$ has type $\tau(\theta_2)$, hence $v'$ has type $a$, but it must have type $\neg a$, as well as the abstraction from which it is derived. Therefore we have a value of type $a$ and $\neg a$, which is impossible with a set-theoretic interpretation.

Another possibility would be to say that the abstraction $\lambda x.e$ has type $\theta_1 \to \theta_2$ if and only if all the values that can be obtained by reducing an expression $e[x \mapsto v]$ with $v$ of type $\theta_1$ are of type $\theta_2$, and that no type error can appear during those reductions. There are also problems with that. On the one hand, we still rely on the type system of the calculus (which is not yet defined, and whose definition will depend on the subtyping relation). On the other hand, the dynamic semantics will itself depend on the subtyping relation (via the pattern matching operation, or simply by the dynamic type-check operation), hence this definition is unfounded. Therefore this only introduces an additional circularity loop (between typing, subtyping and dynamic semantics).

In the calculus we will introduce, abstractions are explicitly annotated by their type. As we want to have overloaded functions, an abstraction is annotated by a finite set of arrow types $\theta_1 \to \theta'_1; \ldots; \theta_n \to \theta'_n$. This abstraction has the atomic type $a = \theta \to \theta'$ if and only if $a$ is a super-type of the intersection $\bigwedge_{i=1..n} \theta_i \to \theta'_i$ (and therefore it has type $\neg a$ if that is not the case). Therefore this typing does not really depend on the body of the abstraction (although we assume that it is well-typed). But we just moved the problem, because we define the predicate $\vdash v : a$ (when $v$ is an abstraction and $a$ an arrow type) using the subtyping relation that we want to define.

**Idea of the proposed solution** The naive approaches presented above all fail. We propose a solution that is to relax the link between the set $D$, used to interpret the types, and the set of values of the calculus that we want to type. We want to be able to interpret the types as sets of values (and subtyping as inclusion between the denoted sets). But we have seen that it is problematic to start from this property to define subtyping. In fact, the precise nature of the elements of $D$ does not matter: this set is only used to define the relation $\leq$. So the choice for $D$ is free enough, the only constraint being that we want to interpret the type atoms in a way that produces a subtyping relation that makes sense with regard to the values.

We will define a concept of *model* that captures this intuition. A model is a way to interpret the types in a set-theoretic way so that the induced subtyping relation corresponds to our intuition on the types and calculus.

Starting from a model, we can define a subtyping relation, and use it to define the type system of the calculus. It is then possible to define *afterwards* a new interpretation of types as sets of values, by using this type system (we take $[\![t]\!] = \{v \mid \vdash v : t\}$). The criteria used to validate our approach is to verify that this interpretation induces the *same* subtyping relation as the model we started from. Thus, we have come full circle, as even if we have *a priori* broken the link between subtyping and values (by defining subtyping via a model), this link is restored *afterwards*.

[1] In the calculus that we are about to introduce, it is not possible to reduce inside abstractions, but it could be done without giving up type safety .

## 4.1 Base types

For every base type $b \in \mathbb{B}$, we assume given a set $\mathbb{B}[\![b]\!]$, whose elements are called the constants of type $b$. These constants belong to the calculus that we want to study.

The sets $\mathbb{B}[\![b]\!]$ are not necessarily two-to-two disjoint: the same constant can have several different base types. For example, in $\mathbb{C}$Duce, the constant 0 has the singleton type 0, the type `Int`, and any interval type that contains 0, ... However, we assume that for any constant $c$, there exists a singleton base type $b_c \in \mathbb{B}$, that verifies:

$$[\![b_c]\!] = \{c\}$$

This implies:

$$\forall b \in \mathbb{B}.\ c \in \mathbb{B}[\![b]\!] \iff \mathbb{B}[\![b_c]\!] \subseteq \mathbb{B}[\![b]\!] \iff \mathbb{B}[\![b_c]\!] \cap \mathbb{B}[\![b]\!] \neq \emptyset$$

We denote by $\mathcal{C} = \bigcup_{b \in \mathbb{B}} \mathbb{B}[\![b]\!]$ the set of constants.

## 4.2 Models

We want to define a set-theoretic model notion for the type algebra. A model has to be a set-theoretic interpretation (Definition 3.3), which ensures that the boolean operators are interpreted in a natural way. The interpretation of type constructors must also be constrained. For example, we can say that the constructor $\times$ must be interpreted as a set-theoretic cartesian product. The set $D$ in which the types are interpreted must then be such that $D \times D \subseteq D$. Things are more complicated for constructor $\to$. An interpretation of arrow types could be sets of set-theoretic functions. This would require that $D^D \subseteq D$, which is impossible for cardinality reasons. Furthermore, we want to interpret types as sets of values of the calculus, hence we need to interpret an arrow type as a set of $\lambda$-abstractions (which are syntactic objects, and not set-theoretic functions).

What we mean is that, in a model, the type constructors behave as if they were interpreted in the natural set-theoretic way (even if they are not actually interpreted that way) from the point of view of subtyping (or, equivalently, from the point of view of the predicate $[\![t]\!] = \emptyset$, since $[\![t]\!] \subseteq [\![s]\!] \iff [\![t \setminus s]\!] = \emptyset$ for a set-theoretic interpretation). The formal definition below formalizes this "as if" idea.

**Definition 4.2 (Extensional interpretation, model)** *Let $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation of $\widehat{T} = \mathcal{B}T_A$ in a set $D$. We write:*

$$\mathbb{E}D := \mathcal{C} + D \times D + \mathcal{P}(D \times D_\Omega)$$

*where $D_\Omega = D + \{\Omega\}$. If $X, Y \subseteq D$, we write:*

$$X \to Y := \{f \subseteq D \times D_\Omega \mid \forall (d, d') \in f.\ d \in X \Rightarrow d' \in Y\} \subseteq \mathcal{P}(D \times D_\Omega)$$

*The* **extensional interpretation** *associated to $[\![\_]\!]$ is then defined as the unique set-theoretic interpretation $\mathbb{E}[\![\_]\!] : \widehat{T} \to \mathcal{P}(\mathbb{E}D)$ such that:*

$$\begin{array}{lcll} \mathbb{E}[\![b]\!] & = & \mathbb{B}[\![b]\!] & \subseteq \mathcal{C} \\ \mathbb{E}[\![\theta_1 \times \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \times [\![\tau(\theta_2)]\!] & \subseteq D \times D \\ \mathbb{E}[\![\theta_1 \to \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \to [\![\tau(\theta_2)]\!] & \subseteq \mathcal{P}(D \times D_\Omega) \end{array}$$

*We say that* $[\![\_]\!]$ *is a* **model** *of the type algebra if:*

$$\forall t \in \widehat{T}.\ [\![t]\!] = \emptyset \iff \mathbb{E}[\![t]\!] = \emptyset$$

*We write* $\mathbb{E}_{\mathbf{basic}} D = \mathcal{C}$, $\mathbb{E}_{\mathbf{prod}} D = D \times D$, $\mathbb{E}_{\mathbf{fun}} D = \mathcal{P}(D \times D_\Omega)$.

The extensional interpretation associated with a set-theoretic interpretation gives a natural set-theoretic interpretation for type constructors: the products are interpreted as cartesian products, the arrows are interpreted as sets of functions. In fact, rather than taking functional graphs, we consider binary relations, or, equivalently, non-deterministic functions. This is motivated by the fact that the calculus we are going to introduce may actually possess non-deterministic traits (in the form of operators, or if we consider an extension with side effects). In addition, this choice simplifies the set-theoretic calculations, avoiding corner cases related to finite sets. Let us give an example (in which we identify types and type nodes): consider the arrow type $t \to (s_1 \vee s_2)$, with $s_1$ and $s_2$ disjoint, and $t$ non-empty. We are wondering if it is a subtype of $(t \to s_1) \vee (t \to s_2)$, that is, if any function of type $t \to (s_1 \vee s_2)$ is necessarily of type $t \to s_i$ for some $i$. If we consider non-deterministic functions, this is not the case (because the function may choose to output in $s_1$ or in $s_2$ in a non-deterministic way). But if we consider set-theoretic functions, this becomes true in a corner case, when $t$ is a singleton (because the value of the function for this point is in $s_1 \vee s_2$, so in one of the two types $s_i$, and then the function itself is in $t \to s_i$). Using non-deterministic functions avoids specific difficult-to-treat cases for the subtyping relation.

The special element $\Omega$, in the codomain of functions, represents a possible type error. If we remove $\Omega$ from the definition of a model, we immediately see that any arrow type is a subtype of $\mathbb{1} \to \mathbb{1}$, that is, any function can be applied to any argument. In previous [Fri01] work, we have studied such a system. The semantics of the associated calculus must indeed be able to apply any abstraction to any argument, without causing a type error. To do this, it is necessary when calling the function to test if the argument is in the domain (declared in the interface of the abstraction), and if this is not the case, it is necessary to avoid evaluating the body of the function (and it is appropriate to return any value, for example a particular value that reports an error). The system we are presenting here ensures that when we evaluate a well-typed application, the argument is in the domain of the function. This is an illustration of the following principle: the definition of the model must capture in a precise enough way the semantics of the calculus to be typed. We will later give (Section 5.9) another illustration of this principle.

The model definition constrains the "local" interpretation of type constructors. We lack a constraint to ensure that the subtyping induced by a model corresponds exactly to that of an interpretation of types as a set of values. Consider a recursive type $\tau(\theta) = \theta \times \theta$. Nothing in the definition of a model ensures that $[\![\tau(\theta)]\!]$ is the empty set (in fact, it is possible to build models for which this set is not empty). But a value $v$ of this type must be a pair $(v_1, v_2)$ where $v_1$ and $v_2$ are still of that type; hence we see that $v$ is necessarily an infinite tree, but the calculus that we are about to introduce does not allow us to build such infinite values. So we want the interpretation of $\tau(\theta)$ to be empty, and we will now introduce a concept of well-founded model to ensure that property.

**Definition 4.3** *A model* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ *is* **well-founded** *if there exists a well-founded order* $\lhd$ *over* $D$ *such that:*

$$\forall t \in \widehat{T}.\forall d \in [\![t]\!].\exists d' \in \mathbb{E}[\![t]\!].d' = (d_1, d_2) \Rightarrow (d_1 \lhd d) \wedge (d_2 \lhd d)$$

**Definition 4.4** *A model* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ *is* **structural** *if* $D$ *is the initial solution of an equation of the form:*

$$D = \mathcal{C} + D \times D + D_{\mathbf{fun}}$$

*for a certain set* $D_{\mathbf{fun}}$*, and if:*

$$\begin{array}{lcl} [\![b]\!] & = & \mathbb{E}[\![b]\!] \\ [\![\theta_1 \times \theta_2]\!] & = & \mathbb{E}[\![\theta_1 \times \theta_2]\!] \\ [\![\theta_1 \to \theta_2]\!] & \subseteq & D_{\mathbf{fun}} \end{array}$$

**Lemma 4.5** *A structural model is well-founded.*

> *Proof*: It is sufficient to consider the well-founded order $\lhd$ induced by the relations $d_1 \lhd (d_1, d_2)$ and $d_2 \lhd (d_1, d_2)$ for every $(d_1, d_2) \in D \times D$. □

## 4.3 Set-theoretic inclusions

To prepare the study of subtyping in the models, we need to establish some simple set-theoretic properties. The property expressed in the following lemma served as the basis for the top-down subtyping algorithm introduced in XDuce [Hos01].

**Lemma 4.6** *Let* $(X_i)_{i\in P}, (X_i)_{i\in N}$ *(resp.* $(Y_i)_{i\in P}, (Y_i)_{i\in N}$*) be two families of parts of a set* $D_1$ *(resp.* $D_2$*). Then:*

$$\left(\bigcap_{i\in P} X_i \times Y_i\right) \setminus \left(\bigcup_{i\in N} X_i \times Y_i\right) = \bigcup_{N'\subseteq N} \left(\bigcap_{i\in P} X_i \setminus \bigcup_{i\in N'} X_i\right) \times \left(\bigcap_{i\in P} Y_i \setminus \bigcup_{i\in N\setminus N'} Y_i\right)$$

*(with conventions:* $\bigcap_{i\in\emptyset} X_i \times Y_i = D_1 \times D_2$*;* $\bigcap_{i\in\emptyset} X_i = D_1$ *and* $\bigcap_{i\in\emptyset} Y_i = D_2$*) In particular:*

$$\bigcap_{i\in P} X_i \times Y_i \subseteq \bigcup_{i\in N} X_i \times Y_i$$
$$\Longleftrightarrow$$
$$\forall N' \subseteq N.\ \left(\bigcap_{i\in P} X_i \subseteq \bigcup_{i\in N'} X_i\right) \vee \left(\bigcap_{i\in P} Y_i \subseteq \bigcup_{i\in N\setminus N'} Y_i\right)$$

> *Proof*: We start by noticing that:
>
> $$\overline{X_i \times Y_i}^{D_1\times D_2} = \left(\overline{X_i}^{D_1} \times D_2\right) \cup \left(D_1 \times \overline{Y_i}^{D_2}\right)$$
>
> We deduce:
>
> $$\bigcap_{i\in N} \overline{X_i \times Y_i}^{D_1\times D_2} =$$
> $$\bigcup_{N'\subseteq N} \left(\bigcap_{i\in N'} \left(\overline{X_i}^{D_1} \times D_2\right) \cap \bigcap_{i\in N\setminus N'} \left(D_1 \times \overline{Y_i}^{D_2}\right)\right)$$
> $$\bigcup_{N'\subseteq N} \left(\bigcap_{i\in N'} \overline{X_i}^{D_1} \times \bigcap_{i\in N\setminus N'} \overline{Y_i}^{D_2}\right)$$

And finally:

$$\left(\bigcap_{i\in P} X_i \times Y_i\right) \cap \left(\bigcap_{i\in N} \overline{X_i \times Y_i}^{D_1\times D_2}\right) = \bigcup_{N'\subseteq N}\left(\left(\bigcap_{i\in P} X_i \cap \bigcap_{i\in N'} \overline{X_i}^{D_1}\right) \times \left(\bigcap_{i\in P} Y_i \cap \bigcap_{i\in N\setminus N'} \overline{Y_i}^{D_2}\right)\right)$$

□

We will now establish the equivalent of Lemma 4.6 but for arrow types. We start by decomposing the set-theoretic operator $\to$ in simpler operators: set of parts, complement, cartesian product.

**Lemma 4.7** *Let $X, Y \subseteq D$. Then:*

$$X \to Y = \mathcal{P}\left(\overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\right)$$

*Proof*: The result comes from a simple calculation:

$$\begin{array}{rcl} X \to Y & = & \{f \subseteq D \times D_\Omega \mid \forall (x,y) \in f.\ \neg(x \in X \wedge y \notin Y)\} \\ & = & \{f \subseteq D \times D_\Omega \mid f \cap X \times \overline{Y}^{D_\Omega} = \emptyset\} \\ & = & \{f \subseteq D \times D_\Omega \mid f \subseteq \overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\} \end{array}$$

□

**Lemma 4.8** *Let $(X_i)_{i\in P}$ and $(X_i)_{i\in N}$ be two families of parts of a set $D$. Then:*

$$\bigcap_{i\in P} \mathcal{P}(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}(X_i) \iff \exists i_o \in N.\ \bigcap_{i\in P} X_i \subseteq X_{i_0}$$

*Proof*: The $\Leftarrow$ implication is trivial. Let us show the converse implication, and so assume that $\bigcap_{i\in P} \mathcal{P}(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}(X_i)$. The set $\bigcap_{i\in P} X_i$ belongs to every $\mathcal{P}(X_i)$ for $i \in P$, hence it is in the union of the $\mathcal{P}(X_i)$ for $i \in N$. Therefore we can find an $i_0 \in N$ such that $\bigcap_{i\in P} X_i \in \mathcal{P}(X_{i_0})$, which concludes the proof. □

**Lemma 4.9** *Let $(X_i)_{i\in P}, (X_i)_{i\in N}$, $(Y_i)_{i\in P}, (Y_i)_{i\in N}$ be four families of parts of a set $D$. Then:*

$$\bigcap_{i\in P} X_i \to Y_i \subseteq \bigcup_{i\in N} X_i \to Y_i$$
$$\iff$$
$$\exists i_0 \in N.\ \forall P' \subseteq P.\ \left(X_{i_0} \subseteq \bigcup_{i\in P'} X_i\right) \vee \left\{ \begin{array}{l} P \neq P' \\ \left(\bigcap_{i\in P\setminus P'} Y_i \subseteq Y_{i_0}\right) \end{array} \right.$$

*(with the convention: $\bigcap_{i\in\emptyset} X_i \to Y_i = \mathcal{P}(D \times D_\Omega)$)*

*Proof*: Corollary of the Lemmas 4.7, 4.8, 4.6, by noticing that in the condition $\bigcap_{i\in P\setminus P'} Y_i \subseteq Y_{i_0}$ that appears, the convention is to interpret the intersection as $D_\Omega$ if $P = P'$, which makes the inclusion impossible. □

**Corollary 4.10** *Let $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ be a model, and $(\theta_i)_{i\in P}$, $(\theta_i)_{i\in N}$, $(\theta'_i)_{i\in P}$, $(\theta'_i)_{i\in N}$ be four families of type nodes. Then:*

$$\bigwedge_{i\in P} \theta_i \to \theta'_i \leq \bigvee_{i\in N} \theta_i \to \theta'_i$$
$$\Longleftrightarrow$$
$$\begin{cases} P \neq \emptyset \\ \exists i_0 \in N.\ \forall P' \subseteq P.\ \left(\tau(\theta_{i_0}) \leq \bigvee_{i\in P'} \tau(\theta_i)\right) \vee \begin{cases} P \neq P' \\ \left(\bigwedge_{i\in P\setminus P'} \tau(\theta'_i) \leq \tau(\theta'_{i_0})\right) \end{cases} \end{cases}$$

## 4.4 Subtyping analysis

We now have all the necessary elements to express the predicate $\mathbb{E}[\![t]\!] = \emptyset$, based on the predicate $[\![t]\!] = \emptyset$, and thus to obtain a more effective definition to ensure that a set-theoretic interpretation is a model.

Let $t$ be a type. We have:

$$t \simeq \bigvee_{(P,N)\in t} \bigwedge_{a\in P} a \wedge \bigwedge_{a\in N} \neg a$$

The extensional interpretation of this type is empty if and only if the one of every term of the union is empty. In particular, this is the case for $(P, N) \in t$ as soon as $P$ contains two atoms of a different kind, since the $\mathbb{E}_u D$ are two-to-two disjoint (for $u \in \{\mathbf{basic}, \mathbf{prod}, \mathbf{fun}\}$). When all the atoms are products (resp. arrows), we can use Lemma 4.6 (resp. 4.9) to decompose the predicate $\mathbb{E}[\![\bigwedge_{a\in P} a \wedge \bigwedge_{a\in N} \neg a]\!] = \emptyset$ in a boolean formula over predicates of the form $[\![t]\!] = \emptyset$, for types $t$ obtained as boolean combinations of the $\tau(\theta)$, for the $\theta$ that appear in atoms $a$. This reasoning is formalized in the following definition and in Theorem 4.13.

**Definition 4.11** *Let $\mathcal{S} \subseteq \widehat{T}$. We define:*

$$\mathbb{E}\mathcal{S} = \{t \in \widehat{T} \mid \forall (P, N) \in t. \forall u.\ (P \subseteq T_u \Rightarrow C_u)\}$$

*where:*

$$C_{\mathbf{basic}} \quad ::= \quad \mathcal{C} \cap \bigcap_{b\in P} \mathbb{B}[\![b]\!] \subseteq \bigcup_{b\in N} \mathbb{B}[\![b]\!]$$

$$C_{\mathbf{prod}} \quad ::= \quad \forall N' \subseteq N \cap T_{\mathbf{prod}}. \begin{cases} \left(\bigwedge_{\theta_1\times\theta_2\in P} \tau(\theta_1)\right) \setminus \left(\bigvee_{\theta_1\times\theta_2\in N'} \tau(\theta_1)\right) \in \mathcal{S} \\ \vee \\ \left(\bigwedge_{\theta_1\times\theta_2\in P} \tau(\theta_2)\right) \setminus \left(\bigvee_{\theta_1\times\theta_2\in N\setminus N'} \tau(\theta_2)\right) \in \mathcal{S} \end{cases}$$

$$C_{\mathbf{fun}} \quad ::= \quad \exists \theta_1'\to\theta_2' \in N.\ \forall P' \subseteq P. \begin{cases} \tau(\theta_1') \setminus \left(\bigvee_{\theta_1\times\theta_2\in P'} \tau(\theta_1)\right) \in \mathcal{S} \\ \vee \\ \begin{cases} P \neq P' \\ \left(\bigwedge_{\theta_1\times\theta_2\in P\setminus P'} \tau(\theta_2)\right) \setminus \tau(\theta_2') \in \mathcal{S} \end{cases} \end{cases}$$

*We say that $\mathcal{S}$ is a* **simulation** *if:*

$$\mathcal{S} \subseteq \mathbb{E}\mathcal{S}$$

**Remark 4.12** *The definition of $\mathbb{E}\mathcal{S}$ does not depend on a fixed model or a set-theoretic interpretation. It is a purely axiomatic definition. For all type $t$ and $(P, N) \in t$, there exists at least one $u$ such that $P \subseteq T_u$, except if $P = \emptyset$, in which case the three conditions $C_u$ have to be verified.*

**Theorem 4.13** *Let $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation. Let us write:*

$$\mathcal{S} = \{t \mid [\![t]\!] = \emptyset\}$$

*Then:*

$$\mathbb{E}\mathcal{S} = \{t \mid \mathbb{E}[\![t]\!] = \emptyset\}$$

*Proof*: Let $t \in \widehat{T}$. We have:

$$\mathbb{E}[\![t]\!] = \bigcup_{(P,N)\in t} \bigcap_{a\in P} \mathbb{E}[\![a]\!] \setminus \bigcup_{a\in N} \mathbb{E}[\![a]\!]$$

Hence:

$$\mathbb{E}[\![t]\!] = \emptyset \iff \forall (P, N) \in t. \bigcap_{a\in P} \mathbb{E}[\![a]\!] \subseteq \bigcup_{a\in N} \mathbb{E}[\![a]\!]$$

Let us decompose on each universe:

$$\bigcap_{a\in P} \mathbb{E}[\![a]\!] \subseteq \bigcup_{a\in N} \mathbb{E}[\![a]\!] \iff \forall u.\ \mathbb{E}_u D \cap \bigcap_{a\in P} \mathbb{E}[\![a]\!] \subseteq \mathbb{E}_u D \cap \bigcup_{a\in N} \mathbb{E}[\![a]\!]$$

Note that for any atom $a$, if $a \in T_u$, then $\mathbb{E}_u D \cap \mathbb{E}[\![a]\!] = \mathbb{E}[\![a]\!]$, and if $a \notin T_u$, then $\mathbb{E}_u D \cap \mathbb{E}[\![a]\!] = \emptyset$. Hence:

$$\mathbb{E}_u D \cap \bigcap_{a\in P} \mathbb{E}[\![a]\!] \subseteq \mathbb{E}_u D \cap \bigcup_{a\in N} \mathbb{E}[\![a]\!]$$

$$\iff$$

$$\left( P \subseteq T_u \Rightarrow \bigcap_{a \in P} \mathbb{E}[\![a]\!] \subseteq \bigcup_{a \in N \cap T_u} \mathbb{E}[\![a]\!] \right)$$

We conclude with Lemmas 4.9 et 4.6. □

**Corollary 4.14** *With the notations of the previous theorem, the interpretation* $[\![\_]\!]$ *is a model if and only if* $\mathcal{S} = \mathbb{E}\mathcal{S}$.

**Corollary 4.15** *Let* $[\![\_]\!]_1 : \widehat{T} \to D_1$ *be a model and* $[\![\_]\!]_2 : \widehat{T} \to D_2$ *be a set-theoretic interpretation. Then the two assertions below are equivalent:*

— $[\![\_]\!]_2$ *is a model and it induces the same subtyping relation as* $[\![\_]\!]_1$

— *for any type* $t \in \widehat{T}$, $[\![t]\!]_1 = \emptyset \iff [\![t]\!]_2 = \emptyset$

This corollary says that whether a set-theoretic interpretation is a model or not depends only on the subtyping relation it induces. In other words, as expected, our model definition only constrains the subtyping relation induced by the set-theoretic interpretation, and not the nature of the elements of the set used to interpret the types.

## 4.5 Universal model

**Definition 4.16 (Universal model)** *A model* $[\![\_]\!]^0$ *is* **universal** *if , for every model* $[\![\_]\!]$*, we have:*

$$\forall t \in \widehat{T}.\ [\![t]\!] = \emptyset \Rightarrow [\![t]\!]^0 = \emptyset$$

A model is universal if it induces the largest possible subtyping relation. Intuitively, having the largest possible subtyping relation allows for more expressions to be typed in the calculus of the next chapter, in view of the subsumption rule (although, formally, this property is false in the type system we are going to introduce, because of the abstraction typing rule).

In this section we will show how to build a universal model. Of course, all universal models induce the same subtyping relation.

The property that defines the model notion imposes a correspondence between the set-theoretic interpretations $[\![\_]\!]$ and $\mathbb{E}[\![\_]\!]$. To build a model, it is natural to consider the following equation:

$$D = \mathbb{E}D$$

For mere cardinality reasons, it is not possible to build a set $D$ that verifies this equation. The problem comes from the fact that the cardinal of $\mathcal{P}(D \times D_\Omega)$ is strictly larger than the cardinal of $D$. The following Lemmas prove that by only considering finite graphs, we do not change set-theoretic inclusion relations, and this is what counts in order to get a model.

**Lemma 4.17** *If we write* $X \to_f Y = (X \to Y) \cap \mathcal{P}_f(D \times D_\Omega)$*, then:*

$$X \to_f Y = \mathcal{P}_f \left( \overline{X \times \overline{Y}^{D_\Omega}}^{D \times D_\Omega} \right)$$

**Lemma 4.18** *Let $(X_i)_{i\in P}$ and $(X_i)_{i\in N}$ be two finite families of parts of a set $D$. Then:*

$$\bigcap_{i\in P} \mathcal{P}_f(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}_f(X_i) \iff \bigcap_{i\in P} \mathcal{P}(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}(X_i)$$

*Proof*: The implication $\Leftarrow$ is immediate. Let us prove the implication $\Rightarrow$. We suppose that every finite part of $X = \bigcap_{i\in P} X_i$ is included in a certain $\mathcal{P}(X_{i_0})$ with $i_0 \in N$. If that was not the case for $X$ itself, we could find for each $i_0 \in N$ a certain element $x_{i_0} \in X \backslash X_{i_0}$, and we would obtain a contradiction by considering the set of those $x_{i_0}$ (which is a finite part of $X$).Then □

**Lemma 4.19** *Let $(X_i)_{i\in P}, (X_i)_{i\in N}$, $(Y_i)_{i\in P}, (Y_i)_{i\in N}$ be four finite families of parts of a set $D$. Then:*

$$\bigcap_{i\in P} X_i \to_f Y_i \subseteq \bigcup_{i\in N} X_i \to_f Y_i \quad\iff\quad \bigcap_{i\in P} X_i \to Y_i \subseteq \bigcup_{i\in N} X_i \to Y_i$$

*(with the convention: $\bigcap_{i\in\emptyset} X_i \to_f Y_i = \mathcal{P}_f(D \times D_\Omega)$)*

*Proof*: Corollary of Lemmas 4.17 and 4.18. □

**Definition 4.20** *Let $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation in a set $D$. We take:*

$$\mathbb{E}_f D := \mathcal{C} + D \times D + \mathcal{P}_f(D \times D_\Omega)$$

*and we then define the* **finite extensional interpretation** *associated with $[\![\_]\!]$ as the unique set-theoretic interpretation $\mathbb{E}_f[\![\_]\!] : \widehat{T} \to \mathcal{P}(\mathbb{E}D)$ such that:*

$$\begin{array}{lcll} \mathbb{E}_f[\![b]\!] & = & \mathbb{B}[\![b]\!] & \subseteq \mathcal{C} \\ \mathbb{E}_f[\![\theta_1 \times \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \times [\![\tau(\theta_2)]\!] & \subseteq D \times D \\ \mathbb{E}_f[\![\theta_1 \to \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \to_f [\![\tau(\theta_2)]\!] & \subseteq \mathcal{P}_f(D \times D_\Omega) \end{array}$$

**Lemma 4.21** *Let $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation. Then:*

$$\mathbb{E}[\![t]\!] = \emptyset \iff \mathbb{E}_f[\![t]\!] = \emptyset$$

This suggest looking at the equation $D = \mathbb{E}_f D$, which, unlike equation $D = \mathbb{E}D$ does have solutions. We will now consider the initial solution $D^0$, consisting of finite "terms" generated by the productions:

$$\begin{array}{rcl} d \in D^0 & ::= & c \\ & | & (d_1, d_2) \\ & | & \{(d_1, d'_1), \ldots, (d_n, d'_n)\} \end{array}$$

In the last production, the $d'_i$ are elements of $D^0_\Omega = D^0 + \{\Omega\}$. We indeed have $D = \mathbb{E}_f D$.

**Theorem 4.22** *There exists a unique set-theoretic interpretation $[\![\_]\!]^0 : \widehat{T} \to \mathcal{P}(D^0)$ that verifies $[\![t]\!]^0 = \mathbb{E}_f[\![t]\!]^0$. It is a structural model.*

*Proof*: Lemma 3.4 translate the condition "$[\![\_]\!]^0$ is a set-theoretic interpretation and $[\![\_]\!]^0 = \mathbb{E}_f[\![\_]\!]^0$" into a recursive definition of the truth value of the assertion "$d \in [\![t]\!]^0$", by induction over the structure of $d$ and simultaneously for every $t \in \widehat{T}$:

$$
\begin{array}{l}
c \in [\![t]\!]^0 \iff \\
\quad \exists (P,N) \in t.\ (P \subseteq T_{\mathbf{basic}}) \wedge \\
\quad\quad (\forall b \in P. c \in \mathbb{B}[\![b]\!]) \wedge \\
\quad\quad (\forall b \in N. c \notin \mathbb{B}[\![b]\!]) \\
(d_1,d_2) \in [\![t]\!]^0 \iff \\
\quad \exists (P,N) \in t.\ (P \subseteq T_{\mathbf{prod}}) \wedge \\
\quad\quad (\forall \theta_1 \times \theta_2 \in P. d_1 \in [\![\tau(\theta_1)]\!]^0 \wedge d_2 \in [\![\tau(\theta_2)]\!]^0) \wedge \\
\quad\quad (\forall \theta_1 \times \theta_2 \in N. d_1 \notin [\![\tau(\theta_1)]\!]^0 \vee d_2 \notin [\![\tau(\theta_2)]\!]^0) \\
f \in [\![t]\!]^0 \iff \\
\quad \exists (P,N) \in t.\ (P \subseteq T_{\mathbf{fun}}) \wedge \\
\quad\quad (\forall \theta_1 \to \theta_2 \in P. \forall (d,d') \in f.\ d \in [\![\tau(\theta_1)]\!]^0 \Rightarrow d' \in [\![\tau(\theta_2)]\!]^0) \wedge \\
\quad\quad (\forall \theta_1 \to \theta_2 \in N. \exists (d,d') \in f.\ d \in [\![\tau(\theta_1)]\!]^0 \wedge d' \notin [\![\tau(\theta_2)]\!]^0)
\end{array}
$$

This proves the existence and the unicity of a set-theoretic interpretation $[\![\_]\!]^0$ that verifies condition $[\![\_]\!]^0 = \mathbb{E}_f[\![\_]\!]^0$. We immediately see that this is a model with Corollary 4.14 and Lemma 4.21. It is clearly structural. □

**Lemma 4.23** *Let $\mathcal{S} \subseteq \widehat{T}$ be a simulation and $t \in \mathcal{S}$. Then $[\![t]\!]^0 = \emptyset$.*

*Proof*: Let $\mathcal{S}$ be a simulation. We will show by induction over the structure of $d \in D^0$ the following property:

$$\forall t \in \widehat{T}.\ d \in [\![t]\!]^0 \Rightarrow t \notin \mathcal{S}$$

So let us assume that this property is verified for every subelement of a certain $d \in D^0$. Let us take a type $t \in \widehat{T}$ such that $d \in [\![t]\!]^0$. We have to show that $t \notin \mathcal{S}$. Since $\mathcal{S}$ is a simulation, we only have to prove that starting from hypothesis $t \in \mathbb{E}\mathcal{S}$, we arrive to a contradiction.

Let $u \in \{\mathbf{basic}, \mathbf{fun}, \mathbf{prod}\}$ be the universe of $d$. We choose $(P,N) \in t$ such that:

$$d \in \bigcap_{a \in P} [\![a]\!]^0 \setminus \bigcup_{a \in N} [\![a]\!]^0$$

For any $a \in P$, we have $d \in [\![a]\!]^0$, hence $P \subseteq T_u$.

Therefore the property $C_u$ of Definition 4.11 is verified for the pair $(P,N)$. Let us distinguish between the value of $u$.

For $u = \mathbf{basic}$, we directly obtain a contradiction.

For $u = \mathbf{prod}$, let us $d = (d_1, d_2)$. We can write:

$$(d_1,d_2) \in \bigcap_{\theta_1 \times \theta_2 \in P} [\![\tau(\theta_1)]\!]^0 \times [\![\tau(\theta_2)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in N} [\![\tau(\theta_1)]\!]^0 \times [\![\tau(\theta_2)]\!]^0$$

According to Lemma 4.6, we can find $N' \subseteq N \cap T_{\mathbf{prod}}$ such that:

$$d_1 \in \bigcap_{\theta_1 \times \theta_2 \in P} [\![\tau(\theta_1)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in N'} [\![\tau(\theta_1)]\!]^0$$

and

$$d_2 \in \bigcap_{\theta_1 \times \theta_2 \in P} [\![\tau(\theta_2)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in N \setminus N'} [\![\tau(\theta_2)]\!]^0$$

By applying the induction hypothesis to $d_1$ and to the type $t_1 = \bigwedge_{\theta_1 \times \theta_2 \in P} \tau(\theta_1) \setminus \bigvee_{\theta_1 \times \theta_2 \in N'} \tau(\theta_1)$, we obtain: $t_1 \notin \mathcal{S}$. Similarly, for $t_2 = \bigwedge_{\theta_1 \times \theta_2 \in P} \tau(\theta_2) \setminus \bigvee_{\theta_1 \times \theta_2 \in N \setminus N'} \tau(\theta_2)$. We indeed obtain a contradiction with condition $C_{\mathbf{prod}}$.

For any $u = \mathbf{fun}$, let us write $d = \{(d_1, d'_1), \ldots, (d_n, d'_n)\}$. Let us choose an element $\theta'_1 \to \theta'_2 \in N$ such as given by condition $C_{\mathbf{fun}}$. Since $d \notin [\![\theta'_1 \to \theta'_2]\!]^0 = [\![\tau(\theta'_1)]\!]^0 \to [\![\tau(\theta'_2)]\!]^0$, we can find $i$ such that:

$$(d_i, d'_i) \in [\![\tau(\theta'_1)]\!]^0 \times \overline{[\![\tau(\theta'_2)]\!]^0}^{D^0_\Omega}$$

Therefore this pair $(d_i, d'_i)$ is in the set:

$$[\![\tau(\theta'_1)]\!]^0 \times \overline{[\![\tau(\theta'_2)]\!]^0}^{D^0_\Omega} \setminus \bigcup_{\theta_1 \to \theta_2 \in P} [\![\tau(\theta_1)]\!]^0 \times \overline{[\![\tau(\theta_2)]\!]^0}^{D^0_\Omega}$$

By reasoning similarly as for the case of products, we see that there exists $P' \subseteq P$ such that:

$$d_i \in [\![\tau(\theta'_1)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in P'} [\![\tau(\theta_1)]\!]^0$$

and

$$d'_i \in D^0_\Omega \cap \left( \bigcap_{\theta_1 \times \theta_2 \in P \setminus P'} [\![\tau(\theta_2)]\!]^0 \right) \setminus [\![\tau(\theta'_2)]\!]^0$$

By applying the induction hypothesis to $d_i$ and to the type $t_1 = \tau(\theta'_1) \setminus \bigvee_{\theta_1 \times \theta_2 \in P'} \tau(\theta_1)$, we obtain $t_1 \notin \mathcal{S}$. In addition, if $P' \neq P$, then $d'_i \neq \Omega$, and we can apply the induction hypothesis to $d'_i$ and to the type $t_2 = \left( \bigwedge_{\theta_1 \times \theta_2 \in P \setminus P'} \tau(\theta_2) \right) \setminus \tau(\theta'_2)$. We have obtained a contradiction. □

**Theorem 4.24** *The model $[\![\_]\!]^0$ is universal.*

*Proof*: Consequence of Corollary 4.14 and of the previous lemma. □

By combining Corollary 4.14 and Lemma 4.23, we obtain:

**Lemma 4.25** *Let $t \in \widehat{T}$. Then $[\![t]\!]^0 = \emptyset$ if and only if there exists a simulation $\mathcal{S}$ that contains $t$.*

The following Lemma will induce a more effective version of this property.

**Lemma 4.26** *The intersection of a simulation and a socle is still a simulation.*

| *Proof*: This is a direct consequence of Definitions 4.11 and 3.7. □

A simulation is a set of types stable by a certain set of decomposition rules (given by the definition of $\mathbb{E}\mathcal{S}$). Lemma 4.26 shows that in order to check that a type $t$ belongs to a certain simulation, we only have to consider a finite number of types (if $t$ is in a socle, the types introduced by the decomposition rules are still in that same socle). By using this Lemma, we obtain a more effective version of Lemma 4.25.

**Lemma 4.27** *Consider $t \in \widehat{T}$, and let $\sqsupseteq$ be a socle that contains it. Then $[\![t]\!]^0 = \emptyset$ if and only if there exists a simulation $\mathcal{S} \subseteq \sqsupseteq$ that contains $t$.*

The existence of a socle that contains $t$ is guaranteed by Theorem 3.8. Since a socle is a finite set, we can enumerate all its subsets and check if there is one that is a simulation. This gives us a naive algorithm to compute the subtyping induced by the universal model. In Chapter 7, we will give more efficient algorithms.

## 4.6 Non-universal models

In this section, we will show that there exists a model that induces a different subtyping relation than the one induced by model $[\![\_]\!]^0$. Even better, we will build a structural model that verifies this property. Building a non-universal model allows us to see that the definition of a model leaves a certain amount of freedom, because it does not completely axiomatize the subtyping relation.

Consider a node $\theta$ such that:

$$\tau(\theta) = (\mathbb{0} \to \mathbb{0}) \setminus (\theta \to \mathbb{0})$$

(here $\mathbb{0}$ is used as a type node , which represents the empty type - see Section 3.3) and let us write $t_0 = \tau(\theta)$. A calculation shows us that :

$$[\![t_0]\!]^0 = \{\{(d_1, d'_1), \ldots, (d_n, d'_n)\} \mid \exists i.\ d_i \in [\![t_0]\!]^0\}$$

Since the terms of $D^0$ are finite, we immediately deduce that $[\![t_0]\!]^0 = \emptyset$ hence $t_0 \leq_{[\![\_]\!]^0} \mathbb{0}$. We will now build a model $[\![\_]\!] : \widehat{T} \to D$ such that $[\![t_0]\!] \neq \emptyset$. For that, we will add to $D^0$ "functions" whose graph is a tree of infinite depth. Consider the set $D^1$ of terms generated by the following productions:

$$\begin{array}{rcl} d \in D^1 & ::= & c \\ & | & (d_1, d_2) \\ & | & \{(d_1, d'_1), \ldots, (d_n, d'_n)\} \\ & | & n \end{array}$$

where the $d_i$ denote elements of $D^1$, the $d'_i$ elements of $D^1_\Omega = D^1 + \{\Omega\}$ and $n$ an integer ($n \in \mathbb{Z}$). Define $D$ as the quotient of $D^1$ by equations $n \simeq \{(n-1, \Omega)\}$ for any $n \in \mathbb{Z}$. It verifies the set-theoretic equation:

$$D = \mathbb{E}_f D$$

**Theorem 4.28** *There exists a model* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ *that verifies:*

$$\begin{array}{lcl} [\![\_]\!] & = & \mathbb{E}_f[\![\_]\!] \\ [\![t_0]\!] & \neq & \emptyset \end{array}$$

The proof of the theorem uses the following technical lemma.

**Lemma 4.29** *Let* $X$ *be a set,* $f$ *a function* $\mathcal{P}(X) \to \mathcal{P}(X)$*. Assume that any element* $x \in X$ *is in a certain finite subset* $\mathcal{S}$ *of* $X$ *verifying:*

$$\forall Y \subseteq X.\ f(Y) \cap \mathcal{S} = f(Y \cap \mathcal{S})$$

*Let* $Z \subseteq X$*. Then there exists a unique sequence* $(Y_n)_{n\in\mathbb{Z}}$ *such that:*
- $\forall n \in \mathbb{Z}.\ Y_{n+1} = f(Y_n)$
- $\exists n_0 \in \mathbb{N}. \forall n \geq n_0.\ Y_n = f^n(Z)$
- *for all* $x \in X$*, the sequence of predicates* $(x \in Y_n)_{n\in\mathbb{N}}$ *is periodic*

> *Proof*: Consider the sequence $(Z_n)_{n\in\mathbb{N}}$ defined by $Z_0 = Z$ and $Z_{n+1} = f(Z_n)$.
>
> Let us treat first the case where $X$ is finite. The sequence $(Z_n)$ has its value in the set $\mathcal{P}(X)$ which is finite; hence it is ultimately periodic. Therefore there is a unique periodic sequence $(Y_n)_{n\in\mathbb{Z}}$ that ultimately corresponds to $(Z_n)_{n\in\mathbb{N}}$. This sequence verifies the induction formula $Y_{n+1} = Y_n$ for all $n \in \mathbb{Z}$.
>
> Let us move on to the case where $X$ is infinite. If $\mathcal{S}$ is a set as in the statement, then we can apply what precedes to the sequence $(Z_n \cap \mathcal{S})_{n\in\mathbb{N}}$. It is also obtained by iteration of the function $f$, and this brings us back to the finite case. This allows us to build a sequence $(Y_n \cap \mathcal{S})_{n\in\mathbb{Z}}$. By unicity in the finite case, this defines indeed a sequence $(Y_n)_{n\in\mathbb{Z}}$. □

Let us move on to the proof of the theorem.

> *Proof*: We are bringing back the proof of Theorem 4.22. The equation $[\![\_]\!] = \mathbb{E}_f[\![\_]\!]$ gives a recursive definition of the truth value of the assertion "$d \in [\![t]\!]$". Therefore this induction is ill-founded, because of the elements $n \in \mathbb{Z}$. Hence, to define a model, we simply have to choose sets $T_n = \{t \in \widehat{T} \mid n \in [\![t]\!]\}$ for all $n \in \mathbb{Z}$, in a way that is compatible with the equality $n = \{(n-1, \Omega)\}$ in $D$. Once this is done, the equations of the proof of Theorem 4.22 clearly define the truth value of the assertion "$d \in [\![t]\!]$", and we have obtained a model.
>
> Note that we must have:
>
> $$\{(n-1, \Omega)\} \in [\![\theta_1 \to \theta_2]\!] \iff n - 1 \notin [\![\tau(\theta_1)]\!]$$
>
> and therefore:
>
> $$n \in [\![\theta_1 \to \theta_2]\!] \iff n - 1 \notin [\![\tau(\theta_1)]\!]$$
>
> which gives, for every $t \in \widehat{T}$:
>
> $$t \in T_n \iff \exists (P, N) \in t. \begin{cases} P \subseteq T_{\mathbf{fun}} \\ \forall \theta_1 \to \theta_2 \in P.\ \tau(\theta_1) \notin T_{n-1} \\ \forall \theta_1 \to \theta_2 \in N.\ \tau(\theta_1) \in T_{n-1} \end{cases}$$

Let us recall that the condition over the $T_n$ is expressed by:

$$(*) \quad \forall n \in \mathbb{Z}.\ T_n = f(T_{n-1})$$

where $f$ is a certain operator $\mathcal{P}(\widehat{T}) \to \mathcal{P}(\widehat{T})$. For any sequence $(T_n)_{n\in\mathbb{Z}}$ that verifies this condition $(*)$, we obtain a unique model.

To build such a sequence, we apply the lemma. Indeed, if $\sqsupset$ is a socle, then $f(T')\cap\sqsupset = f(T'\cap\sqsupset)$, for any $T' \subseteq \widehat{T}$. The lemma does give the existence of a sequence $(T_n)_{n\in\mathbb{Z}}$ that verifies condition $(*)$. In addition, we can arbitrarily fix the set $Z \subseteq \widehat{T}$.

Note that, for any $Z \subseteq \widehat{T}$:

$$t_0 \in f(Z) \iff (\mathbb{0} \notin Z) \wedge (t_0 \in Z)$$

and

$$\mathbb{0} \notin f(Z)$$

So if we choose $Z$ such that $t_0 \in Z$, and $\mathbb{0} \notin Z$, then, for any $n \in \mathbb{N}$, $t_0 \in f^n(Z)$. We deduce that $t_0 \in T_n$ for any $n$, which means that $\mathbb{Z} \subseteq [\![t_0]\!]$ (and in particular $[\![t_0]\!] \neq \emptyset$). □

**Remark 4.30** *If we have several nodes $\theta$ such that $\tau(\theta) = (\mathbb{0} \to \mathbb{0})\backslash(\theta \to \mathbb{0})$, the proof of the theorem shows that for each of those we can independently choose that either $[\![\tau(\theta)]\!] \cap \mathbb{Z} = \emptyset$ or $\mathbb{Z} \subseteq [\![\tau(\theta)]\!]$. In other words, several types defined by the same equation do not necessarily have the same interpretation in a non-universal model.*

**Remark 4.31** *A similar construction to $D^1$ allows for a model in which the type $\tau(\theta) = \theta\times\theta$ is non-empty. Of course, such a model is not well founded.*

## 4.7 Notation conventions

So far, we have carefully made sure to distinguish type expressions $\widehat{T}$, written as $t$, $t_1$, $t_2$, $s$, ..., from type nodes $\mathcal{T}$, written as $\theta$, $\theta_1$,$\theta_2$, ...

To simplify the notations, we are now adopting conventions that will make it easier for us to mix the two notions.

First of all, we overload the semantic brackets so they apply directly to the nodes:

$$[\![\theta]\!] := [\![\tau(\theta)]\!]$$

More generally, we allow the nodes to appear in any context that is expecting a type expression, the node being interpreted as $\tau(\theta)$. In particular, this allows the use of a node as argument (left or right) of a subtyping relation $\leq$.

Conversely, we allow ourselves to use a type expression $t$ when a node is expected. There are two cases:

— It is a "capture" context. For example, in the phrase "if $a$ is a product atom, let us write $a = t_1\times t_2$", the meta-variables $t_1$ and $t_2$ operate as binders. Formally, this phrase means that "if $a$ is a product atom $\theta_1\times\theta_2$, we fix $t_1 = \tau(\theta_1)$ and $t_2 = \tau(\theta_2)$".

— When a type expression $t$ appears where a node is expected, outside of a capture context, we have to *choose* a node $\theta$ such that $\tau(\theta) = t$. This is

always possible because $\theta$ is surjective. Every time we use this convention, it will be clear that the choice of $\theta$ does not ultimately change the object we are talking about or which we are defining. For example, if we write $t_1 \to s_1 \leq t_2 \to s_2$, the truth value of this assertion does not depend on the choice of the nodes used to represent $t_1, t_2, s_1, s_2$.

## 4.8 Semantic reasonings

In this section, we show how the approach we have chosen to define subtyping, through a set-theoretic model notion, makes it easy to obtain properties on the subtyping relation, without even considering an eventual subtyping algorithm.

We fix a model, and we denote by $\leq$ instead of $\leq_{[\![\_]\!]}$ the subtyping relation it induces. We assume that this subtyping relation is decidable (this is the case for universal models when subtyping is decidable for base types). We will denote by $\simeq$ the equivalence induced by this subtyping: $t_1 \simeq t_2 \iff (t_1 \leq t_2) \wedge (t_2 \leq t_1)$

**Lemma 4.32** *The relation $\leq$ is preorder (transitive and reflexive). The relation $\simeq$ is an equivalence relation.*

**Lemma 4.33** *The operators $\vee$ and $\wedge$ are covariant in both their arguments.*

### 4.8.1 Decompositions

**Lemma 4.34** $(t_1 \times t_2) \wedge (t'_1 \times t'_2) \simeq (t_1 \wedge t'_1) \times (t_2 \wedge t'_2)$

*Proof*: This kind of property can directly be observed on the extensional interpretation, by using the fact that $[\![\_]\!]$ and $\mathbb{E}[\![\_]\!]$ are both set-theoretic interpretations. We detail the proof as an example, by taking $t''_i = t_i \wedge t'_i$:

$$\begin{aligned}
\mathbb{E}[\![t''_1 \times t''_2]\!] &= [\![t''_1]\!] \times [\![t''_2]\!] \\
&= ([\![t_1]\!] \cap [\![t'_1]\!] \times ([\![t_2]\!] \cap [\![t'_2]\!]) \\
&= ([\![t_1]\!] \times [\![t_2]\!]) \cap ([\![t'_1]\!] \times [\![t'_2]\!]) \\
&= \mathbb{E}[\![t_1 \times t'_1]\!] \cap \mathbb{E}[\![t_2 \times t'_2]\!] \\
&= \mathbb{E}[\![(t_1 \times t'_1) \wedge (t_2 \times t'_2)]\!]
\end{aligned}$$

□

Here is another example of a property that can be "seen" on the extensional interpretation:

**Lemma 4.35** $(t_1 \times t_2) \setminus (t'_1 \times t'_2) \simeq (t_1 \setminus t'_1) \times t_2 \vee t_1 \times (t_2 \setminus t'_2)$

By combining the two previous lemmas, we obtain the following result, which makes it possible to decompose any type $t$ such that $t \leq \mathbb{1} \times \mathbb{1}$ (that is, $\mathbb{E}[\![t]\!] \subseteq D \times D$) into a finite union of cartesian products.

**Theorem 4.36** *Let $t$ be a type such that $t \leq \mathbb{1} \times \mathbb{1}$. Then there exists a finite set of type pairs $\pi(t)$ such that:*

— $t \simeq \bigvee_{(t_1,t_2) \in \pi(t)} t_1 \times t_2$

— $\forall (t_1, t_2) \in \pi(t).\ t_1 \not\simeq \mathbb{0}, t_2 \not\simeq \mathbb{0}$
— *if* $\square$ *is a socle that contains* $t$*, then:* $\forall (t_1, t_2) \in \pi(t).\ t_1 \in \square, t_2 \in \square$

*We see* $\pi$ *as a partial function* $\widehat{T} \to \mathcal{P}_f(\widehat{T}^2)$*, defined on types* $t$ *such that* $t \leq \mathbb{1}\times\mathbb{1}$*.*

> *Proof*: We start by writing:
>
> $$t \simeq \bigvee_{(P,N)\in t \ \mid\ P\subseteq T_{\mathbf{prod}}} (\mathbb{1}\times\mathbb{1})\wedge \bigwedge_{a\in P} a\wedge \bigwedge_{a\in N\cap T_{\mathbf{prod}}} \neg a$$
>
> Then, by using Lemmas 4.34 and 4.35, we see that we can write $t$ in the form of a union of atoms of the form $t_1\times t_2$ where $t_1$ (resp. $t_2$) can be expressed as a boolean combination of the $\tau(\theta_1)$ (resp. $\tau(\theta_2)$) where $a = \theta_1\times\theta_2 \in P\cup N$ for a certain $(P, N) \in t$. Therefore the $t_i$ are indeed in $\square$ if $\square$ is a socle that contains $t$. The only things left is to remove the atoms $t_1\times t_2$ such that $t_1 \simeq \mathbb{0}$ or $t_2 \simeq \mathbb{0}$. □

This decomposition of the subtypes of $\mathbb{1}\times\mathbb{1}$ in the form of a finite union of product types will often be used in the rest of this presentation. Several functions $\pi$ meet the requirements of the statement, and there is no need to specify the one we choose. In other words, we will only use the properties of the statement when we have to handle $\pi$. In particular, no property allows to *a priori* link $\pi(t)$ and $\pi(s)$, when $t \leq s$. However, there are choices of $\pi$ that make it possible to obtain such relations; we will describe informally how to obtain them. Let $t \leq \mathbb{1}\times\mathbb{1}$. A product type $t_1\times t_2 \leq t$ is said maximal if it is non-empty and if $t_1\times t_2 \leq t'_1\times t'_2 \leq t \Rightarrow t'_1\times t'_2 \leq t_1\times t_2$. The amount of such types, modulo equivalence, is finite, and if we choose a representative in each equivalence class, we obtain a decomposition $\pi$ that is uniquely defined (modulo equivalence). This decomposition verifies the following property, for types $t$ and $s$ such that $t \leq s$:

$$\forall (t_1, t_2) \in \pi(t).\exists (t'_1, t'_2) \in \pi(s).\ t_1 \leq t'_1 \wedge t_2 \leq t'_2$$

In other words, a smaller type has a more precise decomposition. In practice, this decomposition in maximal products can be computed from an arbitrary decomposition $\pi$, by "cutting" and "merging" its elements (via boolean combinations on each component).

The following result gives an decomposition analogous to $\pi$ for arrow types.

**Lemma 4.37** *Let* $t$ *be a type such that* $t \leq \mathbb{0}\to\mathbb{1}$*. Then we can compute a type* $Dom(t)$ *and a finite set of type pairs* $\rho(t)$ *such that:*

$$\forall t_1, t_2.\ (t \leq t_1\to t_2) \iff \begin{cases} t_1 \leq Dom(t) \\ \forall (s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq t_2) \end{cases}$$

*and such that if* $t$ *is in a socle* $\square$*, then* $Dom(t)$ *and the type of* $\rho(t)$ *are also in* $\square$*.*

> *Proof*: The property is true for $t \simeq \mathbb{0}$ by taking $\mathrm{Dom}(t) = \mathbb{1}$ and $\rho(t) = \emptyset$. If the property is true for two types $t$ and $t'$, then it is

true for their union $t \vee t'$ (by taking $\mathrm{Dom}(t \vee t') = \mathrm{Dom}(t) \wedge \mathrm{Dom}(t')$ and $\rho(t \vee t') = \rho(t) \cup \rho(t')$). Now we can write:

$$t \simeq \bigvee_{(P,N) \in t} \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a$$

and this brings us back to the case where $t$ is an intersection of arrow types and negations of arrow types, with, in addition, $t \not\simeq \mathbb{0}$. We can then easily conclude with Lemma 4.9. □

**Remark 4.38** *We easily find that Dom(t) is a smallest solution s (with regard to subtyping) to the inequation $t \leq s \to \mathbb{1}$. We deduce in particular that this function is contravariant: if $t \leq t' \leq \mathbb{0} \to \mathbb{1}$, then $Dom(t') \leq Dom(t)$.*

**Remark 4.39** *The finiteness of the set $\rho(t)$ expresses the fact that even if we have arbitrarily complex functions in the models, at the type level everything comes down to overloaded functions with a* finite *number of branches. This simplification comes from the fact that we only consider finite intersections.*

### 4.8.2 Building equivalent models

Consider a certain operator on types $F : \widehat{T} \to \widehat{T}$ for which we want to prove monotonicity ($t \leq s \Rightarrow F(t) \leq F(s)$), or other properties, such as $F(t_1 \vee t_2) \simeq F(t_1) \vee F(t_2)$. Since relation $\leq$ is not axiomatized and the "rules" that define it are particularly complex, the most natural approach consists in finding a set-theoretic counterpart to the $F$ operator. Indeed, if we can write $[\![F(t)]\!] = \widehat{F}([\![t]\!])$ for a certain operator $\widehat{F} : \mathcal{P}(D) \to \mathcal{P}(D)$, then it all comes down to proving the monotony of $\widehat{F}$ in relation to the set-theoretic inclusion, which is often easier. For example, if $\widehat{F}(X)$ is defined as $\{y \mid xRy\}$ for a certain binary relation $R$, then the property is trivial.

However, we know nothing about the nature of the elements of model $D$, and it is not easy to define such an operator $\widehat{F}$. We would like to have models that induce the same subtyping relation, but for which we could control the nature of their elements, at least superficially. We will see how to do it.

According to Corollary 4.15, if $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ is a model, then $\mathbb{E}[\![\_]\!] : \widehat{T} \to \mathcal{P}(\mathbb{E}D)$ is another, and it is equivalent to $[\![\_]\!]$. The following theorem gives a generalization of this result.

**Theorem 4.40** *Let $[\![\_]\!]_1 : \widehat{T} \to \mathcal{P}(D_1)$ and $[\![\_]\!]_2 : \widehat{T} \to \mathcal{P}(D_2)$ be two models equivalent to $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$. Let us define the unique set-theoretic interpretation $[\![\_]\!]' : \widehat{T} \to \mathcal{P}(D')$ with*

$$D' = \mathcal{C} + D_1 \times D_2 + \mathcal{P}(D \times D_\Omega)$$

*and:*

$$\begin{array}{lcll} [\![t_1 \times t_2]\!]' & = & [\![t_1]\!]_1 \times [\![t_2]\!]_2 & \\ [\![a]\!]' & = & \mathbb{E}[\![a]\!] & \text{for } a \in T_{\mathbf{basic}} \cup T_{\mathbf{fun}} \end{array}$$

*Then $[\![\_]\!]'$ is a model equivalent to $[\![\_]\!]$.*

The proof is easy; it simply consists in repeating the proof of Theorem 4.13. This construction allows us to reason "as if" we had $[\![\_]\!] = \mathbb{E}[\![\_]\!]$. We will show an example of that kind of reasoning, which will be useful to us later.

**Tying functional application** In the following chapter, we are going to introduce a $\lambda$-calculus with pairs. Functional application will in fact be encoded by an operator **app** that takes as arguments a pair $(f, x)$ and applies function $f$ to the argument $x$. To type this operator, we have to define a judgement $\mathbf{app} : t \to s$, that expresses, intuitively, the fact that if operator **app** is applied to a value of type $t$, then the result is necessarily a value of type $s$.

Let us start with a fairly natural syntactic approach. It consists in axiomatizing the relation $\mathbf{app} : \_ \to \_$ using the inductive system of Figure 4.1.

$$\frac{}{\mathbf{app} : ((t \to s) \times t) \to s}\ (\mathbf{app}\to)$$

$$\frac{\mathbf{app} : t_1 \to s_1 \quad \mathbf{app} : t_2 \to s_2}{\mathbf{app} : (t_1 \wedge t_2) \to (s_1 \wedge s_2)}\ (\mathbf{app}\wedge)$$

$$\frac{\mathbf{app} : t_1 \to s_1 \quad \mathbf{app} : t_2 \to s_2}{\mathbf{app} : (t_1 \vee t_2) \to (s_1 \vee s_2)}\ (\mathbf{app}\vee)$$

$$\frac{\mathbf{app} : t' \to s' \quad t \leq t' \quad s' \leq s}{\mathbf{app} : t \to s}\ (\mathbf{app} \leq)$$

Figure 4.1: Axiomatizing the typing of functional application

These rules are "safe", for an intuitive interpretation of arrow types in a strict language. The rule (**app**$\to$) corresponds to a classic typing rule in a $\lambda$-calculus. The others are generic: they are semantically correct for any operator. For example, we can justify the rule (**app**$\vee$) by saying that if the value to which we apply the operator is of type $t_1 \vee t_2$, then it is of type $t_i$ for a certain $i$. The corresponding premise then gives that the result is of type $s_i$, and therefore *a fortiori* of type $\leq s_1 \vee s_2$.

Nevertheless, it would be nice to have a semantic interpretation of the application to give a formal meaning to this safety assertion. It will also allow us to consider the question of the completeness of the system of rules. Furthermore, for a given type $t$, we will need to compute the smallest type $s$ such that $\mathbf{app} : t \to s$, and the axiomatic definition does not allow us to do that easily.

So we are going to propose a semantic definition of the binary relation $\mathbf{app} : \_ \to \_$. To do this, we will "mimic" functional application in a model. This cannot be done in model $[\![\_]\!]$ because we do not know anything about the nature of the elements in $D$. The model $\mathbb{E}[\![\_]\!]$ does not fit either: some elements of $\mathbb{E}D$ are pairs indeed, but their first component is just an element of $D$, when we would like to have an element of $\mathbb{E}D$, and more accurately, of $\mathbb{E}_{\mathbf{fun}}D$, to be able to define functional application the extensional way.

Theorem 4.40 allows us to build a model $[\![\_]\!]' : \widehat{T} \to \mathcal{P}(D')$, with:

$$D' = \mathcal{C} + (\mathbb{E}D) \times D + \mathcal{P}(D \times D_\Omega)$$

and

$$[\![t_1 \times t_2]\!]' = \mathbb{E}[\![t_1]\!] \times [\![t_2]\!]$$

This model is equivalent to the model $[\![\_]\!]$, therefore it is legitimate to work inside of it. We can mimic in $D'$ the expected behavior of functional application.

We define a binary relation $\triangleright$ between $D'$ and $D_\Omega$ by: $d \triangleright \Omega$ if $d$ is not in $\mathcal{P}(D \times D_\Omega) \times D$, and $(f,x) \triangleright y$ if $f \in \mathcal{P}(D \times D_\Omega)$ and $(x,y) \in f$. For any subset $X$ of $D'$, let us write $\mathbf{app}(X) = \{y \mid x \triangleright y, x \in X\} \subseteq D_\Omega$.

**Lemma 4.41** *Let $t_1$,$t_2$ and $s$ be three types. Then:*

$$\mathbf{app}(\llbracket t_1 \times t_2 \rrbracket') \subseteq \llbracket s \rrbracket \iff (t_1 \leq t_2 \to s) \vee (t_2 \simeq \mathbb{0})$$

*Proof*: Let us prove implication $\Rightarrow$. Let us assume $\mathbf{app}(\llbracket t_1 \times t_2 \rrbracket') \subseteq \llbracket s \rrbracket$ and $t_2 \not\simeq \mathbb{0}$, and prove that $\mathbb{E}\llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket \to \llbracket s \rrbracket$. Let $f \in \mathbb{E}\llbracket t_1 \rrbracket$. First of all, we see that $f \in \mathcal{P}(D \times D_\Omega)$. Indeed, for every element $x \in \llbracket t_2 \rrbracket$, we have $(f,x) \in \mathbb{E}\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket = \llbracket t_1 \times t_2 \rrbracket'$. If $f$ was not in $\mathcal{P}(D \times D_\Omega)$, we would have $(f,x) \triangleright \Omega$, and therefore $\Omega \in \mathbf{app}(\llbracket t_1 \times t_2 \rrbracket')$, which contradicts the hypothesis. Let us show now that $f \in \llbracket t_2 \rrbracket \to \llbracket s \rrbracket$. Let us take $x \in \llbracket t_2 \rrbracket$. We now need to prove that if $(x,y) \in f$, then $y \in \llbracket s \rrbracket$. But if $(x,y) \in f$, then $(f,x) \triangleright y$, hence $y \in \mathbf{app}(\llbracket t_1 \times t_2 \rrbracket') \subseteq \llbracket s \rrbracket$.

Let us prove implication $\Leftarrow$. The case $t_2 \simeq \mathbb{0}$ is trivial, because in that case $t_1 \times t_2 \simeq \mathbb{0}$, hence $\mathbf{app}(\llbracket t_1 \times t_2 \rrbracket') = \mathbf{app}(\emptyset) = \emptyset$. Suppose $t_1 \leq t_2 \to s$. Let us take an element in $\llbracket t_1 \times t_2 \rrbracket' = \mathbb{E}\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$. It is therefore a pair $(f,x)$, with $f \in \mathbb{E}\llbracket t_1 \rrbracket$ and $x \in \llbracket t_2 \rrbracket$. Since $t_1 \leq t_2 \to s$, we have $f \in \llbracket t_2 \rrbracket \to \llbracket s \rrbracket$. If $(f,x) \triangleright y$, we have $(x,y) \in f$, which indeed implies that $y \in \llbracket s \rrbracket$. □

**Theorem 4.42** *Let $t$ and $s$ be two types. The following assertions are equivalent:*

*(i)* $\mathbf{app} : t \to s$
*(ii)* $\mathbf{app}(\llbracket t \rrbracket') \subseteq \llbracket s \rrbracket$
*(iii)* $(t \leq \mathbb{1} \times \mathbb{1}) \wedge \forall (t_1,t_2) \in \pi(t).\ t_1 \leq t_2 \to s$

*Proof*: Implication *(i)* $\Rightarrow$ *(ii)* can be proven by induction on the derivation of judgement $\mathbf{app} : t \to s$. The rules (**app** $\leq$), (**app**$\vee$), (**app**$\wedge$) can be treated with elementary set-theoretic reasonings. The case of rule (**app**$\to$) is a consequence of the previous lemma.

Let us prove implication *(ii)* $\Rightarrow$ *(iii)*. Since $\Omega \notin \mathbf{app}(\llbracket t \rrbracket')$, we have $\llbracket t \rrbracket' \subseteq D \times D$, and therefore $t \leq \mathbb{1} \times \mathbb{1}$. For $(t_1,t_2) \in \pi(t)$, we have $t_1 \times t_2 \leq t$, and therefore $\mathbf{app}(\llbracket t_1 \times t_2 \rrbracket') \subseteq \llbracket s \rrbracket$, and the previous lemma gives indeed $t_1 \leq t_2 \to s$ (because $t_2 \not\simeq \mathbb{0}$).

Let us finally prove implication *(iii)* $\Rightarrow$ *(i)*. We have:

$$t \simeq \bigvee_{(t_1,t_2) \in \pi(t)} t_1 \times t_2$$

If $(t_1,t_2) \in \pi(t)$, we have $t_1 \times t_2 \leq (t_2 \to s) \times t_2$. Rule (**app**$\to$) gives $\mathbf{app} : (t_2 \to s) \times t_2 \to s$. By applying rule (**app** $\leq$), we obtain $\mathbf{app} : t_1 \times t_2 \to s$. Rules (**app**$\vee$) and (**app** $\leq$) finally give $\mathbf{app} : t \to s$. We have used a generalization of the rule (**app**$\vee$) for a finite arbitrary union. The case of a non-empty union is immediate. For an empty union, we notice that $\mathbf{app} : \mathbb{0} \to \mathbb{0}$ thanks to rule (**app**$\to$) (which gives $\mathbf{app} : (\mathbb{0} \to \mathbb{0}) \times \mathbb{0} \to \mathbb{0}$), then with rule (**app** $\leq$). □

This theorem provides a better understanding of the binary relation **app** : $\_\!\rightarrow\!\_$ between types. Property $(ii)$ gives the semantic point of view (extensional application). Property $(iii)$ directly gives an algorithm to compute the relation (that is, to decide if $\mathbf{app} : t\!\rightarrow\! s$, with $t$ and $s$ given). Even better: by using Lemma 4.37, we obtain an algorithm to compute a smallest type $s$ such that $\mathbf{app} : t\!\rightarrow\! s$, for a given $t$, when such a type exists.

**Remark 4.43** *We did not need rule* (**app**$\wedge$) *in the proof of* $(iii) \Rightarrow (i)$. *This proves that this rule is admissible in the system consisting of the three other rules* (**app**$\rightarrow$), (**app**$\vee$), (**app** $\leq$). *Direct proof of this fact, without going through semantics, is easy, albeit tedious.*

**Remark 4.44** *The proof* $(i) \Rightarrow (iii)$ *goes through the semantic point of view. The reader can convince himself that a direct proof (without introducing model* $[\![\_]\!]'$*) is difficult to carry out.*

**Corollary 4.45** *Let* $t_1$, $t_2$ *and* $s$ *be three types. Then:*

$$\mathbf{app} : t_1\!\times\! t_2\!\rightarrow\! s \iff (t_1 \leq t_2\!\rightarrow\! s) \vee (t_2 \simeq \mathbb{0})$$

**Lemma 4.46** *Let* $t_f = \bigwedge_{i\in I} t_i\!\rightarrow\! s_i$ *where* $I$ *is a non-empty set, and* $t_x$ *is a type such that* $t_x \leq \bigvee_{i\in I} t_i$. *Let us write:*

$$s = \bigvee_{I'\subseteq I \ \mid \ (*)} \ \bigwedge_{i\in I\setminus I'} s_i$$

*where* $(*)$ *is the condition* $t_x \not\leq \bigvee_{i\in I'} t_i$. *Then we have:*

$$\mathbf{app} : t_f\!\times\! t_x\!\rightarrow\! s$$

*Proof*: Corollary 4.10 allows us to break down the relation $t_f \leq t_x\!\rightarrow\! s$ into:

$$\forall I' \subseteq I.\ \left(t_x \leq \bigvee_{i\in I'} t_i\right) \vee \left\{ \begin{array}{l} I \neq I' \\ \displaystyle\bigwedge_{i\in I\setminus I'} s_i \leq s \end{array} \right.$$

and we find that this property is true, with the definition of $s$ given in the statement. We conclude with Corollary 4.45.

As an illustration, let us give a proof that uses the semantic interpretation of relation **app**. We need to establish that $\mathbf{app}([\![t_f\!\times\! t_x]\!]') \subseteq [\![s]\!]$. Let us take an element in $[\![t_f\!\times\! t_x]\!]'$. It is a pair $(f, x)$, with $f \in \mathbb{E}[\![t_f]\!]$ and $x \in [\![t_x]\!]$. Since $I \neq \emptyset$, we have $\mathbb{E}[\![t_f]\!] \subseteq \mathbb{E}_{\mathbf{fun}} D$, and therefore $f$ is a set of pairs. What is left is to show that if $y$ is such that $(x, y) \in f$, then $y \in [\![s]\!]$. For such a $y$, let us write $I' = \{i \mid x \notin [\![t_i]\!]\}$. Since $t_f \leq t_i\!\rightarrow\! s_i$, we obtain: $x \in [\![t_i]\!] \Rightarrow y \in [\![s_i]\!]$, and therefore $y \in [\![s']\!]$ where $s' = \bigwedge_{i\in I\setminus I'} s_i$. Now $t_x \not\leq \bigvee_{i\in I'} t_i$ (because $x$ is in the interpretation of the left-hand side, and not in the interpretation of the right-hand side), which gives $s' \leq s$. □

## 4.9 Equation systems ∨×

In this section, we will consider a *structural* model $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$. We are going to study some form of equation systems on types. This tool will allow us to define the typing algorithm for pattern matching (Theorem 6.12).

Let us consider equation systems of the form:

$$\begin{cases} \alpha_1 & = & \dots \\ \dots & & \\ \alpha_n & = & \dots \end{cases}$$

where the right-hand side components are themselves variables $\alpha_i$, types $t$, variable products $\alpha_i \times \alpha_j$, or a finite union of terms of that form. Thus, by noting $V$ for the set of the $\alpha_i$, we can see each right-hand side component as a finite set of elements of the set $V + V \times V + \widehat{T}$.

**Definition 4.47** *An* **equation system ∨×** *is a pair $(V, \phi)$ where $V$ is a finite set and $\phi$ is a function $V \to \mathcal{P}_f(V + V \times V + \widehat{T})$. We say that the system is* **well-founded** *if $\phi(\alpha) \subseteq V \times V + \widehat{T}$ for every $\alpha \in V$.*

To define a concept of solution for such a system, we will associate it with a set-theoretic transformation and say that a solution is a fixed point of that transformation.

**Definition 4.48** *To any system $(V, \phi)$, we associate the operator $\widehat{\phi} : \mathcal{P}(D)^V \to \mathcal{P}(D)^V$ defined by:*

$$\widehat{\phi}(\rho)(\alpha) = \bigcup_{\beta \in \phi(\alpha)} \rho(\beta) \cup \bigcup_{(\beta_1,\beta_2) \in \phi(\alpha)} \rho(\beta_1) \times \rho(\beta_2) \cup \bigcup_{t \in \phi(\alpha)} [\![t]\!]$$

*This operator is increasing for the order on $\mathcal{P}(D)^V$ defined by $\rho_1 \leq \rho_2 \iff \forall \alpha \in V.\ \rho_1(\alpha) \subseteq \rho_2(\alpha)$. Its smallest fixed point is written $lfp(\widehat{\phi})$.*

We will now see how to compute the smallest fixed point, whose existence is guaranteed by Kleene's theorem.

**Theorem 4.49** *Let $(V, \phi)$ be the equation system ∨×. Then we can build a family of types $(t_\alpha)_{\alpha \in V}$ such that:*

$$\forall \alpha \in V.\ lfp(\widehat{\phi})(\alpha) = [\![t_\alpha]\!]$$

*Moreover, this family does not depend on the (structural) model chosen.*

The theorem is a consequence of the two following lemmas.

**Lemma 4.50** *For any system $(V, \phi)$, we can compute a well-founded system $(V, \phi')$ such that $lfp(\widehat{\phi}) = lfp(\widehat{\phi'})$.*

> *Proof*: Let us write $\overset{*}{\rightsquigarrow}$ the transitive and reflexive closure of the relation defined by $\alpha \rightsquigarrow \beta \iff \beta \in \phi(\alpha)$. We define the well-founded system $(V, \phi')$ by:
>
> $$\phi'(\alpha) = \bigcup_{\alpha \overset{*}{\rightsquigarrow} \beta} \phi(\beta) \backslash V$$

Let us write $\rho_1 = \mathrm{lfp}(\widehat{\phi})$ and $\rho_2 = \mathrm{lfp}(\widehat{\phi'})$.

First of all, we find that for every $\rho$:

$$\widehat{\phi'}(\rho)(\alpha) \subseteq \bigcup_{\alpha \overset{*}{\rightsquigarrow} \beta} \widehat{\phi}(\rho)(\beta)$$

In addition, if $\alpha \rightsquigarrow \beta$, then $\rho(\beta) \subseteq \widehat{\phi}(\rho)(\alpha)$, and therefore $\rho_1(\beta) \subseteq \rho_1(\alpha)$, and it can be generalized to the case $\alpha \overset{*}{\rightsquigarrow} \beta$. We thus obtain $\widehat{\phi'}(\rho_1)(\alpha) \subseteq \rho_1(\alpha)$, which gives $\rho_2 \leq \rho_1$.

Now let us prove that $\rho_1 \leq \rho_2$. For any $\rho$, we have:

$$\widehat{\phi}(\rho)(\alpha) \subseteq \widehat{\phi'}(\rho)(\alpha) \cup \bigcup_{\alpha \rightsquigarrow \beta} \rho(\beta)$$

Hence in particular:

$$\widehat{\phi}(\rho_2)(\alpha) \subseteq \rho_2(\alpha) \cup \bigcup_{\alpha \rightsquigarrow \beta} \rho_2(\beta)$$

But if $\alpha \rightsquigarrow \beta$, then $\phi'(\beta) \subseteq \phi'(\alpha)$, and therefore $\rho_2(\beta) = \widehat{\phi'}(\rho_2)(\beta) \subseteq \widehat{\phi'}(\rho_2)(\alpha) = \rho_2(\alpha)$. Thus we obtain $\widehat{\phi}(\rho_2)(\alpha) \subseteq \rho_2(\alpha)$, which indeed gives $\rho_1 \leq \rho_2$. □

**Lemma 4.51** *Let $(V, \phi)$ be a well-founded system. Then the operator $\widehat{\phi}$ has a unique fixed point $\rho$, and we can compute a family of types $(t_\alpha)_{\alpha \in V}$, independent from the model, such that:*

$$\forall \alpha \in V.\ \rho(\alpha) = [\![t_\alpha]\!]$$

*Proof*: Starting with proving the unicity of the fixed point. Let $\rho_1$ and $\rho_2$ be two fixed points of $\widehat{\phi}$, and let us prove that $\rho_1 = \rho_2$. To do that, it is enough to establish the following property for every $d \in D$:

$$\forall \alpha \in V.\ d \in \rho_1(\alpha) \iff \rho_2(\alpha)$$

Proceeding by induction over $d$, by noticing that the assertion $d \in \rho_i(\alpha)$ can be written (since $\rho_i = \widehat{\phi}(\rho_i)$):

$$\begin{cases} \exists t \in \phi(\alpha).\ d \in [\![t]\!] \\ \vee \\ \exists (\beta_1, \beta_2) \in \phi(\alpha), d_1 \in \rho_i(\beta_1), d_2 \in \rho_i(\beta_2).\ d = (d_1, d_2) \end{cases}$$

Moving on to the second point of the statement. The regularity property of the $\mathcal{B} \circ F$-coalgebra $\mathbb{T}$ allows us to build a family of types $(t_\alpha)_{\alpha \in V}$ such that:

$$\forall \alpha \in V.\ t_\alpha = \bigvee_{(\beta_1, \beta_2) \in \phi(\alpha)} t_{\beta_1} \times t_{\beta_2} \vee \bigvee_{t \in \phi(\alpha)} t$$

We immediately see that if we fix $\rho(\alpha) = [\![t_\alpha]\!]$, then $\rho$ is a fixed point of $\widehat{\phi}$. □

# Chapter 5

# Calculus

In this chapter, we introduce a typed mini-language (or calculus). We use the type algebra introduced in Chapter 3.

## 5.1 Syntax

Expressions are defined by the grammar of Figure 5.1. Let $\mathcal{E}$ denote the set of expressions.

| | | | | |
|---|---|---|---|---|
| $e$ | ::= | $c$ | $c \in \mathcal{C}$ | constant |
| | \| | $(e_1, e_2)$ | | pair |
| | \| | $\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e$ | $t_i, s_i \in \widehat{T},\ n \geq 1$ | abstraction |
| | \| | $x$ | | variable |
| | \| | $o(e)$ | $o \in \mathcal{O}$ | operator |
| | \| | $(x = e \in t\ ?\ e_1 \mid e_2)$ | $t \in \widehat{T}$ | type test |

Figure 5.1: Expressions (terms) of the calculus

The letters $x$ and $f$ denote *variables*. We assume that there is an infinite number of distinct variables, and, classically, we consider expressions modulo alpha-conversion. Collision problems are avoided by implicit renaming. The set of free variables of an expression $e$ is written $\mathrm{fv}(e)$, and it is defined the following way:

$$\begin{array}{lcl} \mathrm{fv}(c) & = & \emptyset \\ \mathrm{fv}((e_1, e_2)) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\ \mathrm{fv}(\mu f(\ldots).\lambda x.e) & = & \mathrm{fv}(e) \backslash \{f, x\} \\ \mathrm{fv}(x) & = & x \\ \mathrm{fv}(o(e)) & = & \mathrm{fv}(e) \\ \mathrm{fv}((x = e \in t\ ?\ e_1 \mid e_2)) & = & \mathrm{fv}(e) \cup ((\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)) \backslash \{x\}) \end{array}$$

We say that an expression $e$ is closed if $\mathrm{fv}(e) = \emptyset$.

Productions for constants, pairs, and variables do not require any comments. The production for abstractions is a bit special: it allows on the one hand to define recursive functions (the variable $f$ is bound to the function itself in its body) and on the other hand to specify a finite number of types $(t_1 \to s_1; \ldots; t_n \to s_n)$ for the *prototype* (or interface) of these functions. The scope of variables $f$ and $x$ is limited to the body $e$ of the abstraction. If $P$ is a finite non-empty set of arrow types, let $P = \{t_1 \to s_1, \ldots, t_n \to s_n\}$, we allow ourselves to write $\mu f(P).\lambda x.e$ instead of $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$ (this requires us to chose an order of enumeration for the elements of $P$).

Let $\mathcal{O}$ denote a set of operators; it is a parameter of the calculus. We assume that there is always a "functional application" operator $\mathbf{app} \in \mathcal{O}$. We write $e_1 e_2$ instead of $\mathbf{app}((e_1, e_2))$.

The last production allows for a dynamic type check. If the result of expression $e$ is $t$, then the evaluation continues with expression $e_1$, if not with expression $e_2$. Variable $x$ is bound to the result of $e$ in $e_1$ and in $e_2$.

These type checks mean that the semantics of the language is directed by types. So we need to introduce a type system before we consider the semantics of the language.

## 5.2 Type system

The type system is presented in a classic way by an induction system (Figure 5.2). The typing judgement has the form $\Gamma \vdash e : t$ where $\Gamma$ denotes a typing environment, that is, a partial function with finite domain from variables to types.

We need a subtyping relation. In the previous chapter, we saw how to define one from an arbitrary model. Let us take a *well-founded* model $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$, and we will simply write $\leq$, instead of $\leq_{[\![\_]\!]}$, for the subtyping relation it induces. We assume that this subtyping relation is decidable (this is the case for universal models when subtyping is decidable for base types). We denote by $\simeq$ the equivalence induced by this subtyping: $t_1 \simeq t_2 \iff (t_1 \leq t_2) \wedge (t_2 \leq t_1)$

For each operator $o \in \mathcal{O}$, we assume given a binary relation between types, written $o : \_ \to \_$. We assume that if we have $o : t_1 \to s_1$ and $o : t_2 \to s_2$, then we also have $o : (t_1 \wedge t_2) \to (s_1 \wedge s_2)$. This condition will allow us to obtain Lemma 5.3, which in turn allows us to see that the set of types assigned to an expression is a filter (a set of types closed by subsumption and intersection). It is always possible to complete $o : \_ \to \_$ to satisfy this requirement. For the **app** operator , we define the binary relation $\mathbf{app} : \_ \to \_$ defined in Chapter 4.

Outside of the (*abstr*) and (*case*) rules, the system (Figure 5.2) is quite usual. Let us comment on these two rules.

The (*abstr*) rule gives a type to abstractions. In the calculus, functions are overloaded: their prototype can specify several arrow types. The type given to the abstraction corresponds to the intersection of all these arrow types. It is case $m = 0$. We check that each of these arrow types is valid: for this, we must check $n$ constraints on the type of the body, with different assumptions about the type of the argument.

But the rule also allows for the addition of an arbitrary finite number of arrow type negations ($m > 0$) to this intersection, as long as the intersection does not become empty. The reason for allowing these negative types is quite simple: we

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}\ (subsum)$$

$$\frac{}{\Gamma \vdash c : b_c}\ (const)$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

$$\frac{\begin{array}{l} t = \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j) \\ \forall i = 1..n. (f : t), (x : t_i), \Gamma \vdash e : s_i \\ t \not\simeq \mathbb{0} \end{array}}{\Gamma \vdash \mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e : t}\ (abstr)$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (var)$$

$$\frac{\Gamma \vdash e : t \quad o : t \to s}{\Gamma \vdash o(e) : s}\ (op)$$

$$\frac{\Gamma \vdash e : t_0 \quad \left\{ \begin{array}{l} t_1 = t_0 \wedge t \\ t_2 = t_0 \setminus t \end{array} \right. \quad \forall i = 1..2. \left\{ \begin{array}{ll} s_i = \mathbb{0} & \text{if } t_i \simeq \mathbb{0} \\ (x : t_i), \Gamma \vdash e_i : s_i & \text{if } t_i \not\simeq \mathbb{0} \end{array} \right.}{\Gamma \vdash (x = e \in t\ ?\ e_1 | e_2) : s_1 \vee s_2}\ (case)$$

Figure 5.2: Type system

want to give a model structure to all of the values of the language. This will be done by interpreting a type as the set of values of this type. Because a closed and well-typed abstraction is a value, it must have one of two types $t \to s$ or $\neg(t \to s)$ for any choice of $t$ and $s$; indeed, in a model, every element must be either in the interpretation of a type, or in the interpretation of its complement. This explains why arrow type negations can be added to the typing rule, otherwise arrow type negations could never be inferred for abstractions. The following lemma allows us to write the condition $t \not\simeq \mathbb{0}$ differently.

**Lemma 5.1** *Let* $t = \bigwedge_{i=1..n}(t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j)$ *be a type. Then*

$$t \simeq \mathbb{0} \iff \exists j = 1..m.\ \bigwedge_{i=1..n} (t_i \to s_i) \leq t'_j \to s'_j$$

| *Proof*: Consequence of Lemma 4.9. □

In other words, the condition $t \not\simeq \mathbb{0}$ simply means that none of the added arrow types (in negative) could be obtained (in positive) thanks to the subsumption rule from the type $\bigwedge_{i=1..n}(t_i \to s_i)$.

The rule (*case*) is simpler to write. The idea is to refine type $t_0$. If the first branch is selected, then it means that expression $e$ was evaluated as a value of type $t$; therefore we know that $x$ will have type $t$ and type $t_0$, and therefore type $t \wedge t_0$. Similarly, if the second branch is selected, the value is not of type $t$, and therefore is of type $\neg t$ because the set of values must be a model. Both cases must be considered to compute the type of the result. In general, it is the union of the result types of each branch. But we can refine this: indeed, if we are sure that a branch will not be used, it is not necessary to take it into account in this union. This refinement is typically necessary when one types the body of an overloaded function (several type arrows in its prototype). It is then necessary to check several times the type of the body, with different assumptions, and in some cases, it may be necessary not to take into account a certain branch in order to be able to verify the constraint. We can see that in the following example:

$$\mu f(\texttt{int}\rightarrow\texttt{int};\texttt{bool}\rightarrow\texttt{bool}).\lambda x.(y = x \in \texttt{int}\ ?\ 1 \mid \texttt{true})$$

In this example, we are using two disjoint base types `int` and `bool`, and two constants 1 (of type `int`) and `true` (of type `bool`). When verifying the constraint given by $\texttt{int}\rightarrow\texttt{int}$, we do have to ignore the second branch, otherwise the type of the body would be $\texttt{int}\vee\texttt{bool}$, which is not a subtype of `int`.

## 5.3 Syntactic properties of the typing

**Lemma 5.2 (Strengthening assumptions)** *Let $\Gamma_1$ and $\Gamma_2$ be two typing environments such that for all $x$ in the domain of $\Gamma_1$, we have $\Gamma_2(x) \leq \Gamma_1(x)$. If $\Gamma_1 \vdash e : t$, then $\Gamma_2 \vdash e : t$.*

*Proof*: By induction on the derivation of $\Gamma_1 \vdash e : t$. □

**Lemma 5.3** *If $\Gamma \vdash e : t_1$ and $\Gamma \vdash e : t_2$, then $\Gamma \vdash e : t_1 \wedge t_2$.*

*Proof*: By induction over the structure of the two typing derivations for $\Gamma \vdash e : t_1$ and $\Gamma \vdash e : t_2$.

Considering first the case where the last rule applied in one of those two derivations is (*subsum*), let us write:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash e : s_1} \quad s_1 \leq t_1}{\Gamma \vdash e : t_1} \quad \dfrac{\ldots}{\Gamma \vdash e : t_2}$$

The induction hypothesis allows us to obtain $\Gamma \vdash e : s_1 \wedge t_2$. We conclude this case by noticing that $s_1 \wedge t_2 \leq t_1 \wedge t_2$ since $s_1 \leq t_1$.

If none of the two last applied rules is (*subsum*), then those are necessarily two instances of the same rule.

Rules (*const*), (*var*): these rules only allow for a single type $t$ for the expression, and $t \wedge t \simeq t$.

Rule (*op*): a direct consequence of the closure by intersection property of $o : \_\rightarrow\_$.

Rule (*pair*): considering the following situation.

$$\frac{\dfrac{\cdots}{\Gamma \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Gamma \vdash e_2 : t_2}}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad \frac{\dfrac{\cdots}{\Gamma \vdash e_1 : t_1'} \quad \dfrac{\cdots}{\Gamma \vdash e_2 : t_2'}}{\Gamma \vdash (e_1, e_2) : t_1' \times t_2'}$$

Let $t_1'' = t_1 \wedge t_1'$ and $t_2'' = t_2 \wedge t_2'$. By applying twice the induction hypothesis, we obtain $\Gamma \vdash e_1 : t_1''$ and $\Gamma \vdash e_2 : t_2''$. Rule (*pair*) then gives $\Gamma \vdash (e_1, e_2) : t_1'' \times t_2''$. To conclude, it suffices to notice that $t_1'' \times t_2'' \simeq (t_1 \times t_2) \wedge (t_1' \times t_2')$ according to Lemma 4.34.
Rule (*case*): consider the following situation.

$$\frac{\dfrac{\cdots}{\Gamma \vdash e : t_0} \quad \dfrac{\cdots}{(x : t_i), \Gamma \vdash e_i : s_i}}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s_1 \vee s_2} \qquad \frac{\dfrac{\cdots}{\Gamma \vdash e : t_0'} \quad \dfrac{\cdots}{(x : t_i'), \Gamma \vdash e_i : s_i'}}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s_1' \vee s_2'}$$

with $t_1 = t_0 \wedge t$, $t_2 = t_0 \setminus t$, $t_1' = t_0' \wedge t$, $t_2' = t_0' \setminus t$. First of all, the induction hypothesis gives: $\Gamma \vdash e : t_0''$ with $t_0'' = t_0 \wedge t_0'$. We then fix $t_1'' = t_0'' \wedge t$ and $t_2'' = t_0'' \setminus t$. Let $i = 1..2$. We find that $t_i'' \leq t_i$, hence Lemma 5.2 gives $(x : t_i''), \Gamma \vdash e_i : s_i$. Similarly, we obtain $(x : t_i''), \Gamma \vdash e_i : s_i'$, hence by applying the induction hypothesis $(x : t_i''), \Gamma \vdash e_i : s_i''$ where $s_i'' = s_i \wedge s_i'$. With rule (*case*), we then prove that $\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s_1'' \vee s_2''$, and we conclude by noticing that $s_1'' \vee s_2'' \leq (s_1 \vee s_2) \wedge (s_1' \vee s_2')$.

Special cases (when $t_i \simeq \mathbb{0}$ or $t_i' \simeq \mathbb{0}$) do not pose any problem.
Rule (*abstr*): consider two applications of rule (*abstr*) to the same abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$, with the following types:

$$\begin{aligned} t &= \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t_j' \to s_j') \\ t' &= \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=m+1..m'} \neg(t_j' \to s_j') \end{aligned}$$

where $t \not\simeq \mathbb{0}$ and $t' \not\simeq \mathbb{0}$. Writing:

$$t'' = \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m'} \neg(t_j' \to s_j')$$

We have $t'' \simeq t \wedge t'$. Verifying that rule (*abstr*) allows the type $t''$ to be given to the abstraction under consideration. For $i = 1..n$, by hypothesis we have $(f : t), (x : t_i), \Gamma \vdash e : s_i$, therefore, by Lemma 5.2: $(f : t''), (x : t_i), \Gamma \vdash e : s_i$. All that remains is to verify that $t'' \not\simeq \mathbb{0}$, which immediately comes from Lemma 5.1. Note that in that case, we have not used the induction hypothesis.
□

**Corollary 5.4** *Let $\Gamma$ be an environment and $e$ be a well-typed expression under $\Gamma$. Then the set $\{t \in \widehat{T} \mid (\Gamma \vdash e : t) \vee (\Gamma \vdash e : \neg t)\}$ contains $\mathbb{0}$ and is stable by boolean operators.*

*Proof*: Let $E$ be the set introduced in the statement. It is clearly stable by operator $\neg$ and invariant by equivalence $\simeq$. We have $\Gamma \vdash e : \mathbb{1}$ by rule (*subsum*), hence $\mathbb{1} \in E$. We obtain $\mathbb{0} \in E$ by writing $\mathbb{0} \simeq \neg\mathbb{1}$. All that remains is the prove that $E$ is stable by $\vee$, because then it will also be stable by $\wedge$ in view of $t_1 \wedge t_2 \simeq \neg((\neg t_1) \vee (\neg t_2))$. Taking $t_1$ and $t_2$ in $E$, and showing that $t_1 \vee t_2$ is also in $E$. Assume that $\Gamma \not\vdash e : t_1 \vee t_2$. Then, by rule (*subsum*), we obtain $\Gamma \not\vdash e : t_1$ and $\Gamma \not\vdash e : t_2$. Therefore we have $\Gamma \vdash e : \neg t_1$ and $\Gamma \vdash e : \neg t_2$. Lemma 5.3 gives $\Gamma \vdash e : \neg t_1 \wedge \neg t_2$. Now $\neg t_1 \wedge \neg t_2 \simeq \neg(t_1 \vee t_2)$. □

**Lemma 5.5 (Substitution)** *Let $e, e_1, \ldots, e_n$ be expressions, $x_1, \ldots, x_n$ distinct variables, $t, t_1, \ldots, t_n$ types, and $\Gamma$ a typing environment. We write $e[x_1 := e_1; \ldots; x_n := e_n]$ the expression obtained from $e$ by replacing (without capture) the free occurrences of $x_i$ by $e_i$. We then have:*

$$\begin{cases} (x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t \\ \forall i = 1..n.\ \Gamma \vdash e_i : t_i \end{cases} \Rightarrow \Gamma \vdash e[x_1 := e_1; \ldots; x_n := e_n] : t$$

*Proof*: By induction on the typing derivation $(x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t$. □

## 5.4 Values

Among all the expressions, we isolate those that are *closed*, *well-typed*, and that are produced by the grammar:

$$\begin{array}{rcl} v & ::= & c \\ & | & (v_1, v_2) \\ & | & \mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e \end{array}$$

When talking about induction on the structure of a value, we consider the case of abstractions as a base case.

Writing $\mathcal{V}$ the set of values, and writing

$$[\![t]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash v : t\}$$

According to the subsumption rule, this definition is compatible with the semantic of types: if $t_1 \leq t_2$, then $[\![t_1]\!]_{\mathcal{V}} \subseteq [\![t_2]\!]_{\mathcal{V}}$, and in particular, if $t_1 \simeq t_2$, then $[\![t_1]\!]_{\mathcal{V}} = [\![t_2]\!]_{\mathcal{V}}$.

**Theorem 5.6** *Function $[\![\_]\!]_{\mathcal{V}} : \widehat{T} \to \mathcal{P}(\mathcal{V})$ is a structural model. The subtyping relation it induces is $\leq$.*

This theorem states that whatever the well-founded model $[\![\_]\!]$ we have chosen to define the subtyping relation $\leq$ used in the type system, we obtain a new interpretation of the subtyping relation $t \leq s$ as the inclusion of the set of values of type $t$ in the set of values of type $s$. In other words, we can "forget" the original model for interpreting the subtyping relation.

Different choices for the original model can give rise to different subtyping relations, and therefore to different type systems for the same calculus. It is interesting to note that, since the semantics of the calculus is driven by types, it is also affected by those choices.

**Corollary 5.7** *For any well-founded model, we can build a structural model that induces the same subtyping relation.*

The following results help establish Theorem 5.6.

**Lemma 5.8** *No value has type* $\mathbb{0}$.

> *Proof*: We show by an easy induction of the typing derivation $\vdash v : t$ that $t$ cannot be $\mathbb{0}$. □

**Lemma 5.9** *For every type* $t \in \widehat{T}$, $[\![t]\!]_{\mathcal{V}} \cup [\![\neg t]\!]_{\mathcal{V}} = \mathcal{V}$.

> *Proof*: We need to show that for every value $v$ and every type $t$, we have $\vdash v : t$ or $\vdash v : \neg t$. By induction on the structure of $v$. Corollary 5.4 brings us back to the case where $t$ is an atom $a$.
>
> If $v$ is a constant $c$, then $\vdash c : b_c$. If $a$ is not a base type, then $b_c \leq \neg a$ and therefore $\vdash c : \neg a$. If $a$ is a base type, then we have either $b_c \leq a$, or $b_c \leq \neg a$ (because $b_c$ is a singleton type, see Section 4.1), which concludes this first case.
>
> Consider the case where $v$ is an abstraction with prototype $(t_1 \to s_1; \ldots; t_n \to s_n)$. Writing $t_0 = \bigwedge_{i=1..n} t_i \to s_i$. We have $\vdash v : t_0$. If $a$ is not a type arrow, then $t_0 \leq \neg a$, therefore $\vdash v : \neg a$. If $a$ is a type arrow such that $t_0 \leq a$, then $\vdash v : a$ by rule (*subsum*). Otherwise, we can apply rule (*abstr*) with $a$ as negative arrow type, and we obtain $\vdash v : t_0 \wedge \neg a$, and therefore $\vdash v : \neg a$.
>
> Finally, consider the case $v = (v_1, v_2)$. If $a$ is not a product type, we easily get $\vdash v : \neg a$. Now assume $a = t_1 \times t_2$. If $\vdash v_1 : t_1$ and $\vdash v_2 : t_2$, then rule (*pair*) immediately gives $\vdash v : a$. If that is not the case, say $\not\vdash v_1 : t_1$, then the induction hypothesis gives $\vdash v_1 : \neg t_1$. Writing $t'_1 = \neg t_1$. We have $\vdash v_1 : t'_1$, and also $\vdash v_2 : \mathbb{1}$ because $v_2$ is well-typed. We obtain $\vdash v : t'_1 \times \mathbb{1} \leq \neg(t_1 \times t_2)$. □

**Lemma 5.10** *Function* $[\![\_]\!]_{\mathcal{V}} : \widehat{T} \to \mathcal{P}(\mathcal{V})$ *is a set-theoretic interpretation.*

> *Proof*: The condition $[\![\mathbb{0}]\!]_{\mathcal{V}} = \mathbb{0}$ is expressed by Lemma 5.8. We obtain $[\![\mathbb{1}]\!]_{\mathcal{V}} = \mathcal{V}$ by noticing that by definition, every value is well-typed, and can therefore be given the type $\mathbb{1}$ with rule (*subsum*).
>
> Condition $[\![t_1 \wedge t_2]\!]_{\mathcal{V}} = [\![t_1]\!]_{\mathcal{V}} \cap [\![t_2]\!]_{\mathcal{V}}$ is obtained using Lemma 5.3. In particular, it gives for every type $t$: $[\![t]\!]_{\mathcal{V}} \cap [\![\neg t]\!]_{\mathcal{V}} = [\![t \wedge \neg t]\!]_{\mathcal{V}} = [\![\mathbb{0}]\!]_{\mathcal{V}} = \emptyset$. By combining this with Lemma 5.9, we obtain $[\![\neg t]\!]_{\mathcal{V}} = \mathcal{V} \backslash [\![t]\!]_{\mathcal{V}}$.
>
> Finally, the union condition $[\![t_1 \vee t_2]\!]_{\mathcal{V}} = [\![t_1]\!]_{\mathcal{V}} \cup [\![t_2]\!]_{\mathcal{V}}$ can be obtained by writing $t_1 \vee t_2 \simeq \neg(\neg t_1 \wedge \neg t_2)$ and by using the properties established for negation and intersection. □

According to Corollary 4.15, to conclude that $[\![\_]\!]_{\mathcal{V}}$ is a model that defines the same subtyping relation as $[\![\_]\!]$, we now only have to establish the following lemma.

**Lemma 5.11** *For any type* $t$*:*

$$[\![t]\!] = \emptyset \iff [\![t]\!]_{\mathcal{V}} = \emptyset$$

*Proof*: The implication $\Rightarrow$ comes from Lemma 5.8. To establish implication $\Leftarrow$, we will use the fact that $[\![\_]\!]$ is a well-founded model (Definition 4.3). We have a well-founded order $\lhd$ over $D$. Let us reason by induction on this order to prove that for every type $t$, if $d \in [\![t]\!]$, then there exists $v \in [\![t]\!]_{\mathcal{V}}$. Assuming $d \in [\![t]\!]$. The definition of a well-founded model gives a $d' \in \mathbb{E}[\![t]\!]$ such that, if $d' = (d_1, d_2)$, then $d_1 \lhd d$ and $d_2 \lhd d$. Let us distinguish according to the form of $d'$.

If $d' = (d_1, d_2)$, then $d_1 \lhd d$ and $d_2 \lhd d$. This element $d'$ is in the set $\mathbb{E}[\![t]\!]$, and therefore in:

$$\bigcup_{(P,N)\in t \ \mid \ P\subseteq T_{\mathbf{prod}}} \left( \bigcap_{t_1\times t_2\in P} [\![t_1]\!] \times [\![t_2]\!] \right) \setminus \left( \bigcup_{t_1\times t_2\in N} [\![t_1]\!] \times [\![t_2]\!] \right)$$

By using Lemma 4.6, we find $(P, N) \in t$, and $N' \subseteq N$ such that $P \subseteq T_{\mathbf{prod}}$, $d_1 \in [\![s_1]\!]$ and $d_2 \in [\![s_2]\!]$ where:

$$\begin{cases} s_1 &= \bigwedge_{t_1\times t_2\in P} t_1 \setminus \bigvee_{t_1\times t_2\in N'} t_1 \\ s_2 &= \bigwedge_{t_1\times t_2\in P} t_2 \setminus \bigvee_{t_1\times t_2\in N\setminus N'} t_2 \end{cases}$$

By applying the induction hypothesis to $d_1$ and $d_2$, we find two values $v_1$ and $v_2$ such that $\vdash v_1 : s_1$ and $\vdash v_2 : s_2$. Taking $v = (v_1, v_2)$. Rule (*pair*) gives $\vdash v : s_1 \times s_2$. All that remains is to see that $s_1 \times s_2 \leq t$, which again comes from Lemma 4.6. We do have $v \in [\![t]\!]_{\mathcal{V}}$.

If $d'$ is a constant $c$, then we find $(P, N) \in t$ such that $P \subseteq T_{\mathbf{basic}}$, $\forall b \in P.\ c \in \mathbb{B}[\![b]\!]$ and $\forall b \in N.\ c \notin \mathbb{B}[\![b]\!]$. We deduce that $b_c \leq \bigwedge_{b\in P} b \setminus \bigvee_{b\in N} b \leq t$, which gives $\vdash c : t$.

Finally, consider the case where $d' \in \mathcal{P}(D \times D_\Omega)$. We find $(P, N) \in t$ such that $P \subseteq T_{\mathbf{fun}}$ and $d' \in \mathbb{E}[\![t']\!]$ where

$$t' = \bigwedge_{t_1\to t_2\in P} t_1 \to t_2 \wedge \bigwedge_{t_1\to t_2\in N} \neg(t_1 \to t_2)$$

We have $t' \leq t$ and $t' \not\simeq \mathbb{0}$ (since $\mathbb{E}[\![t']\!] \neq \emptyset$). We conclude by considering a closed well-typed abstraction, whose prototype is given by $P$, for example $\mu f(P).\lambda x.\ f\ x$. Rule (*abstr*) gives the type $t'$ to this value. □

To establish the safety of our semantics, we will need the result below.

**Lemma 5.12 (Inversion)**

$$\begin{array}{lcl} [\![t_1\times t_2]\!]_{\mathcal{V}} &=& \{(v_1, v_2) \mid \vdash v_1 : t_1, \vdash v_2 : t_2\} \\ [\![b]\!]_{\mathcal{V}} &=& \{c \mid b_c \leq b\} \\ [\![t\to s]\!]_{\mathcal{V}} &=& \{(\mu f(t_1\to s_1; \ldots; t_n\to s_n).\lambda x.e) \in \mathcal{V}. \mid \bigwedge_{i=1..n} t_i\to s_i \leq t\to s\} \end{array}$$

*Proof*: For each of those equalities, inclusion $\supseteq$ is immediate. Showing the three inclusions $\subseteq$ successively. Let us start by noticing that if $v$ is a value, then we have $\vdash v : a$ for a certain atom $a$ of the same kind of $v$: if $v$ is a constant (resp. pair)(resp. function), then $a$ is a base type (resp. product type)(resp. arrow type). Furthermore if we have $\vdash v : t_0$, then by Lemma 5.3 we obtain $\vdash v : t_0 \wedge a$. According to Lemma 5.8, we therefore have $t_0 \wedge a \not\simeq \mathbb{0}$, thus $t_0$ is also an atom, of the same kind of $v$ (because two atoms of a different kind have an empty intersection).

Starting with case $t_0 = t_1 \times t_2$. If we have $\vdash v : t_1 \times t_2$, then according to the above remark, $v$ must be a pair: $v = (v_1, v_2)$. The judgement $\vdash v : t_1 \times t_2$ can only be obtained by applying rule (*subsum*) a certain number of times after an instance of rule (*pair*). This gives two types $t'_1$ and $t'_2$ such that $\vdash v_1 : t'_1$, $\vdash v_2 : t'_2$, and $t'_1 \times t'_2 \leq t_1 \times t_2$. Now $t'_1 \not\leq \mathbb{0}$ and $t'_2 \not\leq \mathbb{0}$ according to Lemma 5.8, hence we obtain $t'_1 \leq t_1$ and $t'_2 \leq t_2$. We do have: $v \in \{(v_1, v_2) \mid \vdash v_1 : t_1, \vdash v_2 : t_2\}$.

Consider now the case $t_0 = b$. Here, the value $v$ is necessarily a constant $c$, and any typing derivation for $c$ consists in a finite application of rule (*subsum*) after rule (*const*). We therefore obtain $b_c \leq b$.

Finally, looking at case $t_0 = t \to s$. The value $v$ is an abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$. We see, with a reasoning as the one above, that there exists a finite number of arrow types $(t'_j \to s'_j)_{j=1..m}$ such that $s_0 \leq t \to s$ and $s_0 \not\simeq \mathbb{0}$ where:

$$s_0 = \bigwedge_{i=1..n} t_i \to s_i \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j)$$

Hence Lemma 5.1 gives:

$$\bigwedge_{i=1..n} t_i \to s_i \leq t \to s$$

as expected. □

## 5.5 Semantics

We can now define the semantics of the calculus. We will in fact define a small-step operational semantics, that is, a rewriting relation on *closed* expressions.

We materialize the type error with symbol $\Omega$, and we write $\mathcal{E}_\Omega = \mathcal{E} + \{\Omega\}$.

For any operator $o \in \mathcal{O}$, we assume given a binary relation, written $v \overset{o}{\rightsquigarrow} e$, where $v \in \mathcal{V}$ and $e \in \mathcal{E}_\Omega$. For the **app** operator , we take the $\beta$-reduction $(v_1, v_2) \overset{\mathbf{app}}{\rightsquigarrow} e[f := v_1; x := v_2]$ when $v_1 = \mu f(\ldots).\lambda x.e$ and $v \overset{\mathbf{app}}{\rightsquigarrow} \Omega$ when $v$ is not of the form $(\mu f(\ldots).\lambda x.e, v_2)$.

The (immediate) evaluation contexts are defined by the productions:

$$\begin{array}{rcl} C[] & ::= & ([], e) \\ & | & (e, []) \\ & | & o([]) \\ & | & (x = [] \in t \ ? \ e_1 | e_2) \end{array}$$

We define the rewriting relation $e \rightsquigarrow e'$ (where $e \in \mathcal{E}$ and $e' \in \mathcal{E}_\Omega$) by the following inductive system.

$$\frac{e \rightsquigarrow \Omega}{C[e] \rightsquigarrow \Omega} \quad \frac{e \rightsquigarrow e' \quad e' \neq \Omega}{C[e] \rightsquigarrow C[e']}$$

$$\frac{v \overset{o}{\rightsquigarrow} e}{o(v) \rightsquigarrow e}$$

$$\frac{i \in \{1,2\} \quad (i = 1 \iff \vdash v : t)}{(x = v \in t \ ? \ e_1 | e_2) \rightsquigarrow e_i[x := v]}$$

Note that with this semantics, a value is not reduced.

## 5.6 Type safety

**Definition 5.13** *Let $o$ be an operator, and $t$,$s$ two types. We write $(o : t \Longrightarrow s)$ if:*

$$\forall v \in \mathcal{V}, \forall e \in \mathcal{E}_\Omega.\ (\vdash v : t) \wedge (v \overset{o}{\rightsquigarrow} e) \quad \Rightarrow \quad \vdash e : s$$

**Definition 5.14** *We say that an operator $o$ is* **well-typed** *if:*

$$\forall t, s.\ (o : t \rightarrow s) \Rightarrow (o : t \Longrightarrow s)$$

**Definition 5.15** *Let $o$ be a well-typed operator . We say that the typing of $o$ is* **exact** *if, for any type $t$ such that $(o : t \Longrightarrow \mathbb{1})$, there exists $s$ such that $(o : t \rightarrow s)$ and:*

$$\forall v' \in [\![s]\!]_{\mathcal{V}}. \exists v \in [\![t]\!]_{\mathcal{V}}. \exists e \in \mathcal{E}.\ (v \overset{o}{\rightsquigarrow} e) \wedge (e \overset{*}{\rightsquigarrow} v')$$

**Remark 5.16** *This exactness condition kind of serves as a converse to the implication of Definition 5.14. It also asserts a very strong property, namely the existence of a type that captures exactly all the results that can be obtained by applying the operator on an arbitrary value of a given type. The typing of an exact operator does not introduce any approximation regarding its semantics and regarding the approximation already made on the type of the argument.*

**Lemma 5.17** *Let $o$ be an operator and $t_1, t_2, s_1, s_2$ be four types. Then:*
- $(o : t_1 \Longrightarrow s_1) \wedge (o : t_2 \Longrightarrow s_2) \Rightarrow (o : (t_1 \wedge t_2) \Longrightarrow (s_1 \wedge s_2))$,
- $(o : t_1 \Longrightarrow s_1) \wedge (o : t_2 \Longrightarrow s_2) \Rightarrow (o : (t_1 \vee t_2) \Longrightarrow (s_1 \vee s_2))$,
- $(o : t_1 \Longrightarrow s_1) \wedge (t_2 \leq t_1) \wedge (s_1 \leq s_2) \Rightarrow (o : t_2 \Longrightarrow s_2)$,

> *Proof*: Assume that $(o : t_1 \Longrightarrow s_1)$ and $(o : t_2 \Longrightarrow s_2)$. Proving first of all that $(o : (t_1 \wedge t_2) \Longrightarrow (s_1 \wedge s_2))$. Consider a value $v$ of type $t_1 \wedge t_2$ and a reduction $v \overset{o}{\rightsquigarrow} e$. The value $v$ is also of type $t_1$,

hence by hypothesis $\vdash e : s_1$. Similarly, $\vdash e : s_2$, and we deduce $\vdash e : s_1 \wedge s_2$ by using Lemma 5.3.

Proving now that $(o : (t_1 \vee t_2) \Longrightarrow (s_1 \vee s_2))$. Consider a value $v$ of type $t_1 \vee t_2$ and a reduction $v \overset{o}{\rightsquigarrow} e$. The value $v$ is of type $t_1$ or of type $t_2$. For example, if $v$ is of type $t_1$, by hypothesis we have $\vdash e : s_1$ and therefore also $\vdash e : s_1 \vee s_2$.

The last point is trivially proven by using the subsumption rule. □

**Lemma 5.18** *The* **app** *operator is well-typed.*

*Proof*: In view of the lemma above and the axiomatic definition of relation **app** : _→_ (Figure 4.1), we just need to check that (**app** : $((t \to s) \times t) \Longrightarrow s$) for arbitrary types $t$ and $s$. Let $v$ be a value of type $(t \to s) \times t$. According to Lemma 5.12, we see that it is a pair $v = (v_f, v_x)$ with $\vdash v_x : t$ and $\vdash v_f : t \to s$. The value $v_f$ is therefore an abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$, with $\bigwedge_{i=1..n} t_i \to s_i \leq t \to s$.

If $v \overset{\mathbf{app}}{\rightsquigarrow} e'$, we then have $e' = e[f := v_f; x := v_x]$. Now proving that $\vdash e' : s$.

Let $I = \{1, \ldots, n\}$ and $I' = \{i \in I \mid \vdash v_x : t_i\}$. We have $t \not\leq \bigvee_{i \in I \setminus I'} t_i$. Indeed, in the opposite case, we would have $v_x \in [\![\bigvee_{i \in I \setminus I'} t_i]\!]$, hence there would be an $i \notin I'$ with $\vdash v_x : t_i$, which would contradict the definition of $I'$.

Since $\bigwedge_{i=1..n} t_i \to s_i \leq t \to s$ and $t \not\leq \bigvee_{i \in I \setminus I'} t_i$, we obtain with Lemma 4.9 that $I' \neq \emptyset$ and:

$$\bigwedge_{i \in I'} s_i \leq s$$

To establish $\vdash e' : s$, it is enough, in view of Lemma 5.3, to prove that $\vdash e' : s_i$ for every $i \in I'$.

Let us take $i$ such that $\vdash v_x : t_i$. If the value $v_f$ is well-typed; then by considering an instance of rule (*abstr*) that gives a type $t'$ to $v_f$, we see that $(f : t'), (x : t_i) \vdash e : s_i$. Since $\vdash v_f : t'$ and $\vdash v_x : t_i$, Lemma 5.5 indeed gives $\vdash e' : s_i$. □

We can now establish a classic result of type preservation through reduction (*subject reduction*). Since we have made the error $\Omega$ explicit in the semantics of the calculus, this theorem ensures the absence of any type error when evaluating a well-typed expression.

**Theorem 5.19 (Subject reduction)** *If all the operators are well-typed, then typing is preserved through reduction: if $e \rightsquigarrow e'$ and $\vdash e : t$, then $\vdash e' : t$. In particular, $e' \neq \Omega$.*

*Proof*: By induction on the judgement derivation $\vdash e : t$. If the last rule applied is (*subsum*), we in fact have $\vdash e : s$ with $s \leq t$. By induction, we see that $\vdash e' : s$ and rule (*subsum*) then gives $\vdash e' : t$.

We now assume that the derivation of $\vdash e : t$ does not end by an application of rule (*subsum*). The syntactic form of $e$ then gives the last rule applied. Constant, variable and abstraction cases are impossible.

The rules of reduction under a context are trivial to handle. The case of the reduction of an operator comes from the definition of a well-typed operator.

Consider the of case of an expression $e = (x = v \in t\ ?\ e_1|e_2)$. The typing rule used is:

$$\frac{\Gamma \vdash v : t_0 \quad \begin{cases} t_1 = t_0 \wedge t \\ t_2 = t_0 \setminus t \end{cases} \quad \begin{cases} s_i = \mathbb{0} & \text{if } t_i \simeq \mathbb{0} \\ (x : t_i), \Gamma \vdash e_i : s_i & \text{if } t_i \not\simeq \mathbb{0} \end{cases}}{\Gamma \vdash (x = v \in t\ ?\ e_1|e_2) : s_1 \vee s_2}$$

There are two (mutually exclusive) cases: $\vdash v : t$ or $\vdash v : \neg t$. Consider for example the first case. The reduction is then:

$$e \rightsquigarrow e_1[x := v]$$

We have $\vdash v : t_0 \wedge t = t_1$, which already gives $t_1 \not\simeq \mathbb{0}$ and then $(x : t_1), \Gamma \vdash e_1 : s_1$. By applying Lemma 5.5 to this judgement, we obtain $\vdash e_1[x := v] : s_1 \leq s$. Case $\vdash v : \neg t$ is handled the same way. □

The theorem below gives an additional justification to our approach: even if we have defined the typing of functional application **app** in a purely set-theoretic way, it corresponds very exactly to the semantics of the calculus.

**Theorem 5.20** *The typing of the* **app** *operator is exact.*

*Proof*: Let $t$ be a type such that $(\mathbf{app} : t \Longrightarrow \mathbb{1})$. We already deduce that $t \leq \mathbb{1} \times \mathbb{1}$, because otherwise we could find a value $v$ such that $\vdash v : t$ and $v \overset{\mathbf{app}}{\rightsquigarrow} \Omega$.

If we can prove the exactness property for two types $t_1$ and $t_2$, we will deduce the property for type $t_1 \vee t_2$, by using rule (**app**$\vee$). Writing (Theorem 4.36)

$$t \simeq \bigvee_{(t_f, t_x) \in \pi(t)} t_f \times t_x$$

and decomposing every $t_f$ into the form

$$t_f \simeq \bigvee_{(P,N) \in t_f} \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a$$

we reduce the problem to the case where $t = t_f \times t_x$, with $t_f = \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a$, for two finite sets $P$ and $N$ of atomic arrow types, with $t_f \not\simeq \mathbb{0}$. Writing $P = \{t_i \to s_i \mid i \in I\}$ where $I = \{1, \ldots, n\}$.

Let $t_0$ be the type $\bigvee_{i \in I} t_i$. If $t_x \not\leq t_0$, then we choose a value $v_x$ in $t_x \setminus t_0$, and we take:

$$v_f = \mu f(P).\lambda x.(y = x \in t_0\ ?\ \mathbf{app}((f, x))|\mathbf{app}(f))$$

This expression has type $t_f$, because the second branch of the body is ignored during typing. The value $(v_f, v_x)$ then has type $t$ but $(v_f, v_x) \overset{\mathbf{app}}{\rightsquigarrow} e$ where $e$ is an ill-typed expression (because it reduces to $\mathbf{app}(v_f)$ which is ill-typed), which contradicts the assumption $(\mathbf{app} : t \Longrightarrow \mathbb{1})$.

Hence we can assume that $t_x \leq t_0$. Writing:

$$s = \bigvee_{I' \subseteq I \ \mid \ (*)} \bigwedge_{i \in I \setminus I'} s_i$$

where $(*)$ is the condition $t_x \not\leq \bigvee_{i \in I'} t_i$. Lemma 4.46 gives $\mathbf{app} : t \to s$. Let $v'$ be a value in $s$. We can find $I'$ which verifies $(*)$ and such that $v' \in \bigwedge_{i \in I \setminus I'} s_i$. Let $v_x$ be a value in $t_x \setminus \bigvee_{i \in I'} t_i$ and $v_f$ the value:

$$v_f = \mu f(P).\lambda x.(y = x \in \bigvee_{i \in I'} t_i \ ? \ \mathbf{app}((f, x)) | v')$$

We notice that $(v_f, v_x)$ does have type $t$, and that $(v_f, v_x) \overset{\mathbf{app}}{\rightsquigarrow} e$ with $e \overset{*}{\rightsquigarrow} v'$. □

## 5.7 Type inference

In this section, we will study the problem of type inference for the type system introduced in Section 5.2. The difficulty comes from the fact that the system does not have *principal types*: given a typing environment $\Gamma$ and an expression $e$ well-typed in $\Gamma$, there does not necessarily exist a type $t$ such that:

$$\forall s.\ (\Gamma \vdash e : s) \iff t \leq s$$

This essentially comes from typing rule (*abstr*) that always allows the addition of negative arrow types. Rule (*op*) can also prevent the existence of principal types.

### 5.7.1 Filters, schemas

The idea is to introduce syntactic objects called schemas that each represent a set of types, in order to express with a unique schema all of the types of a certain expression $e$ in an environment $\Gamma$. The grammar for schemas is:

$$\begin{array}{lcll} \mathbb{t} & ::= & t & t \in \widehat{T} \\ & \mid & [t_1 \to s_1; \ldots; t_n \to s_n] & n \geq 1; t_i, s_i \in \widehat{T} \\ & \mid & \mathbb{t}_1 \otimes \mathbb{t}_2 & \\ & \mid & \mathbb{t}_1 \oslash \mathbb{t}_2 & \\ & \mid & \Omega & \end{array}$$

We will also write $[t_i \to s_i]_{i=1..n}$ for $[t_1 \to s_1; \ldots; t_n \to s_n]$.

To each $\mathbb{t}$, we want to associate a set of types $\{\mathbb{t}\}$. Since the idea is to represent the set of types of an expression by a schema, we will be guided by the following definition and lemma.

**Definition 5.21** *A* **filter** *is a set of types $\mathcal{F} \subseteq \widehat{T}$ closed by intersection and subsumption:*
— $\forall t_1, t_2.\ t_1 \in \mathcal{F} \wedge t_2 \in \mathcal{F} \Rightarrow t_1 \mathbf{\wedge} t_2 \in \mathcal{F}$
— $\forall t_1, t_2.\ t_1 \in \mathcal{F} \wedge t_1 \leq t_2 \Rightarrow t_2 \in \mathcal{F}$

**Lemma 5.22** *Let $\Gamma$ be a typing environment, $e$ an expression, then the set $\{t \mid \Gamma \vdash e : t\}$ is a filter.*

> *Proof*: It is a consequence of rule (*subsum*) and Lemma 5.3. □

**Definition 5.23** *We define the function $\mathbf{\{\_\}}$ that associates to a schema a set of types:*

$$
\begin{array}{lcl}
\mathbf{\{}t\mathbf{\}} & = & \{s \mid t \leq s\} \\
\mathbf{\{}[t_i \mathbf{\rightarrow} s_i]_{i=1..n}\mathbf{\}} & = & \{s \mid \exists s_0 = \bigwedge_{i=1..n} (t_i \mathbf{\rightarrow} s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \mathbf{\rightarrow} s'_j).\ \mathbb{0} \not\simeq s_0 \leq s\} \\
\mathbf{\{}\mathbb{t}_1 \otimes \mathbb{t}_2\mathbf{\}} & = & \{s \mid \exists t_1 \in \mathbf{\{}\mathbb{t}_1\mathbf{\}}, t_2 \in \mathbf{\{}\mathbb{t}_2\mathbf{\}}.\ t_1 \mathbf{\times} t_2 \leq s\} \\
\mathbf{\{}\mathbb{t}_1 \oslash \mathbb{t}_2\mathbf{\}} & = & \{s \mid \exists t_1 \in \mathbf{\{}\mathbb{t}_1\mathbf{\}}, t_2 \in \mathbf{\{}\mathbb{t}_2\mathbf{\}}.\ t_1 \mathbf{\vee} t_2 \leq s\} \\
\mathbf{\{}\Omega\mathbf{\}} & = & \emptyset
\end{array}
$$

**Lemma 5.24** *For any schema $\mathbb{t}$, the set $\mathbf{\{}\mathbb{t}\mathbf{\}}$ is a filter. We can deduce whether it is empty.*

> *Proof*: By easy induction on the syntax of schemas. The case $[t_i \mathbf{\rightarrow} s_i]_{i=1..n}$ is dealt with by Lemma 5.1.
>
> If we see a schema $\mathbb{t}$ as a binary tree whose nodes are $\otimes$ and $\oslash$, then the filter $\mathbf{\{}\mathbb{t}\mathbf{\}}$ is empty if and only if one of the leafs is $\Omega$. □

**Remark 5.25** *It is easy to verify the equality:*

$$\mathbf{\{}\mathbb{t}_1 \oslash \mathbb{t}_2\mathbf{\}} = \mathbf{\{}\mathbb{t}_1\mathbf{\}} \cap \mathbf{\{}\mathbb{t}_2\mathbf{\}}$$

**Lemma 5.26** *Let $t_0$ be a type and $\mathbb{t}$ a schema. Then we can compute a schema, written $t_0 \oslash \mathbb{t}$, such that:*

$$\mathbf{\{}t_0 \oslash \mathbb{t}\mathbf{\}} = \{s \mid \exists t \in \mathbf{\{}\mathbb{t}\mathbf{\}}.t \mathbf{\wedge} t_0 \leq s\}$$

*and in addition $\#(t_0 \oslash \mathbb{t}) \leq \#\mathbb{t}$, where $\#\mathbb{t}$ denotes the maximal number of constructors $\otimes$ nested in $\mathbb{t}$.*

> *Proof*: The proof gives a construction for $t_0 \oslash \mathbb{t}$. The verification of the property is easy. We proceed by induction on the syntax of $\mathbb{t}$. If $\mathbb{t}$ is a type $t$, we take:
>
> $$t_0 \oslash t = t_0 \mathbf{\wedge} t$$
>
> If $\mathbb{t}$ is a union $\mathbb{t}_1 \oslash \mathbb{t}_2$, we distribute:
>
> $$t_0 \oslash (\mathbb{t}_1 \oslash \mathbb{t}_2) = (t_0 \oslash \mathbb{t}_1) \oslash (t_0 \oslash \mathbb{t}_2)$$
>
> If $\mathbb{t}$ is $\Omega$, we take $t_0 \oslash \mathbb{t} = \Omega$.
>
> In both cases $[t_i \mathbf{\rightarrow} s_i]_{i=1..n}$ and $\mathbb{t}_1 \otimes \mathbb{t}_2$, we will use Lemma 3.1, that allows us to see $t_0$ as a boolean combination of atoms. The

two following equations bring us back to the case where $t_0$ is an atom or a negation of an atom:

$$(t_1 \vee t_2) \owedge \mathbb{t} = (t_1 \owedge \mathbb{t}) \ovee (t_2 \owedge \mathbb{t})$$

$$(t_1 \wedge t_2) \owedge \mathbb{t} = t_1 \owedge (t_2 \owedge \mathbb{t})$$

For the case $\mathbb{t} = \mathbb{t}_1 \otimes \mathbb{t}_2$, we take:

$$(t_1 \times t_2) \owedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = (t_1 \owedge \mathbb{t}_1) \otimes (t_2 \owedge \mathbb{t}_2)$$

$$\neg(t_1 \times t_2) \owedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = ((\neg t_1 \owedge \mathbb{t}_1) \otimes \mathbb{t}_2) \ovee (\mathbb{t}_1 \otimes (\neg t_2 \owedge \mathbb{t}_2))$$

and if $a$ is an atom that is not of the form $t_1 \times t_2$:

$$a \owedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = \mathbb{0}$$

$$\neg a \owedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = (\mathbb{t}_1 \otimes \mathbb{t}_2)$$

For the case $\mathbb{t} = [t_i \to s_i]_{i=1..n}$, we take:

$$(t \to s) \owedge [t_i \to s_i]_{i=1..n} = \begin{cases} [t_i \to s_i]_{i=1..n} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \leq t \to s \\ \mathbb{0} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \not\leq t \to s \end{cases}$$

$$\neg(t \to s) \owedge [t_i \to s_i]_{i=1..n} = \begin{cases} \mathbb{0} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \leq t \to s \\ [t_i \to s_i]_{i=1..n} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \not\leq t \to s \end{cases}$$

and if $a$ is an atom that is not of the form $t \to s$:

$$a \owedge [t_i \to s_i]_{i=1..n} = \mathbb{0}$$

$$\neg a \owedge [t_i \to s_i]_{i=1..n} = [t_i \to s_i]_{i=1..n}$$

□

**Lemma 5.27** *Let $\mathbb{t}$ be a schema and $t$ a type. Then we can decide whether the assertion $t \in \{\!|\mathbb{t}|\!\}$ is true or false. We write $\mathbb{t} \leq t$ when it is true.*

*Proof*: Let us start by observing the equivalence the equivalence:

$$t \in \{\!|\mathbb{t}|\!\} \iff \mathbb{0} \in \{\!|(\neg t) \owedge \mathbb{t}|\!\}$$

We find indeed that: $\mathbb{0} \in \{\!|(\neg t) \owedge \mathbb{t}|\!\} \iff \exists s \in \{\!|\mathbb{t}|\!\}.\ (\neg t) \wedge s \leq \mathbb{0} \iff \exists s \in \{\!|\mathbb{t}|\!\}.\ s \leq t \iff t \in \{\!|\mathbb{t}|\!\}$.

This brings us back to the case $t = \mathbb{0}$, and we conclude by induction on the syntax of $\mathbb{t}$, by using:

$$\begin{array}{lll} \mathbb{0} \in \{\!|t|\!\} & \iff & t \simeq \mathbb{0} \\ \mathbb{0} \notin \{\!|[t_i \to s_i]_{i=1..n}|\!\} & & \\ \mathbb{0} \in \{\!|\mathbb{t}_1 \otimes \mathbb{t}_2|\!\} & \iff & (\mathbb{0} \in \{\!|\mathbb{t}_1|\!\}) \vee (\mathbb{0} \in \{\!|\mathbb{t}_2|\!\}) \\ \mathbb{0} \in \{\!|\mathbb{t}_1 \ovee \mathbb{t}_2|\!\} & \iff & (\mathbb{0} \in \{\!|\mathbb{t}_1|\!\}) \wedge (\mathbb{0} \in \{\!|\mathbb{t}_2|\!\}) \\ \mathbb{0} \notin \{\!|\Omega|\!\} & & \end{array}$$

□

### 5.7.2 Typing the operators

We have almost all the necessary ingredients to reformulate the type system using schemas. All that remains is to assume that the operators behave well with regard to the schemas. For an operator $o$ and a schema $\mathbb{t}$, we consider the set:

$$\{s \mid \exists t' \geq \mathbb{t}.\exists s' \leq s.\ o : t'\rightarrow s'\}$$

It is clear that it is a filter.

**Definition 5.28** *A* **typing function** *for the operator $o$ is a function from schemas to schemas, written $o[\_]$, such that, for any schema $\mathbb{t}$:*

$$\mathbf{\{}o[\mathbb{t}]\mathbf{\}} = \{s \mid \exists t' \geq \mathbb{t}.\exists s' \leq s.\ o : t'\rightarrow s'\}$$

**Lemma 5.29** *There exists a computable typing function for the* **app** *operator .*

*Proof*: The proofs rests on the Theorem 4.42 and Lemma 4.37. For a filter $\mathcal{F}$, let us write:

$$\begin{array}{rcl} X_{\mathcal{F}} & = & \{s \mid \exists t' \in \mathcal{F}.\exists s' \leq s.\ \mathbf{app} : t'\rightarrow s'\} \\ & = & \{s \mid \exists t' \in \mathcal{F}.\ \mathbf{app} : t'\rightarrow s\} \end{array}$$

By using Theorem 4.42, wee see that if $\mathbb{t} \not\leq (\mathbb{0}\rightarrow\mathbb{1}) \times \mathbb{1}$, then $X_{\mathbf{\{}\mathbb{t}\mathbf{\}}} = \emptyset$, and in that case, we take $\mathbf{app}[\mathbb{t}] = \Omega$. This makes it possible to deal in particular with cases $\mathbb{t} = \Omega$, $\mathbb{t} = [t_i\rightarrow s_i]_{i=1..n}$ and $\mathbb{t} = (\mathbb{t}_1 \otimes \mathbb{t}_2) \otimes \mathbb{t}_3$. If $\mathbb{t} \leq \mathbb{0}$, we take $\mathbf{app}[\mathbb{t}] = \mathbb{0}$. From now on, we assume that $\mathbb{t} \leq (\mathbb{0}\rightarrow\mathbb{1}) \times \mathbb{1}$ and $\mathbb{t} \not\leq \mathbb{0}$.

By using rule (**appV**), we immediately establish the equality:

$$X_{\mathcal{F}_1 \cap \mathcal{F}_2} = X_{\mathcal{F}_1} \cap X_{\mathcal{F}_2}$$

This allows us to write:

$$\begin{array}{lcl} \mathbf{app}[\mathbb{t}_1 \vee \mathbb{t}_2] & := & \mathbf{app}[\mathbb{t}_1] \vee \mathbf{app}[\mathbb{t}_2] \\ \mathbf{app}[(\mathbb{t}_1 \vee \mathbb{t}'_1) \otimes \mathbb{t}_2] & := & \mathbf{app}[\mathbb{t}_1 \otimes \mathbb{t}_2] \vee \mathbf{app}[\mathbb{t}'_1 \otimes \mathbb{t}_2] \\ \mathbf{app}[t] & := & \bigvee\limits_{(t_1,t_2)\in\pi(t)} \mathbf{app}[t_1 \otimes t_2] \end{array}$$

By using these definitions, we come back to the case $\mathbb{t}_1 \otimes \mathbb{t}_2$ where $\mathbb{t}_1$ is either a type $t_1$ (with $t_1 \leq \mathbb{0}\rightarrow\mathbb{1}$ and $t_1 \not\leq \mathbb{0}$) or of the form $[t'_i\rightarrow s'_i]_{i=1..n}$. Corollary 4.45 gives:

$$X_{\mathbf{\{}\mathbb{t}_1\otimes\mathbb{t}_2\mathbf{\}}} = \{s \mid \exists t_2 \geq \mathbb{t}_2.\ \mathbb{t}_1 \leq t_2\rightarrow s\}$$

Insofar as:

$$[t'_i\rightarrow s'_i]_{i=1..n} \leq t_2\rightarrow s \iff \bigwedge_{i=1..n} t'_i\rightarrow s'_i \leq t_2\rightarrow s$$

we are brought back to the unique case where $\mathbb{t}_1$ is a type $t_1$, and we have:

$$X_{\mathbf{\{}t_1\otimes\mathbb{t}_2\mathbf{\}}} = \{s \mid \exists t_2 \geq \mathbb{t}_2.\ t_1 \leq t_2\rightarrow s\}$$

Lemma 4.37 gives:

$$t_1 \leq t_2 \to s \iff \begin{cases} t_2 \leq \mathrm{Dom}(t_1) \\ \forall (s_1, s_2) \in \rho(t_1).\ (t_2 \leq s_1) \vee (s_2 \leq s) \end{cases}$$

We easily obtain the following equivalences:

$$\begin{array}{ll} & \exists t_2 \geq \mathbb{t}_2.\ t_1 \leq t_2 \to s \\ \iff & \exists \mathbb{t}_2 \leq t_2 \leq \mathrm{Dom}(t_1).\ \forall (s_1, s_2) \in \rho(t_1).\ (t_2 \leq s_1) \vee (s_2 \leq s) \\ \overset{*}{\iff} & \mathbb{t}_2 \leq \mathrm{Dom}(t_1) \wedge \forall (s_1, s_2) \in \rho(t_1).\ (\mathbb{t}_2 \leq s_1) \vee (s_2 \leq s) \\ \iff & \mathbb{t}_2 \leq \mathrm{Dom}(t_1) \wedge s \geq \bigvee_{(s_1,s_2)\in\rho(t_1)\ |\ \mathbb{t}_2 \not\leq s_1} s_2 \end{array}$$

The only implication that deserves a comment is the $\Leftarrow$ direction of the equivalence marked by a star. Assume that for each pair $(s_1, s_2) \in \rho(t_1)$, we have $\mathbb{t}_2 \leq s_1$ or $s_2 \leq s$. We then define $t_2$ as the intersection of $\mathrm{Dom}(t_1)$ and some types $s_1$ such that $(s_1, s_2) \in \rho(t_1)$ and $\mathbb{t}_2 \leq s_1$. We have $\mathbb{t}_2 \leq t_2$.

We finally deduce:

$$\begin{array}{lcll} \mathbf{app}[t_1 \otimes \mathbb{t}'2] & := & \Omega & \text{if } \mathbb{t}_2 \not\leq \mathrm{Dom}(t_1) \\ & := & \bigvee_{(s_1,s_2)\in\rho(t_1)\ |\ \mathbb{t}_2 \not\leq s_1} s_2 & \text{if } \mathbb{t}_2 \leq \mathrm{Dom}(t_1) \end{array}$$

□

### 5.7.3 Typing algorithm

For the rest of this section, we assume that all operators have a computable typing function. We can then define a typing algorithm. A "schematic typing environment" $\mathbb{\Gamma}$ is a partial function, with finite domain, from variables to schemas, such that $\{\mathbb{\Gamma}(x)\} \neq \emptyset$ for all $x$ in the domain of $\mathbb{\Gamma}$. The algorithm is presented in Figure 5.3 in the form of a function that associates to an environment $\mathbb{\Gamma}$ and to an expression $e$ a schema, written $\mathbb{\Gamma}[e]$. This function is defined by induction on the syntax of $e$.

If $\Gamma$ (resp. $\mathbb{\Gamma}$) is a typing environment (resp. schematic typing environment), then we write $\mathbb{\Gamma} \leq \Gamma$ when $\mathbb{\Gamma}$ and $\Gamma$ have the same domain and for all $x$: $\mathbb{\Gamma}(x) \leq \Gamma(x)$.

**Lemma 5.30** *If $\mathbb{\Gamma} \leq \Gamma_1$ and $\mathbb{\Gamma} \leq \Gamma_2$, then there exists an environment $\Gamma$ such that: $\mathbb{\Gamma} \leq \Gamma$, $\Gamma \leq \Gamma_1$, $\Gamma \leq \Gamma_2$.*

*Proof*: The three environments $\mathbb{\Gamma}$, $\Gamma_1$ and $\Gamma_2$ have the same domain. If $x$ is in that domain, we fix $\Gamma(x) = \Gamma_1(x) \mathbf{\wedge} \Gamma_2(x)$, otherwise $\Gamma(x)$ is not defined. □

**Lemma 5.31 (Correction)** *If $\mathbb{\Gamma}[e] \leq t$, then there exists $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e : t$.*

*Proof*: By induction on the structure of expression $e$. We deal with the two most interesting cases.
Case $e' = (x = e \in t\ ?\ e_1 | e_2)$. We take $\mathbb{t}_0, \mathbb{t}_1, \mathbb{t}_2, \mathbb{s}_1, \mathbb{s}_2$ as in the definition of $\mathbb{\Gamma}[(x = e \in t\ ?\ e_1|e_2)]$. Let $s_1, s_2$ such that $\mathbb{s}_1 \leq s_1$

$$
\begin{cases}
\mathbb{\Gamma}[c] := b_c \\
\mathbb{\Gamma}[(e_1, e_2)] := \mathbb{\Gamma}[e_1] \otimes \mathbb{\Gamma}[e_2] \\
\mathbb{\Gamma}[\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e] := \begin{cases} \mathbb{t} & \text{if } \forall i = 1..n.\ \mathbb{s}_i \leq s_i \\ \Omega & \text{otherwise} \end{cases} \\
\qquad\qquad \text{where } \begin{cases} \mathbb{t} = [t_i \rightarrow s_i]_{i=1..n} \\ \mathbb{s}_i = ((f : \mathbb{t}), (x : t_i), \mathbb{\Gamma})[e] \quad (i = 1..n) \end{cases} \\
\mathbb{\Gamma}[x] := \begin{cases} \mathbb{\Gamma}(x) & \text{if } \mathbb{\Gamma}(x) \text{ is defined} \\ \Omega & \text{otherwise} \end{cases} \\
\mathbb{\Gamma}[o(e)] := o[\mathbb{\Gamma}[e]] \\
\mathbb{\Gamma}[(x = e \in t\ ?\ e_1 | e_2)] := \mathbb{s}_1 \mathbin{\mathbb{\oslash}} \mathbb{s}_2 \\
\text{where } \begin{cases} \mathbb{t}_0 := \mathbb{\Gamma}[e] \\ \mathbb{t}_1 := t \mathbin{\mathbb{\oslash}} \mathbb{t}_0 \\ \mathbb{t}_2 := (\neg t) \mathbin{\mathbb{\oslash}} \mathbb{t}_0 \\ \mathbb{s}_i := \begin{cases} ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i] & \text{if } \mathbb{t}_i \not\leq \mathbb{0}, \{\mathbb{t}_i\} \neq \emptyset \\ \mathbb{0} & \text{if } \mathbb{t}_i \leq \mathbb{0} \\ \Omega & \text{if } \{\mathbb{t}_i\} = \emptyset \end{cases} \quad (i = 1..2) \end{cases}
\end{cases}
$$

Figure 5.3: Typing algorithm

and $\mathbb{s}_2 \leq s_2$. The goal is to see there exists $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e' : s_1 \vee s_2$. Taking $i = 1..2$. Since $\mathbb{s}_i \leq s_i$, we have $\mathbb{s}_i \neq \Omega$. There are two cases. If $\mathbb{t}_i \not\leq \mathbb{0}$, then $\mathbb{s}_i = ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i] \leq s_i$. According to the induction hypothesis, there exists $\Gamma_i \geq \mathbb{\Gamma}$ and $t_i \geq \mathbb{t}_i$ such that $(x : t_i), \Gamma_i \vdash e_i : s_i$. If $\mathbb{t}_i \leq \mathbb{0}$, then we fix $t_i = \mathbb{0}$.

Writing $t_0 = (t_1 \wedge t) \vee (t_2 \setminus t)$. Proving that $t_0 \geq \mathbb{t}_0$. We have $t_1 \geq \mathbb{t}_1 = t \mathbin{\mathbb{\oslash}} \mathbb{t}_0$, so there exists $t'_1 \geq \mathbb{t}_0$ such that $t \wedge t'_1 \leq t_1$. Similarly, there exists $t'_2 \geq \mathbb{t}_0$ such that $(\neg t) \wedge t'_2 \leq t_2$. We deduce: $t_0 \geq (t \wedge t'_1 \wedge t'_2) \vee ((\neg t) \wedge t'_1 \wedge t'_2) \simeq t'_1 \wedge t'_2 \geq \mathbb{t}_0$. Writing $t''_1 = t_0 \wedge t \leq t_1$ and $t''_2 = t_0 \setminus t \leq t_2$.

We can then apply the induction hypothesis again, which gives the existence of a $\Gamma_0 \geq \mathbb{\Gamma}$ such that $\Gamma_0 \vdash e : t_0$. By using Lemmas 5.2 and 5.30, we see there exists an environment $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e : t_0$, and $(x : t''_i), \Gamma \vdash e_i : s_i$ for $i = 1..2$ (when $\mathbb{t}_i \not\leq \mathbb{0}$). We conclude, by using rule (*case*) and possibly (*subsum*), that $\Gamma \vdash e' : s_1 \vee s_2$.

Case $e' = \mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e$. Taking $\mathbb{t}$ and $\mathbb{s}_i$ as in the definition of $\mathbb{\Gamma}[e']$. Since $\mathbb{\Gamma}[e'] \neq \Omega$ (because $\mathbb{\Gamma}[e'] \leq t$), we have $\mathbb{\Gamma}[e'] = \mathbb{t}$ and $\mathbb{s}_i \leq s_i$ for any $i = 1..n$. The induction hypothesis shows that for every $i$, we can find an environment $\Gamma = ((f : t^i), (x : t''_i), \Gamma_i) \geq ((f : \mathbb{t}), (x : t_i), \mathbb{\Gamma})$ such that $\Gamma \vdash e : s_i$. Writing $t' = \bigwedge_{i=1..n} t^i \wedge t$.

We have $t' \geq \mathbb{t}$, so we can find a family $(t'_j \rightarrow s'_j)_{j=1..m}$ such that $t' \geq t'' \not\simeq \mathbb{0}$ where $t'' = \bigwedge_{i=1..n} (t_i \rightarrow s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \rightarrow s'_j)$.

By using Lemma 5.2 and 5.30, we obtain $\Gamma \geq \mathbb{\Gamma}$ such that, for

all $i$: $(f : t''), (x : t_i), \Gamma \vdash e : s_i$. We can apply rule (*abstr*) to obtain $\Gamma \vdash e' : t''$, which allows us to conclude, since $t'' \leq t$. □

**Lemma 5.32 (Completeness)** *If* $\mathbb{\Gamma} \leq \Gamma$ *and* $\Gamma \vdash e : t$, *then* $\mathbb{\Gamma}[e] \leq t$.

*Proof*: By induction on the typing derivation $\Gamma \vdash e : t$. The proof is mechanical. Consider for example the case where the derivation ends with an instance of rule (*case*):

$$\frac{\Gamma \vdash e : t_0 \quad \begin{cases} t_1 = t_0 \mathbin{\mathsf{\wedge}} t \\ t_2 = t_0 \mathbin{\mathsf{\setminus}} t \end{cases} \quad \begin{cases} s_i = \mathbb{0} & \text{if } t_i \simeq \mathbb{0} \\ (x : t_i), \Gamma \vdash e_i : s_i & \text{if } t_i \not\simeq \mathbb{0} \end{cases}}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s_1 \mathbin{\mathsf{\vee}} s_2}$$

We take $\mathbb{t}_0, \mathbb{t}_1, \mathbb{t}_2, \mathbb{s}_1, \mathbb{s}_2$ as in the definition of $\mathbb{\Gamma}[(x = e \in t \ ?\ e_1 | e_2)]$. By applying the induction hypothesis, we obtain $\mathbb{t}_0 = \mathbb{\Gamma}[e] \leq t_0$, and deduce $\mathbb{t}_1 \leq t_1$ and $\mathbb{t}_2 \leq t_2$. Taking $i = 1..2$. If $\mathbb{t}_i \leq \mathbb{0}$, then $\mathbb{s}_i = \mathbb{0}$ and then $\mathbb{s}_i \leq s_i$. Otherwise, since $\{\!|\mathbb{t}_i|\!\} \neq \emptyset$, we have $\mathbb{s}_i = ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i]$. But $t_i \not\leq \mathbb{0}$ (because otherwise $\mathbb{t}_i \leq \mathbb{0}$), therefore $(x : t_i), \Gamma \vdash e_i : s_i$. The induction hypothesis still gives $\mathbb{s}_i \leq s_i$ in that case. Finally, we do have $\mathbb{s}_1 \mathbin{\mathbb{\vee}} \mathbb{s}_2 \leq s_1 \mathbin{\mathsf{\vee}} s_2$. □

By combining the two previous lemmas, we obtain the following theorem.

**Theorem 5.33** *Let* $\mathbb{\Gamma}$ *be a schematic typing environment and* $e$ *an expression. Then:*

$$\{\!|\mathbb{\Gamma}[e]|\!\} = \{t \mid \exists \Gamma.\ (\mathbb{\Gamma} \leq \Gamma) \wedge (\Gamma \vdash e : t)\}$$

The next corollary gives an algorithm to decide whether an expression is well-typed in a typing environment.

**Corollary 5.34** *Let* $\Gamma$ *be a typing environment. We can also see it as a schematic typing environment. Then, for all expression* $e$ *and all type* $t$, *we have:*

$$\Gamma \vdash e : t \iff \Gamma[e] \leq t$$

*In particular, expression* $e$ *is well-typed in the environment* $\Gamma$ *if and only if* $\{\!|\Gamma[e]|\!\} \neq \emptyset$.

### 5.7.4 Set-theoretic interpretation of schemas

We introduced diagrams as syntactic representatives of sets of types assigned to a value by the type system. A dual vision consists in saying that a *non-empty* schema represents a set of values, defined by:

$$[\![\mathbb{t}]\!]_{\mathcal{V}} := \bigcap_{t \in \{\!|\mathbb{t}|\!\}} [\![t]\!]_{\mathcal{V}}$$

This definition is consistent with the case where $\mathbb{t}$ is a type. We will see that the different operators that we have introduced on schemas behave as expected from a set-theoretic point of view. The following properties are easy to establish:

$$[\![\mathbb{t}_1 \otimes \mathbb{t}_2]\!]_{\mathcal{V}} = [\![\mathbb{t}_1]\!]_{\mathcal{V}} \times [\![\mathbb{t}_2]\!]_{\mathcal{V}}$$

$$[\![\mathbb{t}_1 \vee \mathbb{t}_2]\!]_{\mathcal{V}} = [\![\mathbb{t}_1]\!]_{\mathcal{V}} \cup [\![\mathbb{t}_2]\!]_{\mathcal{V}}$$
$$[\![t \wedge \mathbb{t}]\!]_{\mathcal{V}} = [\![t]\!]_{\mathcal{V}} \cap [\![\mathbb{t}]\!]_{\mathcal{V}}$$

The set $[\![[t_i \to s_i]_{i=1..n}]\!]_{\mathcal{V}}$ contains all abstractions whose intersection of interface types is equivalent to $\bigwedge_i (t_i \to s_i)$; in particular, it is never empty. We then see, by induction over $\mathbb{t}$, that $[\![\mathbb{t}]\!]_{\mathcal{V}} = \emptyset \iff \mathbb{t} \leq \mathbb{0}$, and deduce that $\mathbb{t} \leq t \iff [\![\mathbb{t}]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}}$ (because $\mathbb{t} \leq t \iff (\neg t) \wedge \mathbb{t} \leq \mathbb{0}$).

It is interesting to note that these properties generally become false if schemas are interpreted as sets of elements of a model. For example, in the universal model of Section 4.5, the set

$$\bigcap_{t \in \{\mathbb{t}\}} [\![t]\!]^0$$

is always empty for $\mathbb{t} = [\mathbb{0} \to \mathbb{0}]$. An element of this set would indeed be a graph *fini* $f = \{(d_1, d'_1), \ldots, (d_n, d'_n)\}$ with $d'_i \in D^0_\Omega$. But $\mathbb{t} \leq \neg(t \to \mathbb{1})$ as soon as $t \not\simeq \mathbb{0}$, so for any non-empty type $t$, there must be a pair $(d_i, \Omega)$ in $f$, with $d_i$ in $[\![t]\!]^0$. Since we build an infinite number of type $t$ two-to-two disjoint, this contradicts the finiteness of $f$.

This may seem contradictory to the fact that the model for values is equivalent to the original model, i.e. the interpretations of a type in either model are simultaneously empty or non-empty. In fact, we have simply shown that this property does not translate to schemas which are, intuitively, *infinite* type intersections. This is why we do not put forward the set-theoretic interpretation of schemas in the formalism introduced.

## 5.8 Operators

The calculus we have introduced is parameterized by a set of operators $\mathcal{O}$. For each operator $o \in \mathcal{O}$, we have to define:

— a binary relation $\_ \overset{o}{\rightsquigarrow} \_$ where the first argument is a value and the second is either an expression or $\Omega$;
— a binary relation between types $o : \_ \to \_$, closed by intersection (if $o : t_1 \to s_1$ and $o : t_2 \to s_2$, then $o : (t_1 \wedge t_2) \to (s_1 \wedge s_2)$) and such that $o$ is well-typed (Definition 5.14);
— a typing function (Definition 5.28).

Of course, the choice of operators depends on the base types chosen. For example, if we have a base type `int` that represents integers, we will probably have an operator $+$ with $+ : \texttt{int} \times \texttt{int} \to \texttt{int}$.

However, we can define generic operators, which do not depend on the base types chosen. We have already studied the **app** operator . We will now introduce the first projection operator $\pi_1$ (operator $\pi_2$ is defined similarly). Its semantics are given by: $(v_1, v_2) \overset{\pi_1}{\rightsquigarrow} v_1$ and $v \overset{\pi_1}{\rightsquigarrow} \Omega$ when $v$ is not of the form $(v_1, v_2)$. Relation $\pi_1 : \_ \to \_$ is given by the inductive system of Figure 5.4.

**Lemma 5.35** *The operator $\pi_1$ is well-typed.*

> *Proof*: According to Lemma 5.17, we only have to consider rule $(\pi_1 \times)$. If $\vdash v : t_1 \times t_2$ and $v \overset{\pi_1}{\rightsquigarrow} e$, then we have to show that $e \neq \Omega$ and $\vdash e : t_1$. But according to Lemma 5.12, if $\vdash v : t_1 \times t_2$, we have

$$\frac{}{\pi_1 : (t_1 \times t_2) \to t_1}\ (\pi_1\times)$$

$$\frac{\pi_1 : t_1 \to s_1 \quad \pi_1 : t_2 \to s_2}{\pi_1 : (t_1 \wedge t_2) \to (s_1 \wedge s_2)}\ (\pi_1\wedge)$$

$$\frac{\pi_1 : t_1 \to s_1 \quad \pi_1 : t_2 \to s_2}{\pi_1 : (t_1 \vee t_2) \to (s_1 \vee s_2)}\ (\pi_1\vee)$$

$$\frac{\pi_1 : t' \to s' \quad t \leq t' \quad s' \leq s}{\pi_1 : t \to s}\ (\pi_1 \leq)$$

Figure 5.4: Axiomatizing the typing of the first projection

$v = (v_1, v_2)$ with $\vdash v_1 : t_1$, and the only possible reduction for $v$ is $v \overset{\pi_1}{\rightsquigarrow} v_1$. □

Now let us look at the typing of the $\pi_1$ operator. We begin by establishing an equivalent to Theorem 4.42. The proof is more direct because we can directly interpret the first projection in model $\mathbb{E}D$.

**Lemma 5.36** *Let $t$ and $s$ be two types. The following assertions are equivalent:*
*(i)* $\pi_1 : t \to s$
*(ii)* $t \leq s \times \mathbb{1}$ *(i.e.:* $\mathbb{E}[\![t]\!] \subseteq [\![s]\!] \times D$*)*
*(iii)* $(t \leq \mathbb{1} \times \mathbb{1}) \wedge \forall (t_1, t_2) \in \pi(t).\ t_1 \leq s$

*Proof*: The proof of $(i) \Rightarrow (ii)$ is an easy induction on the derivation of $\pi_1 : t \to s$. Proving $(ii) \Rightarrow (iii)$. If $(t_1, t_2) \in \pi(t)$, we have $t_1 \times t_2 \leq t$, which gives $t_1 \times t_2 \leq s \times \mathbb{1}$, and since $t_2 \not\simeq \mathbb{0}$, we do obtain $t_1 \leq s$. The proof of $(iii) \Rightarrow (i)$ is similar to the one of Theorem 4.42. □

**Corollary 5.37** *Let $t_1$, $t_2$ and $s$ be three types. Then:*

$$\pi_1 : t_1 \times t_2 \to s \iff (t_1 \leq s) \vee (t_2 \simeq \mathbb{0})$$

It is easy to see then that the function $\pi_1[\_]$ defined below is a typing function for $\pi_1$:

$$\begin{array}{lcl}
\pi_1[t] & = & \begin{cases} \bigvee_{(t_1,t_2)\in\pi(t)} t_1 & \text{if } t \leq \mathbb{1}\times\mathbb{1} \\ \Omega & \text{otherwise} \end{cases} \\
\pi_1[t_i \to s_i]_{i=1..n} & = & \Omega \\
\pi_1[\mathbb{t}_1 \otimes \mathbb{t}_2] & = & \begin{cases} \mathbb{0} & \text{if } \mathbb{t}_2 \leq \mathbb{0} \\ \mathbb{t}_1 & \text{otherwise} \end{cases} \\
\pi_1[\mathbb{t}_1 \oslash \mathbb{t}_2] & = & \pi_1[\mathbb{t}_1] \oslash \pi_1[\mathbb{t}_2] \\
\pi_1[\Omega] & = & \Omega
\end{array}$$

**Lemma 5.38** *The typing of the $\pi_1$ operator is exact.*

*Proof*: Let $t$ such that $(\pi_1 : t \Longrightarrow \mathbb{1})$. We can see that it means $t \leq \mathbb{1}\times\mathbb{1}$. Writing $s = \pi_1[t]$. It is easy to see that $\pi_1 : t\to s$. Let $v'$ be a value in $s$. We can find a pair $(t_1, t_2) \in \pi(t)$ such that $v' \in [\![t_1]\!]$. Writing $v = (v', v_2)$ where $v_2$ is an arbitrary value of type $t_2$ (there is one because $t_2$ is non-empty). We do have $v \overset{\pi_1}{\rightsquigarrow} v'$. □

## 5.9 Variant of the calculus without overloaded functions

The $\lambda$-abstractions of the calculus introduced in this chapter represent overloaded functions: their interface can specify several arrow types simultaneously. Consider the syntactic restriction that only allows for a single arrow. The syntax of abstractions is then:

$$\mu f(t\to s).\lambda x.e$$

This syntactic restriction is stable by our semantics. In particular, the application operator does not introduce overloaded functions in a program that does not contain any. We immediately infer that type preservation through reduction is still valid in the restricted calculus.

However, the subtyping relation, induced by the original model, no longer corresponds to inclusion between sets of values. In other words, Theorem 5.6 is wrong. Let us see why. Let $t_1, t_2, s_1, s_2$ be four types. Consider the type $(t_1\to s_1)\wedge(t_2\to s_2)$. Values of that type are well-typed abstractions of the form $\mu f(t\to s).\lambda x.e$, that have both types $t_1\to s_1$ and $t_2\to s_2$, that is, such that $t\to s \leq t_i\to s_i$ for $i = 1..2$. But $t\to s \leq t_i\to s_i$ decomposes into $(t \simeq \mathbb{0}) \vee (t_i \leq t \wedge s \leq s_i)$. So the condition is equivalent to $(t \simeq \mathbb{0}) \vee (t_1\vee t_2 \leq t \wedge s \leq s_1\wedge s_2)$, or too $t\to s \leq (t_1\vee t_2)\to(s_1\wedge s_2)$. This reasoning shows that in the restricted calculus, we have the property:

$$[\![(t_1\to s_1)\wedge(t_2\to s_2)]\!]_{\mathcal{V}} = [\![(t_1\vee t_2)\to(s_1\wedge s_2)]\!]_{\mathcal{V}}$$

But it is easy to see that the subtyping

$$(t_1\to s_1)\wedge(t_2\to s_2) \leq (t_1\vee t_2)\to(s_1\wedge s_2)$$

is generally false.

We will show in the rest of this section how to recover Theorem 5.6. The extensional interpretation of functions as graphs (non-deterministic binary relations) in the definition of a model is consistent with the possibility of defining several arrow types in the interface of abstractions. To recover Theorem 5.6 in the restricted calculus, we have to change this interpretation. Instead of interpreting functions as graphs, we will stick closer to the calculus. We modify the definition of the extensional interpretation with:

$$\mathbb{E}D := \mathcal{C} + D \times D + \mathcal{P}(D) \times \mathcal{P}(D)$$

and

$$X \to Y := \{(X', Y') \mid X \subseteq X' \wedge Y' \subseteq Y\}$$

Intuitively, we only retain the interface of functions, given by a pair $(X, Y)$, with $X \subseteq D, Y \subseteq D$. The set $X$ (resp. $Y$) represents the domain (resp. the

codomain) of the function. The extensional interpretation of the type $t{\rightarrow}s$ is the set of "functions" $(X,Y)$ with $[\![t]\!] \subseteq X$ and $Y \subseteq [\![s]\!]$, that is, those whose interface gives a constraint as least as strong as type $t{\rightarrow}s$. To study the subtyping relation that we obtain, we have to adapt Lemma 4.9. We use those two simple observations. First of all, we find that:

$$\bigcap_{i\in P} X_i \to Y_i = (\bigcup_{i\in P} X_i) \to (\bigcap_{i\in P} Y_i)$$

This allows us to treat intersections. We then study the assertion

$$X \to Y \subseteq \bigcup_{i\in N} X_i \to Y_i$$

In the left hand side, we find element $(X,Y)$, which must also be found, if the assertion is correct, in one of the $X_i \to Y_i$, which means $X_i \subseteq X \wedge Y \subseteq Y_i$. Conversely, if $(X,Y) \in X_i \to Y_i$ for a certain $i \in N$, then the assertion is true. We deduce the equivalent to Lemma 4.9:

**Lemma 5.39** *Let $(X_i)_{i\in P}, (X_i)_{i\in N}$, $(Y_i)_{i\in P}, (Y_i)_{i\in N}$ be four families of parts of a set $D$. Then:*

$$\bigcap_{i\in P} X_i \to Y_i \subseteq \bigcup_{i\in N} X_i \to Y_i$$
$$\Longleftrightarrow$$
$$\exists i_0 \in N.\ X_{i_0} \subseteq \bigcup_{i\in N} X_i \wedge \bigcap_{i\in P} Y_i \subseteq Y_{i_0}$$

The rest of the theory is easily adapted. For example, condition $C_{\mathbf{fun}}$ in the Definition 4.11 becomes:

$$C_{\mathbf{fun}} ::= \exists \theta'_1{\rightarrow}\theta'_2 \in N.\ \begin{cases} \tau(\theta'_1) \backslash \left( \bigvee_{\theta_1 \times \theta_2 \in P} \tau(\theta_1) \right) \in \mathcal{S} \\ \left( \bigwedge_{\theta_1 \times \theta_2 \in P} \tau(\theta_2) \right) \backslash \tau(\theta'_2) \in \mathcal{S} \end{cases}$$

To build a universal model, we always go back to using finite sets to represent function. We define $X \to_f Y$ as the subset of $X \to Y$ made of pairs $(X',Y')$ with finite $Y'$ and *cofinite* $X'$ .

Lemma 5.1, which expresses the strong disjunction of arrow types (an intersection is included in a union if and only if it is included in one of the terms of the union) is still valid.

We adapt the typing of the application by taking the approach of Section 4.8. For some pair $(f,x)$ with $f = (X,Y) \in \mathbb{E}_{\mathbf{fun}}D = \mathcal{P}(D) \times \mathcal{P}(D)$ and $x \in D$, we take $(f,x) \triangleright y$ if $(x \in X \Rightarrow y \in Y)$. This makes Theorem 4.42 work. Corollary 4.45 now simply becomes:

$$\mathbf{app} : t_1 \times t_2 {\rightarrow} s \iff t_1 \leq t_2 {\rightarrow} s$$

**Remark 5.40** *Our new definition of $X \to Y$ ignores the special case $X = \emptyset$ (which gives rives to condition $t_2 \simeq \mathbb{0}$ in Corollary 4.45). We could have defined $X \to Y$ as the set of pairs $(X',Y')$ with $(X \subseteq X' \wedge Y \subseteq Y') \vee X = \emptyset$, to keep this special case.*

To study the subtyping $t_1 \leq t_2 \to s$, we can set up an equivalent result to Lemma 4.37. In addition to the domain, we can define the codomain of a function type.

**Lemma 5.41** *Let $t$ be a type such that $t \leq \mathbb{0} \to \mathbb{1}$. Then there exists two types $Dom(t)$ and $Codom(t)$ such that:*

$$\forall t_1, t_2.\ (t \leq t_1 \to t_2) \iff \begin{cases} t_1 \leq Dom(t) \\ Codom(t) \leq t_2 \end{cases}$$

This lemma allows us to deduce a typing function for the **app** operator (similar to Lemma 5.29, with a simpler proof).

Of course, we obtain, as expected, a version of Theorem 5.6 for the restricted calculus. The only change to be made is in the case $d' \in \mathbb{E}_{\mathbf{fun}} D$ in the proof of Lemma 5.11. Having a type

$$t' = \bigwedge_{t_1 \to t_2 \in P} t_1 \to t_2 \wedge \bigwedge_{t_1 \to t_2 \in N} \neg(t_1 \to t_2)$$

with $t' \leq t$ and $t' \not\simeq \mathbb{0}$. We then find that:

$$t' \simeq \left( \left( \bigvee_{t_1 \to t_2 \in P} t_1 \right) \to \left( \bigwedge_{t_1 \to t_2 \in P} t_2 \right) \right) \wedge \bigwedge_{t_1 \to t_2 \in N} \neg(t_1 \to t_2)$$

Then we take a closed well-typed abstraction whose unique arrow type in the interface is $(\bigvee_{t_1 \to t_2 \in P} t_1) \to (\bigwedge_{t_1 \to t_2 \in P} t_2)$, for example $\mu f(\ldots).\lambda x.\ f\ x$. Rule (*abstr*) gives the type $t'$ to this value.

# Chapter 6

# Pattern matching

We will extend the calculus introduced in the previous chapter with a *pattern matching* operation. A pattern allows one or more sub-values of an input value to be extracted and named. Patterns are recursive objects, just like types, making it possible to express extractions of unbounded depths.

## 6.1 Patterns

We are going to modify the formalism from Chapter 2 to define the pattern algebra. So we have to define a set-theoretic endofunctor $G$. For any set $Y$, we define $GY$ as the set of terms generated by the following productions:

$$\begin{array}{rcll} p \in GY & ::= & x & x \text{ variable} \\ & | & t & t \in \widehat{T} \\ & | & (q_1, q_2) & q_1, q_2 \in Y \\ & | & p_1 | p_2 & p_1, p_2 \in GY \\ & | & p_1 \& p_2 & p_1, p_2 \in GY \\ & | & (x := c) & x \text{ variable}, c \in \mathcal{C} \end{array}$$

We choose a recursive and regular $G$-coalgebra $\mathbb{P} = (P, \sigma)$. The elements of $P$ (resp. $\widehat{P} = GP$) are called pattern nodes (resp. pattern expressions, or just patterns).

The productions in the definition of the functor $G$ are introducing patterns called respectively variable, type check (ou just type), product, disjunction (or union), conjunction (or intersection), and constant. Note that the components of the product patterns are pattern nodes, not patterns, and that they are the only possible use of pattern nodes to form patterns: this allows for the introduction of recursive patterns, while ensuring that the recursion loops are well formed (i.e., they cross paths with a product pattern constructor).

**Definition 6.1** *A set $S$ of patterns is* **stable** *if:*
- $\forall (q_1, q_2) \in S.\ \sigma(q_1) \in S, \sigma(q_2) \in S$
- $\forall p_1 | p_2 \in S.\ p_1 \in S, p_2 \in S$
- $\forall p_1 \& p_2 \in S.\ p_1 \in S, p_2 \in S$

**Definition 6.2** *Let $p_1$ and $p_2$ be two patterns. We say that $p_2$ is* **accessible from** *$p_1$ if $p_2$ is in any set of stable patterns that contains $p_1$.*

**Lemma 6.3** *Let $p$ be a pattern. The set of patterns accessible from $p$ is finite and stable.*

> *Proof*: This is a consequence of the regularity of the coalgebra $\mathbb{P}$.
> □

**Remark 6.4** *The concept of finite stable set plays a similar role for patterns than the one of socle for types, that is, it bounds by a finite set the set of possible objects manipulated by an algorithm, ensuring its termination.*

**Definition 6.5** *Let $p$ be a pattern. We denote by $Var(p)$ the* **set of variables** *of $p$, which is defined by:*

$$Var(p) = \{x \mid x \text{ accessible from } p\} \cup \{x \mid (x := c) \text{ accessible from } p\}$$

**Definition 6.6** *A pattern $p$ is* **well-formed** *if:*
— *for any pattern $p_1|p_2$ accessible from $p$: $Var(p_1) = Var(p_2)$;*
— *for any pattern $p_1\&p_2$ accessible from $p$: $Var(p_1) \cap Var(p_2) = \emptyset$.*

Note that there is no well-formedness condition for the sets of variables $\mathrm{Var}(\sigma(q_1))$ and $\mathrm{Var}(\sigma(q_2))$ in a pattern $(q_1, q_2)$.

## 6.2 Semantics

To define the semantics of patterns, we use a *structural* model $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ (Definition 4.4). For any element $d$ of $D$ and any pattern $p$, we write $d/p$ the result of $p$ applied to $d$; it is either a function $\gamma$ from $\mathrm{Var}(p)$ to $D$, or $\Omega$ (which represents the failure to pattern match $d$ with $p$). We use the letter $r$ to denote these results. Figure 6.1 gives a definition of $d/p$, by induction on the pair $(d, p)$ (using the lexicographic order).

The same capture variable $x$ can appear in the two pattern nodes of a product pattern $(q_1, q_2)$. If that pattern succeeds for some element $(d_1, d_2)$, we obtain the result $d'_1$ for $x$ by $q_1$ applied to $d_1$ and another result $d'_2$ for $x$ by $q_2$ applied to $d_2$. According to the definition of the operator $\oplus$, the semantics then consists in build the result $(d'_1, d'_2)$ for $x$. This semantics will allow us to express the capture of sub-sequences for an encoding of sequences using nested pairs (Section 10.3).

## 6.3 Typing

To type a well-formed pattern $p$, we have to compute on the one hand the part of the type accepted by $p$ (which represents the set of values that do not make $p$ fail), and on the other hand the type of the result of $p$ for an input type $t$ (an environment that associates to each capture variable $x \in \mathrm{Var}(p)$ a type that represents the set of values that can be obtained by applying $p$ to a value of $t$).

**Definition 6.7** *For $p$, we denote by $(\!|p|\!)$ the set of elements of the model accepted by $p$:*

$$(\!|p|\!) = \{d \in D \mid v/p \neq \Omega\}$$

$$
\begin{array}{lcll}
d/x & = & \{x \mapsto d\} & \\
d/t & = & \begin{cases} \{\} & \text{if } d \in [\![t]\!] \\ \Omega & \text{if } d \in [\![\neg t]\!] \end{cases} & \\
d/(q_1, q_2) & = & \begin{cases} d_1/\sigma(q_1) \oplus d_2/\sigma(q_2) & \text{if } d = (d_1, d_2) \\ \Omega & \text{otherwise} \end{cases} & \\
d/p_1 | p_2 & = & d/p_1 \mid d/p_2 & \\
d/p_1 \& p_2 & = & d/p_1 \oplus d/p_2 & \\
d/(x := c) & = & \{x \mapsto c\} &
\end{array}
$$

where:

$$
\begin{array}{lcl}
\gamma|r & = & \gamma \\
\Omega|r & = & r \\
\Omega \oplus r & = & \Omega \\
r \oplus \Omega & = & \Omega \\
\gamma_1 \oplus \gamma_2 & = & \{x \mapsto \gamma_1(x) \mid x \in \mathrm{Dom}(\gamma_1) \backslash \mathrm{Dom}(\gamma_2)\} \\
& \cup & \{x \mapsto \gamma_2(x) \mid x \in \mathrm{Dom}(\gamma_2) \backslash \mathrm{Dom}(\gamma_1)\} \\
& \cup & \{x \mapsto (\gamma_1(x), \gamma_2(x)) \mid x \in \mathrm{Dom}(\gamma_1) \cap \mathrm{Dom}(\gamma_2)\}
\end{array}
$$

Figure 6.1: Semantics of pattern matching

**Lemma 6.8** *We have the following equalities:*

$$
\begin{array}{lcl}
(\!|x|\!) & = & D \\
(\!|t|\!) & = & [\![t]\!] \\
(\!|(q_1, q_2)|\!) & = & (\!|\sigma(q_1)|\!) \times (\!|\sigma(q_2)|\!) \\
(\!|p_1 | p_2|\!) & = & (\!|p_1|\!) \cup (\!|p_2|\!) \\
(\!|p_1 \& p_2|\!) & = & (\!|p_1|\!) \cap (\!|p_2|\!) \\
(\!|(x := c)|\!) & = & D
\end{array}
$$

*Proof*: It is immediate by using the definition from Figure 6.1. □

**Theorem 6.9** *For every pattern $p$, we can compute a type $\lfloor p \rfloor$, independent from the model, such that:*

$$[\![\lfloor p \rfloor]\!] = (\!|p|\!)$$

*Proof*: In fact, we will build a function $\lfloor \_ \rfloor$ from pattern to types, such that:

$$
\begin{array}{lcl}
\lfloor x \rfloor & = & \mathbb{1} \\
\lfloor t \rfloor & = & t \\
\lfloor (q_1, q_2) \rfloor & = & \lfloor \sigma(q_1) \rfloor \times \lfloor \sigma(q_2) \rfloor \\
\lfloor p_1 | p_2 \rfloor & = & \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor \\
\lfloor p_1 \& p_2 \rfloor & = & \lfloor p_1 \rfloor \wedge \lfloor p_2 \rfloor \\
\lfloor (x := c) \rfloor & = & \mathbb{1}
\end{array}
$$

If $\lfloor \_ \rfloor$ verifies those equations, then $[\![\lfloor \_ \rfloor]\!]$ easily verifies the same equations as $(\!|\_|\!)$ in the next lemma. An induction on the (lexicographically ordered) pair $(d, p)$ then shows that $d \in [\![\lfloor p \rfloor]\!] \iff d \in (\!|p|\!)$ for all $d \in D$ and all pattern $p$.

To build the function $\wr\_\int$, we will use the technique developed in Section 2.4.3. We have to define a transfer between the signature $G$ of patterns, and the signature $\mathcal{B} \circ F$ of types. To do that, we have seen that we can just define a natural transformation $\beta : G \to \mathcal{B} \circ F(T + \_)$. For some $Y$, and $p \in GY$, we define $\beta_Y(p)$ by induction over $p$:

$$\begin{array}{lcl}
\beta_Y(x) & = & \mathbb{1} \\
\beta_Y(t) & = & t \\
\beta_Y((q_1, q_2)) & = & q_1 \boldsymbol{\times} q_2 \\
\beta_Y(p_1 \boldsymbol{|} p_2) & = & \beta_Y(p_1) \boldsymbol{\vee} \beta_Y(p_2) \\
\beta_Y(p_1 \boldsymbol{\&} p_2) & = & \beta_Y(p_1) \boldsymbol{\wedge} \beta_Y(p_2) \\
\beta_Y((x := c)) & = & \mathbb{1}
\end{array}$$

The right hand sides of those equations are indeed in $\mathcal{B} \circ F(T + Y)$ (by using the implicit inclusion $@ : F(Y) \to \mathcal{B} \circ F(Y)$). We can then apply Theorem 2.35, and we obtain a function $\wr\_\int$ that verifies the expected equations. □

**Definition 6.10** *Let $p$ be a well-formed pattern, $t$ a type such that $t \leq \wr p \int$, and $x$ a variable of $p$. We write:*

$$(t /\!\!/ p)(x) := \{(d/p)(x) \mid d \in [\![t]\!]\}$$

**Lemma 6.11** *In the following equations, when the left-hand side is well-defined, then so is the right-hand side, and these equalities are true:*

$$\begin{array}{lcl}
(t /\!\!/ x)(x) & = & [\![t]\!] \\
(t /\!\!/ (q_1, q_2))(x) & = & \begin{cases}
(\pi_1[t] /\!\!/ \sigma(q_1))(x) & \text{if } x \in Var(\sigma(q_1)) \setminus Var(\sigma(q_2)) \\
(\pi_2[t] /\!\!/ \sigma(q_2))(x) & \text{if } x \in Var(\sigma(q_2)) \setminus Var(\sigma(q_1)) \\
\displaystyle\bigcup_{(t_1,t_2) \in \pi(t)} (t_1 /\!\!/ \sigma(q_1))(x) \times (t_2 /\!\!/ \sigma(q_2))(x) & \\
\qquad \text{if } x \in Var(\sigma(q_2)) \cap Var(\sigma(q_1))
\end{cases} \\
(t /\!\!/ (p_1 \boldsymbol{|} p_2))(x) & = & ((t \boldsymbol{\wedge} \wr\, p_1 \int) /\!\!/ p_1)(x) \cup ((t \boldsymbol{\setminus} \wr\, p_1 \int) /\!\!/ p_2)(x) \\
(t /\!\!/ (p_1 \boldsymbol{\&} p_2))(x) & = & \begin{cases}
(t /\!\!/ p_1)(x) & \text{if } x \in Var(p_1) \\
(t /\!\!/ p_2)(x) & \text{if } x \in Var(p_2)
\end{cases} \\
(t /\!\!/ (x := c))(x) & = & \begin{cases}
\{c\} & \text{if } t \not\simeq \mathbb{0} \\
\mathbb{0} & \text{if } t \simeq \mathbb{0}
\end{cases}
\end{array}$$

Recall that the operator on types $\pi_1[\_]$ was defined by:

$$\pi_1[t] = \bigvee_{(t_1,t_2) \in \pi(t)} t_1$$

The operator $\pi_2[\_]$ is defined in a similar way.

**Theorem 6.12** *Let $p$ be a well-formed pattern and $t$ a type such that $t \leq \wr p \int$. Then we can compute a typing environment $(t/p) : Var(p) \to \widehat{T}$, which only depends on the equivalence class of the model (that is, on the induced subtyping relation) such that:*

$$\forall x \in Var(p).\ [\![(t/p)(x)]\!] = (t /\!\!/ p)(x)$$

*Proof*: Let us fix $x$. Let $S$ be the (finite) set of patterns accessible from $p$ and of which $x$ is a variable. Let $\sqsupseteq$ be a socle that contains $t$, as well as any $\wr p' \int$ for $p' \in S$. Writing:

$$V = \{[t', p'] \mid t' \in \sqsupseteq, p' \in S, t' \leq \wr p' \int\}$$

(we just use $[t', p']$ to denote the pair $(t', p')$ in order to avoid encumbering the equations with many identical parentheses). Let us define a system $\vee\!\times$ (Definition 4.47), that is, a function $\phi : V \to \mathcal{P}_f(V + V \times V + \widehat{T})$:

$$\begin{array}{lcl}
\phi([t', x]) & = & \{t'\} \\
\phi([t', (q_1, q_2)]) & = & \left\{ \begin{array}{l} \{[\pi_1[t'], \sigma(q_1)]\} \\ \qquad \text{if } x \in \mathrm{Var}(\sigma(q_1)) \backslash \mathrm{Var}(\sigma(q_2)) \\ \{[\pi_2[t'], \sigma(q_2)]\} \\ \qquad \text{if } x \in \mathrm{Var}(\sigma(q_2)) \backslash \mathrm{Var}(\sigma(q_1)) \\ \{([t_1, \sigma(q_1)], [t_2, \sigma(q_2)]) \mid (t_1, t_2) \in \pi(t')\} \\ \qquad \text{if } x \in \mathrm{Var}(\sigma(q_2)) \cap \mathrm{Var}(\sigma(q_1)) \end{array} \right. \\
\phi([t', p_1 | p_2]) & = & \{[t' \wedge \wr p_1 \int, p_1]; [t' \setminus \wr p_1 \int, p_2]\} \\
\phi([t', p_1 \& p_2]) & = & \left\{ \begin{array}{ll} \{[t', p_1]\} & \text{if } x \in \mathrm{Var}(p_1) \\ \{[t', p_2]\} & \text{if } x \in \mathrm{Var}(p_2) \end{array} \right. \\
\phi([t', (x := c)]) & = & \left\{ \begin{array}{ll} \{b_c\} & \text{if } t' \not\simeq \mathbb{0} \\ \emptyset & \text{if } t' \simeq \mathbb{0} \end{array} \right.
\end{array}$$

We see that this system only depends on the equivalence class of the model (because of the definition of $V$, and also because of the operators $\pi$, $\pi_1$, $\pi_2$, it is not completely independent from the model).

According to the previous lemma, if we write $\rho([t', p']) = (t' /\!\!/ p')(x)$, then $\rho$ is a fixed point of the operator $\widehat{\phi}$ (Definition 4.48). We will show that it is its smallest fixed point. We have to show that for any fixed point $\rho_0$, we have:

$$\forall [t', p'] \in V.\ (t' /\!\!/ p')(x) \subseteq \rho_0([t', p'])$$

This is written:

$$\forall [t', p'] \in V.\ \forall d \in [\![t']\!].\ (d/p')(x) \in \rho_0([t', p'])$$

This property is proved by induction on the pair $(d, p')$ (simultaneously for every $t'$), by discriminating over the form $p'$. Consider for example the case $p' = (q_1, q_2)$, with $x \in \mathrm{Var}(\sigma(q_1)) \backslash \mathrm{Var}(\sigma(q_2))$. Since $d \in [\![t']\!]$ and $t' \leq \wr p' \int \simeq \wr \sigma(q_1) \int \times \wr\, \sigma(q_2) \int$, the element $d$ is necessarily a pair $(d_1, d_2)$. We have:

$$(d/p')(x) = (d_1/\sigma(q_1))(x)$$

and

$$\rho_0([t', p']) = \widehat{\phi}(\rho_0)([t', p']) = \rho_0([\pi_1[t'], \sigma(q_1)])$$

Now $d_1 \in [\![\pi_1[t']]\!]$, and $d_1$ is strictly smaller than $d$, so we can apply the induction hypothesis to the pair $(d_1, \sigma(q_1))$; giving

$(d_1/\sigma(q_1))(x) \in \rho_0([\pi_1[t'], \sigma(q_1)])$, which concludes. All the other cases are dealt with in a similar way, by using the semantics of pattern matching and the fact that $\rho_0$ is a fixed point of $\widehat{\phi}$.

Theorem 4.49 allows us to build types that represent the smallest fixed point of the system given by $\phi$. These types are denoted by $(t'/p')(x)$, and they verify the property expressed in the statement of the theorem. □

**Remark 6.13** *In practice, the system of equations* $\vee\times$ *introduced in the proof of the theorem is build lazily: we only consider the effectively accessible pairs* $[t', p']$*. We can also use some simplifications that are justified by the set-theoretic definitions, for example by taking* $\emptyset$ *for* $\phi([t', p'])$ *when* $t' \simeq \mathbb{0}$ *(because then* $(t'/\!\!/p')(x) = \emptyset$*).*

**Lemma 6.14** *Let* $p$ *be a well-formed pattern and* $x$ *a variable of* $p$*. Then the function* $t \mapsto (t/p)(x)$ *is increasing and commutes with the operator* $\vee$*:*

$$\forall t_1, t_2 \leq \lfloor p \rfloor . \ (t_1 \vee t_2/p)(x) \simeq (t_1/p)(x) \vee (t_2/p)(x)$$

*Proof*: It is immediate, in view of the semantical definition of $(t/\!\!/p)(x)$. □

## 6.4 Extension of the calculus

We can now extend the calculus introduced in Chapter 5 with a pattern matching operation.

We add a construction in the syntax of expressions:

$$\begin{array}{rcl} e & ::= & \ldots \\ & | & \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \end{array}$$

where $p_1$ and $p_2$ denote well-formed patterns. The variables $p_i$ are considered to be bound in $e_i$. We then define $\mathrm{fv}(\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2)$ as

$$\mathrm{fv}(e) \cup (\mathrm{fv}(e_1) \backslash \mathrm{Var}(p_1)) \cup (\mathrm{fv}(e_2) \backslash \mathrm{Var}(p_2))$$

Intuitively, this expression behaves in the following way: the expression $e$ is evaluated into a value $v$. If the pattern $p_1$ accepts this value, then the expression $e_1$ is evaluated, in the environment resulting from the matching of $v$ with $p_1$. Otherwise, the expression $e_2$ is evaluated, in the environment resulting from the matching of $v$ by $p_2$, which has to succeed. It will be the role of the type system to guarantee this property (pattern exhaustivity), as well as to compute the type of variables bound by the pattern $p_i$ in $e_i$.

This new construction is in fact a generalization of the dynamic type check operation. Indeed, we can encode the expression $(x = e \in t\ ?\ e_1 | e_2)$ with $\texttt{match}\ e\ \texttt{with}\ (x\&t) \to e_1 \mid (x\&\neg t) \to e_2$.

**Typing** We take, as a first model, the same structural model used so far in this chapter.

The typing rule associated with the new construction is:

$$\frac{\left\{\begin{array}{l}\Gamma \vdash e : t_0 \\ t_0 \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor \\ t_1 = t_0 \wedge \lfloor p_1 \rfloor \\ t_2 = t_0 \setminus \lfloor p_1 \rfloor\end{array}\right. \quad \forall i = 1..2.\ \left\{\begin{array}{ll} s_i = \mathbb{0} & \text{if } t_i \simeq \mathbb{0} \\ (t_i/p_i), \Gamma \vdash e_i : s_i & \text{if } t_i \not\simeq \mathbb{0}\end{array}\right.}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 : s_1 \vee s_2} \ (match)$$

Every syntactic property of the type system can be extended without problem, in particular Theorem 5.6.

**Semantics** We have the structural model of values $[\![\_]\!]_{\mathcal{V}} : \widehat{T} \to \mathcal{P}(\mathcal{V})$. For any value $v$ and any well-formed pattern $p$, we can compute $v/p$, which is either $\Omega$, or a function from $\mathrm{Var}(p)$ to $\mathcal{V}$ (that we see as a substitution).

This allows us to define the operational semantics of pattern matching. We extend the evaluation contexts to reduce pattern matched expressions:

$$\begin{array}{rcl} C[] & ::= & \ldots \\ & \mid & \texttt{match}\ []\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \end{array}$$

And we add the following reduction rules:

$$\frac{v/p_1 \neq \Omega}{\texttt{match}\ v\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \rightsquigarrow e_1[v/p_1]}$$

$$\frac{v/p_1 = \Omega \quad v/p_2 \neq \Omega}{\texttt{match}\ v\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \rightsquigarrow e_2[v/p_2]}$$

$$\frac{v/p_1 = \Omega \quad v/p_2 = \Omega}{\texttt{match}\ v\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \rightsquigarrow \Omega}$$

**Safety** The value model being equivalent to the original model, we can give an interpretation of the pattern typing operators in terms of values.

**Lemma 6.15** *Let $p$ be a well-formed pattern and $v$ a value. Then:*

$$v/p \neq \Omega \iff \vdash v : \lfloor p \rfloor$$

**Lemma 6.16** *Let $p$ be a well-formed pattern, $t$ a type such that $t \leq \lfloor p \rfloor$, $x$ a variable of $p$, and $v$ a value. Then:*

$$(\exists v'.\ (\vdash v' : t) \wedge ((v'/p)(x) = v)) \iff \vdash v : (t/p)(x)$$

From this, we can easily infer a type preservation through reduction theorem.

**Remark 6.17** *At this point, one might wonder why we did not directly choose to work in the value model to define the pattern typing operators, which would have had the effect of replacing the two lemmas above with definitions. There is a simple reason for that: the definition of the value model rests of the definition of the type system, which uses those typing operators. We can also note that the value model of Chapter 5 is not identical to the one of the calculus studied in this chapter, because the values are not the same (in the body of an abstraction, a value can use the new pattern matching construction).*

**Type inference** The approach of Section 5.7 can smoothly be extended to deal with the new construction.

**Lemma 6.18** *For every pattern $p$ and every schema $\mathbb{t}$ such that $\mathbb{t} \leq \lbag p \rbag$, and every variables $x$ of $p$, we can compute a schema $((\mathbb{t}/p))(x)$ such that:*

$$\forall s.\ (((\mathbb{t}/p))(x) \leq s \iff \exists t.\ (\mathbb{t} \leq t) \wedge ((t/p))(x) \leq s)$$

*Proof*: We define $(\mathbb{t}/p)(x)$ for any pattern $\mathbb{t}$ such that $\mathbb{t} \leq \lbag p \rbag$ by induction on the lexicographically ordered triplet $(\#\mathbb{t}, p, \mathbb{t})$, where $\#\mathbb{t}$ represents the maximal number of constructors $\otimes$ nested in $\mathbb{t}$. Let us remember that, according to Lemma 5.26, we have $\#(t_0 \oslash \mathbb{t}) \leq \#\mathbb{t}$. We then let ourselves be guided by Lemma 6.11. The equations below deal with all cases, and they are well-founded for induction on the triplet $(\#\mathbb{t}, p, \mathbb{t})$.

$$\begin{array}{lcl}
(\mathbb{t}/x)(x) & = & \mathbb{t} \\
(t/(q_1, q_2))(x) & = & (t/(q_1, q_2))(x) \text{ (seen as a schema)} \\
((\mathbb{t}_1 \oslash \mathbb{t}_2)/(q_1, q_2))(x) & = & (\mathbb{t}_1/(q_1, q_2))(x) \oslash (\mathbb{t}_2/(q_1, q_2))(x) \\
((\mathbb{t}_1 \otimes \mathbb{t}_2)/(q_1, q_2))(x) & = & \left\{ \begin{array}{l} (\mathbb{t}_1/\sigma(q_1))(x) \\ \quad \text{if } x \in \mathrm{Var}(\sigma(q_1)) \backslash \mathrm{Var}(\sigma(q_2)) \\ (\mathbb{t}_2/\sigma(q_2))(x) \\ \quad \text{if } x \in \mathrm{Var}(\sigma(q_2)) \backslash \mathrm{Var}(\sigma(q_1)) \\ (\mathbb{t}_1/\sigma(q_1))(x) \otimes (\mathbb{t}_2/\sigma(q_2))(x) \\ \quad \text{if } x \in \mathrm{Var}(\sigma(q_1)) \cap \mathrm{Var}(\sigma(q_2)) \end{array} \right. \\
(\mathbb{t}/p_1|p_2)(x) & = & ((\lbag p_1 \rbag \oslash \mathbb{t})/p_1)(x) \oslash \\
 & & \quad (((\neg \lbag p_1 \rbag) \oslash \mathbb{t})/p_2)(x) \\
(\mathbb{t}/p_1 \& p_2)(x) & = & \left\{ \begin{array}{ll} (\mathbb{t}/p_1)(x) & \text{if } x \in \mathrm{Var}(p_1) \\ (\mathbb{t}/p_2)(x) & \text{if } x \in \mathrm{Var}(p_2) \end{array} \right. \\
(\mathbb{t}/(x := c))(x) & = & \left\{ \begin{array}{ll} b_c & \text{if } \mathbb{t} \not\leq \mathbb{0} \\ \mathbb{0} & \text{if } \mathbb{t} \leq \mathbb{0} \end{array} \right.
\end{array}$$

It is then easily established that each case of this inductive definition preserves the property of the statement. Consider for example the case $p = p_1|p_2$. Let $s$ be a type. We have:

$$\begin{array}{ll}
 & (\mathbb{t}/p_1|p_2)(x) \leq s \\
\iff & \left\{ \begin{array}{l} ((\lbag p_1 \rbag \oslash \mathbb{t})/p_1)(x) \leq s \\ (((\neg \lbag p_1 \rbag) \oslash \mathbb{t})/p_2)(x) \leq s \end{array} \right. \\
\iff & \left\{ \begin{array}{l} \exists t_1 \geq \lbag p_1 \rbag \oslash \mathbb{t}.\ (t_1/p_1)(x) \leq s \\ \exists t_2 \geq (\neg \lbag p_1 \rbag) \oslash \mathbb{t}.\ (t_2/p_2)(x) \leq s \end{array} \right. \\
\iff & \exists t_1, t_2 \geq \mathbb{t}.\ \left\{ \begin{array}{l} ((\lbag p_1 \rbag \wedge t_1)/p_1)(x) \leq s \\ (((\neg \lbag p_1 \rbag) \wedge t_2)/p_2)(x) \leq s \end{array} \right. \\
\iff & \exists t \geq \mathbb{t}.\ (t/p_1|p_2)(x) \leq s
\end{array}$$

For the last equivalence, we take $t = t_1 \wedge t_2$ for implication $\Rightarrow$ and $t_1 = t_2 = t$ for implication $\Leftarrow$. The other cases are dealt with in a similar way. □

We can then define the missing case in the definition of the typing algorithm

(Figure 5.3):

$$\mathbb{\Gamma}[\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2] := \mathbb{s}_1 \oslash \mathbb{s}_2$$

$$\text{where} \begin{cases} \mathbb{t}_0 := \mathbb{\Gamma}[e] \\ \mathbb{t}_1 := \wr p_1 \int \oslash \mathbb{t}_0 \\ \mathbb{t}_2 := (\neg \wr\, p_1 \int) \oslash \mathbb{t}_0 \\ \mathbb{s}_i := \begin{cases} ((\mathbb{t}_i / p_i), \mathbb{\Gamma})[e_i] & \text{if } \mathbb{t}_i \not\leq \mathbb{0}, \mathbb{t}_i \leq \wr p_i \int \\ \mathbb{0} & \text{if } \mathbb{t}_i \leq \mathbb{0} \\ \Omega & \text{if } \mathbb{t}_i \not\leq \wr p_i \int \end{cases} & (i = 1..2) \end{cases}$$

It is easy to prove the corresponding version of Theorem 5.33.

**Pattern matching with $n$ branches** We have chosen to introduce two-branch pattern matching to simplify the notations. There is no problem with considering pattern matching with $n$ branches (with $n \geq 1$). In fact, we can encode pattern matching with $n$ branches in two-branch pattern matching without losing precision in terms of typing. For $n = 1$, let us define:

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 := \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_1 \to e_1$$

And for $n \geq 3$, we define:

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots := \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid x \to \texttt{match}\ x\ \texttt{with}\ \ldots$$

($x$ is a fresh variable)

## 6.5 Normalization

In this section, we show how the categorical approach used to define the $G$-coalgebra $\mathbb{P} = (P, \sigma)$ allows us to describe pattern transformations that preserve their semantics. In this section, we consider only well-formed patterns.

**Definition 6.19** *If $p$ and $p'$ are two patterns, we write $p \simeq p'$ if $Var(p) = Var(p')$ and $v/p = v/p'$ for any value $v$. If $q$ and $q'$ are two pattern nodes, we write $q \simeq q'$ for $\sigma(q) \simeq \sigma(q')$.*

### 6.5.1 Method

We consider a normalization specification given by a restriction of $G$, that is, a set-theoretic endofunctor $G'$ such that, for all set $X$, we have $G'X \subseteq GX$, and for all function $f : X \to Y$, $G'f$ is the restriction of $Gf$. We also assume that if $X \subseteq Y$, then $GX \cap G'Y \subseteq G'X$. Intuitively, the functor $G'$ imposes certain constraints on the structure of the patterns. Of course, these constraints have to apply recursively to the $\sigma(q)$ for any pattern node $q$ that appears in the patterns considered.

For any $G$-coalgebra $\mathbb{P}' = (P', \sigma')$, we call $G'$-basis a finite part $B \subseteq P'$ such that $\sigma'(B) \subseteq G'B$ (it is, also, a basis). The patterns that are elements of $G'B$ for a certain $G'$-basis of $\mathbb{P}$ are called $G$-patterns. We are trying to compute for any pattern $p$ a $G'$-pattern $\phi(p)$ such that $p \simeq \phi(p)$. This equivalence is proved in a coinductive way over patterns, making up for a cumbersome direct proof. The formalism we introduce in this section allows us to factor in this coinductive

reasoning. The next section gives an example of application: we will see that the proof is brought back to a simple induction that uses algebraic properties of the equivalence $\simeq$.

Here is how to build function $\phi$. We give ourselves a normalization function $N$ that associates to a pattern $p$ a new pattern $N(p)$ that verifies the constraints of $G'$, but which is built on nodes that are not from $P$, but from another set $P'$. Formally, $N$ is a function $GP \to G'P'$. It does the transformation that we expect on a higher level. Generally, it will be defined by induction over $p$. To be able to "come back" to $P$, we give ourselves another function $f : P' \to P$, which somehow defines the "semantics" of the elements of $P'$. The correction of this transformation $N$ is expressed by:

$$\forall p \in \widehat{P}.\ Gf \circ N(p) \simeq p$$

and this property is proven by induction over $p$.

We then take $\sigma' = N \circ \sigma \circ f$, which defines a $G$-coalgebra $\mathbb{P}' = (P', \sigma')$. We assume that this coalgebra is regular. Intuitively, this means that if we iterate function $N$, we will only see a finite number of different nodes from $P'$.

This regularity assumption ensures (Theorem 2.33) the existence of a morphism of $G$-coalgebras $\Phi : \mathbb{P}' \to \mathbb{P}$. We then take $\phi = G\Phi \circ N : GP \to GP$.

We prove that for any pattern $p$, $\phi(p)$ is a $G'$-pattern and $\phi(p) \simeq p$, as expected.

First of all, let us see that $\phi(p)$ is a $G'$-pattern. Any basis $B$ of $\mathbb{P}'$ is a $G'$-basis (because $\sigma'(B) \subseteq GB$ and $\sigma'(B) = N(\sigma \circ f(B)) \subseteq G'P'$, and $GB \cap G'P' \subseteq G'B$ according to the assumption over $G'$), and it is sent by the morphism $\Phi$ over a $G'$-basis of $\mathbb{P}$ (because $\sigma(\Phi(B)) = G\Phi(\sigma'(B)) \subseteq G\Phi(G'B) \subseteq G'(\Phi(B))$). Thus, all $G\Phi(p')$ for $p' \in G'P'$ are $G'$-patterns, and $\phi(p)$ is indeed of that form (with $p' = N(p)$).

It remains to be established that $\phi(p) \simeq p$ for any pattern $p$, that is:

$$\forall v. \forall p.\ v/(G\Phi \circ N(p)) = v/p$$

We prove this assertion by induction over $v$. Let $v$ be a value. By assumption, we know that $v/p = v/(Gf \circ N(p))$. We then have to prove that for all $p' \in G'P'$: $v/G\Phi(p') = v/Gf(p')$. We then reason by (local) induction over $p'$. The functoriality of $G$ and the compositionality of the pattern matching semantics allow us to easily deal with every case except the one of a pattern pair $p' = (q'_1, q'_2)$ with $q'_i \in P'$. We have $G\Phi(p') = (\Phi(q'_1), \Phi(q'_2))$ and $Gf(p') = (f(q'_1), f(q'_2))$. If the value $v$ is not a pair, then the two members are $\Omega$. If $v = (v_1, v_2)$, we only need to see that $v_i/\sigma(\Phi(q'_i)) = v_i/\sigma(f(q'_i))$ for $i = 1, 2$. Now $\sigma(\Phi(q'_i)) = G\Phi \circ \sigma'(q'_i)$ ($\Phi$ is a morphism of $G$-coalgebras) and $\sigma'(q'_i) = N \circ \sigma \circ f(q'_i)$. Writing $p_i = \sigma \circ f(q'_i)$. The aim is to prove that $v_i/(G\Phi \circ N(p_i)) = v_i/p_i$, which is given by the global induction hypothesis, applied to the values $v_1$ and $v_2$.

### 6.5.2 Elimination of the intersection

We illustrate the formalism introduced above by the following example: we want to put patterns in a normal form where type checks are restricted to $t$ types with $t \mathbf{\wedge} \mathbb{1}_{\mathbf{prod}} \simeq \mathbb{0}$ (where $\mathbb{1}_{\mathbf{prod}} = \mathbb{1} \mathbf{\times} \mathbb{1}$) and where the intersection patterns **&** are eliminated (except those needed to express the capture).

Let us define the set-theoretic endofunctor $G'$ that captures these constraints. We write:

$$\begin{array}{lcll} p \in G'Y & ::= & x\&p & \\ & | & (x := c)\&p & \\ & | & t & t \leq \neg\mathbb{1}_{\mathbf{prod}} \\ & | & p_1 \mid p_2 & p_1, p_2 \in GY \\ & | & (q_1, q_2) & q_1, q_2 \in Y \end{array}$$

This functor verifies the assumptions of the previous section. For the set $P'$, we take:

$$P' = \{(t, S) \mid t \in \widehat{T}, S \in \mathcal{P}_f(P)\}$$

Let $\mathfrak{q}$ denote the elements of $P'$. The function $f : P' \to P$ gives an intuitive meaning to the elements of $P'$. For any element $\mathfrak{q} = (t, S)$ with $S = \{q_1, \ldots, q_n\}$, we write $f(\mathfrak{q}) = q$ where $q$ is chosen such that $\sigma(q) = t\&\sigma(q_1)\& \ldots \&\sigma(q_n)$ (there exists such a pattern node because $\mathbb{P}$ is recursive). Writing $g = \sigma \circ f$.

We have a natural concept of intersection over $P'$; if $\mathfrak{q} = (t, S)$ and $\mathfrak{q}' = (t', S')$, we write:

$$\mathfrak{q}_1 \cap \mathfrak{q}_2 = (t_1 \wedge t_2, S_1 \cup S_2)$$

It is clear that: $g(\mathfrak{q}_1 \cap \mathfrak{q}_2) \simeq g(\mathfrak{q}_1)\&g(\mathfrak{q}_2)$.

We also define an intersection operator over $G'P'$ (when there are no rules to define $p \cap p'$, we write $p \cap p' = p' \cap p$):

$$\begin{array}{lcl} (x\&p) \cap p' & = & x\&(p \cap p') \\ ((x := c)\&p) \cap p' & = & (x := c)\&(p \cap p') \\ (p_1 \mid p_2) \cap p' & = & (p_1 \cap p') \mid (p_2 \cap p') \\ t \cap t' & = & t \wedge t' \\ t \cap ((t_1, S_1), (t_2, S_2)) & = & ((\mathbb{0}, S_1), (\mathbb{0}, S_2)) \\ (\mathfrak{q}_1, \mathfrak{q}_2) \cap (\mathfrak{q}'_1, \mathfrak{q}'_2) & = & (\mathfrak{q}_1 \cap \mathfrak{q}'_1, \mathfrak{q}_2 \cap \mathfrak{q}'_2) \end{array}$$

By unfolding these definitions, it becomes clear that if $p, p' \in G'P'$, then $G'f(p \cap p') \simeq G'f(p)\&G'f(p')$.

**Remark 6.20** *In the case $t \cap (\mathfrak{q}_1, \mathfrak{q}_2)$, we just have to compute a pattern $p$ that always fails ($\lceil p \rfloor \simeq \mathbb{0}$), but which preserves the set of variables (to preserve the well-formedness of the patterns). We could also have taken the pattern $x_1\& \ldots x_n\&\mathbb{0}$, where $\{x_i \mid i = 1..n\} = Var((\mathfrak{q}_1, \mathfrak{q}_2))$.*

We can then define the translation $N : GP \to G'P'$, by a simple induction over patterns. For any type $t$, we denote by $\mathfrak{q}(t)$ the element $(t, \emptyset) \in P'$ so that $g(\mathfrak{q}(t)) \simeq t.$. Similarly, for any pattern node $q \in P$, we denote by $\mathfrak{q}(q)$ the element $(\mathbb{1}, \{q\}) \in P'$, so that $g(\mathfrak{q}(q)) \simeq \sigma(q)$. For any type $t \leq \mathbb{1}_{\mathbf{prod}}$, we choose an order of enumeration of the elements of $\pi(t)$.

$$\begin{array}{lcl} N(t) & = & (t \backslash \mathbb{1}_{\mathbf{prod}}) \mid (\mathfrak{q}(t_1^1), \mathfrak{q}(t_2^1)) \mid \ldots \mid (\mathfrak{q}(t_1^n), \mathfrak{q}(t_2^n)) \\ & & \text{where } \pi(t \wedge \mathbb{1}_{\mathbf{prod}}) = \{(t_1^i, t_2^i) \mid i = 1..n\} \\ N(x) & = & x\&N(\mathbb{1}) \\ N((x := c)) & = & (x := c)\&N(\mathbb{1}) \\ N((q_1, q_2)) & = & (\mathfrak{q}(q_1), \mathfrak{q}(q_2)) \\ N(p_1 \mid p_2) & = & N(p_1) \mid N(p_2) \\ N(p_1\&p_2) & = & N(p_1) \cap N(p_2) \end{array}$$

An induction over $p$ immediately gives $Gf(N(p)) \simeq p$ for every $p$. To apply the construction of the previous section, all it remains is to check that the $G$-coalgebra $\mathbb{P}' = (P', N \circ \sigma \circ f)$ is regular. We just have to check that for every basis $B$ of $\mathbb{P}$, and every socle $\beth$ that contains at least $\mathbb{1}_{\mathbf{prod}}$, the subset $B'_{\beth} = \beth \times \mathcal{P}_f(B)$ is a basis $\mathbb{P}'$ (indeed, any element of $P'$ is in such a $B'_{\beth}$). Now if $\mathfrak{q} \in B'_{\beth}$, we see, according to the definition of $f$ that $g(\mathfrak{q})$ is in $GB$ and that the only type that appears within it is $\beth$. Is is then easy to prove by induction over $p$ that if $p \in GB$ and if all of the types that appear in it are in $\beth$, then $N(p) \in G'(B'_{\beth})$, which concludes. For the case $p = p_1 \& p_2$, we use the fact that $G'(B'_{\beth})$ is stable by the binary operator $\cap$.

# Part II

# Algorithmic aspects

# Chapter 7

# Subtyping algorithm

In this chapter, we study the algorithmic aspect of the subtyping relation induced by universal models (Section 4.5).

We reduce the study of subtyping to that of emptiness checking, by using the equivalence $t_1 \leq t_2 \iff [\![t_1 \backslash t_2]\!] = \emptyset$. We modularize the presentation of the subtyping algorithm by expressing the unary predicate $[\![\_]\!] \neq \emptyset$ in inductive form, that is, as the smallest fixed point of a monotonic operator, and by studying separately the computation of these kind of predicates. This modularization separates the difficulties and allows us to optimize the two stages separately. Section 7.1 introduces the underlying formalism as well as two computational algorithms, Section 7.2 shows how to deduce from the definition of universal models the definition of a monotonic operator for which the predicate $[\![\_]\!] \neq \emptyset$ is the smallest fixed point, and Section 7.3 presents variations and optimizations on the definition of that operator.

## 7.1 Computation of an inductive predicate

In this section, we work with a finite set $\supseteq$ of variables, written as $t, t', \ldots$. We write $B = \{0, 1\}$. The binary operators $\vee, \wedge$ are naturally defined over $B$, as well as the order relation $0 \leq 1$.

### 7.1.1 Formulas, systems, induction

**Definition 7.1** *A* **formula** *is a term generated by the following productions:*

$$\phi ::= t \mid 0 \mid 1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2$$

*We denote by $\mathcal{F}$ the set of formulas.*

**Definition 7.2** *A* **congruence** *is a function $\rho : \mathcal{F} \to B$ such that $\rho(0) = 0$, $\rho(1) = 1$, $\rho(\phi_1 \vee \phi_2) = \rho(\phi_1) \vee \rho(\phi_2)$, $\rho(\phi_1 \wedge \phi_2) = \rho(\phi_1) \wedge \rho(\phi_2)$. We identify congruences with the functions $\supseteq \to B$.*

**Definition 7.3** *A* **system** *$S$ is a set of constraints of the form $(\phi \Rightarrow t)$. For any congruence $\rho$, we write $\rho \vDash S$ when $\rho(\phi) \leq \rho(t)$ for any constraint $(\phi \Rightarrow t)$ of $S$.*

**Definition 7.4** *For any system $S$ and any formula $\phi$, we write $S \vDash \phi$ when:*

$$\forall \rho \vDash S.\ \rho(\phi) = 1$$

*We extend this notation to a set of formulas: $S \vDash \{\phi_1, \ldots, \phi_n\} \iff \forall i.\ S \vDash \phi_i$. If $S$ and $S'$ are two systems, we write $S \simeq S'$ for: $\forall \phi.\ S \vDash \phi \iff S' \vDash \phi$.*

We want to *compute* this judgement $S \vDash \phi$. We begin by giving an alternative presentation in axiomatic form, which, although it does not directly give us an algorithm, allows us to *observe* all the properties that will be needed over $\vDash$.

**Definition 7.5** *We define the judgement $S \vdash \phi$ by the inductive system below:*

$$\frac{S \vdash \phi \quad (\phi \Rightarrow t) \in S}{S \vdash t} \qquad \frac{}{S \vdash 1}$$

$$\frac{S \vdash \phi_1}{S \vdash \phi_1 \vee \phi_2} \qquad \frac{S \vdash \phi_2}{S \vdash \phi_1 \vee \phi_2} \qquad \frac{S \vdash \phi_1 \quad S \vdash \phi_2}{S \vdash \phi_1 \wedge \phi_2}$$

*We define a function $[\![S]\!] : \mathcal{F} \to B$ by:*

$$[\![S]\!](\phi) = 1 \iff S \vdash \phi$$

**Lemma 7.6** *The function $[\![S]\!]$ is a congruence and $[\![S]\!] \vDash S$.*

> *Proof*: To see that $[\![S]\!]$ is a congruence, we just need to inspect the rules that define the judgement $\vdash$. Let us prove that $[\![S]\!] \vDash S$. Let $(\phi \Rightarrow t)$ be a constraint of $S$. Assuming that $[\![S]\!](\phi) = 1$. That means $S \vdash \phi$. By applying a rule, we obtain $S \vdash t$, i.e. $[\![S]\!](t) = 1$. We have proven that $[\![S]\!](\phi) \leq [\![S]\!](t)$. □

**Theorem 7.7** *For any system $S$ and any formula $\phi$, we have:*

$$S \vdash \phi \iff S \vDash \phi$$

> *Proof*: Suppose $S \vDash \phi$, that is, $\forall \rho \vDash S.\ \rho(\phi) = 1$. By taking $\rho = [\![S]\!]$, we obtain $[\![S]\!](\phi) = 1$, that is, $S \vdash \phi$.
>
> The implication $S \vdash \phi \Rightarrow S \vDash \phi$ is proven by induction on the derivation of $S \vdash \phi$. Every case is easy. □

The axiomatic definition of the judgement does not directly give an algorithm; indeed, if we try to build a derivation for $S \vdash \phi$, it may be that we find the same judgement again, and that we start trying to build an infinite derivation.

**Definition 7.8** *We define the judgement $S, N \vdash \phi$, where $S$ is a system, $N$ a set of variables and $\phi$ a formula by the following inductive system:*

$$\frac{S, N \cup \{t\} \vdash \phi \quad (\phi \Rightarrow t) \in S \quad t \notin N}{S, N \vdash t} \qquad \frac{}{S, N \vdash 1}$$

$$\frac{S, N \vdash \phi_1}{S, N \vdash \phi_1 \vee \phi_2} \qquad \frac{S, N \vdash \phi_2}{S, N \vdash \phi_1 \vee \phi_2} \qquad \frac{S, N \vdash \phi_1 \quad S, N \vdash \phi_2}{S, N \vdash \phi_1 \wedge \phi_2}$$

The set $N$ "blocks" the construction of derivations by preventing the same variable from being unfolded several times on a branch.

We see that the derivations for the judgment $S, \emptyset \vdash \phi$ correspond bijectively and naturally to the derivations for $S \vdash \phi$ that do not use twice the same variable on the same branch. Now if $S \vdash \phi$, there is always such a derivation: we just have to consider a minimal size derivation. Indeed, a derivation that uses the same variable twice on the same branch can be "short-circuited" to obtain a strictly smaller derivation. Similarly, it can be seen that the derivations for the judgement $S, N \vdash \phi$ correspond to the derivations for the judgement $S_N, \emptyset \vdash \phi$, where $S_N = \{(\phi \Rightarrow t) \in S \mid t \notin N\}$.

**Theorem 7.9** *For any system $S$, any formula $\phi$ and $N \subseteq \beth$:*

$$S, N \vdash \phi \iff S_N \vdash \phi$$

*In particular,:*

$$S, \emptyset \vdash \phi \iff S \vdash \phi$$

For any finite system $S$, the definition of judgement $S, N \vdash \phi$ does give a non-deterministic algorithm (which terminates).

**Lemma 7.10** *Let $S$ be a system and $(\phi \Rightarrow t)$ be a constraint. Then:*

$$S \nVdash \phi \Rightarrow S \simeq S \cup \{(\phi \Rightarrow t)\}$$

| *Proof*: Writing $S' = S \cup \{(\phi \Rightarrow t)\}$. It is clear that $[\![S]\!] \leq [\![S']\!]$ because $S \subseteq S'$. Writing $\rho = [\![S]\!]$. We have $\rho \vDash \{(\phi \Rightarrow t)\}$ because $S \nVdash \phi$, that is, $\rho(\phi) = 0$. By combining this with $\rho \vDash S$, we obtain $\rho \vDash S'$, therefore $[\![S']\!] \leq \rho$. □

**Lemma 7.11** *Let $S$ be a system and $N \subseteq \beth$. Then:*

$$t \in N \Rightarrow S_N \nVdash t$$

| *Proof*: It is obvious with the axiomatic definition of $S_N \vdash t$. □

### 7.1.2 Cache

In this section, we present an algorithm to compute $[\![S]\!]$ that carefully stores intermediate results in order to avoid unnecessary computations. Assuming that system $S$ is given by a function $\Phi$ from variables to formulas:

$$S = \{(\Phi(t) \Rightarrow t) \mid t \in \beth\}$$

**Informal presentation** The algorithm maintains two disjoint sets of variables $N$ and $P$. The variables $t$ in $P$ are those for which the algorithm has already definitely established that $[\![S]\!](t) = 1$. The set $N$ plays a similar role to the one it played in the definition of the predicate $S, N \vdash \phi$ of the previous section: it keeps track of the variables $t$ that have already been considered, those that it is forbidden to unfold again if we want to avoid infinite recursion. When the algorithm considers a variable $t$ that is in $N$ (resp. in $P$), it immediately outputs 0 (resp. 1). Otherwise, it temporarily adds $t$ to $N$ and starts evaluating

the formula $\Phi(t)$. If the result is 0, the algorithm outputs 0 for $t$; but contrarily to the predicate $S, N \vdash \phi$ of the previous section, the algorithm then lets $t$ in $N$, as well as all the other variables that have been added to it during the computation of $\Phi(t)$. If the result of the computation of $\Phi(t)$ is 1, then the algorithm does output 1 for $t$, which it adds to $P$ and removes from $N$. But it must also remove from $N$ all the variables that were added to it during the computation of $\Phi(t)$. Indeed, these variables $t'$ are those for which the algorithm thought it had proven $[\![S]\!](t') = 0$, by wrongly assuming that $[\![S]\!](t) = 0$, which ended up being false.

**Formal presentation** Figure 7.1 defines an algorithm to compute a function $\texttt{eval}(\phi, N, P)$ where $\phi$ is a formula, $N, P \subseteq \beth$, $N \cap P = \emptyset$. The result is a triplet $(\epsilon, N', P')$, with $\epsilon \in B$, and $N', P' \subseteq \beth$, $N \subseteq N'$, $P \subseteq P'$, and $N' \cap P' = \emptyset$.

```
eval(t, N, P) :=
  if t ∈ P then (1, N, P)
  else if t ∈ N then (0, N, P)
  else match eval(Φ(t), N ∪ {t}, P) with
         | (0, N', P') → (0, N', P')
         | (1, N', P') → (1, N, P' ∪ {t})
eval(φ1 ∧ φ2, N, P) :=
  match eval(φ1, N, P) with
         | (0, N', P') → (0, N', P')
         | (1, N', P') → eval(φ2, N', P')
eval(φ1 ∨ φ2, N, P) :=
  match eval(φ1, N, P) with
         | (0, N', P') → eval(φ2, N', P')
         | (1, N', P') → (1, N', P')
eval(0, N, P) := (0, N, P)
eval(1, N, P) := (1, N, P)
```

Figure 7.1: Algorithm with cache

**Theorem 7.12 (Termination)** *The algorithm given in Figure 7.1 terminates.*

> *Proof*: By induction on the lexicographically ordered pair $(N, \phi)$, with $N' \leq N \iff N \subseteq N'$ (it is a well-founded order because $\beth$ is finite). □

**Remark 7.13** *One could think that if* $\texttt{eval}(\phi, N, P) = (1, N', P')$*, then* $N' = N$*, which would allows us to replace the return expression* $(1, N, P' \cup \{t\})$ *in the last line of the definition of* $\texttt{eval}(t, N, P)$ *with* $(1, N' \backslash \{t\}, P' \cup \{t\})$*. This is wrong because of the case* $\phi_1 \vee \phi_2$*, as the first recursive call can output* $(0, N', P')$*, with* $N \subsetneq N'$*.*

*To find back* $N$ *from* $N'$*, it is not enough to remove the* $t$ *element: it may be necessary to remove an* a priori *arbitrary number of elements.*

We write:

$$S(N,P) = \{\Phi(t) \Rightarrow t \mid t \notin N\} \cup \{1 \Rightarrow t \mid t \in P\}$$

This system captures the current invariant of the algorithm: the $\Phi(t) \Rightarrow t$ implications with $t$ in $N$ are not taken into account (because we can assume that the truth value of $t$ is 0), and the variables $t$ in $P$ are assumed to have a truth value of 1, which translates into the constraints $1 \Rightarrow t$.

**Theorem 7.14 (Invariant)** *If* $\texttt{eval}(\phi, N, P) = (\epsilon, N', P')$ *with* $\epsilon \in \{0,1\}$*, then:* $S(N,P) \simeq S(N',P')$ *and* $[\![S(N,P)]\!](\phi) = \epsilon$*.*

*Proof*: The proof is done by induction by using, of course, the same order as for the proof of termination.
Case $\phi = t$: if $t \in P$, then clearly $S(N,P) \vDash t$. If $t \in N$, then $S(N,P) \nvDash t$ by using Lemma 7.11. Now let us assume that $t \notin P \cup N$, and fix $(\epsilon, N', P') = \texttt{eval}(\Phi(t), N \cup \{t\}, P)$. By induction, we have: $S(N \cup \{t\}, P) \simeq S(N', P')$, $[\![S(N \cup \{t\}, P)]\!](\Phi(t)) = \epsilon$.

Consider first the case $\epsilon = 0$. By applying Lemma 7.10 to the system $S(N \cup \{t\}, P)$ and to the constraint $(\Phi(t) \Rightarrow t)$, we obtain $S(N \cup \{t\}, P) \simeq S(N,P)$. We deduce $S(N,P) \simeq S(N',P')$ and $S(N,P) \nvDash t$ (because $S(N \cup \{t\}, P) \nvDash t$ according to Lemma 7.11).

Now consider the case $\epsilon = 1$. As $S(N \cup \{t\}, P) \subseteq S(N,P)$, we obtain, *a fortiori*: $S(N,P) \vDash \Phi(t)$ and $S(N,P) \vDash P'$. But $(\Phi(t) \Rightarrow t) \in S(N,P)$, therefore $S(N,P) \vDash t$. Since $S(N,P) \vDash P' \cup \{t\}$, we do have $S(N,P) \simeq S(N, P' \cup \{t\})$.
Case $\phi = \phi_1 \wedge \phi_2$: let us write $(\epsilon, N', P') = \texttt{eval}(\phi_1, N, P)$. By induction, we have: $S(N',P') \simeq S(N,P)$ and $[\![S(N,P)]\!](\phi_1) = \epsilon$. If $\epsilon = 0$, then we also have $[\![S(N,P)]\!](\phi) = 0$, which concludes that case. If $\epsilon = 1$, then, by writing $(\epsilon', N'', P'') = \texttt{eval}(\phi_2, N, P)$, we obtain, by induction: $S(N'',P'') \simeq S(N',P')$ and $[\![S(N',P')]\!](\phi_2) = \epsilon'$. Hence we deduce $S(N,P) \simeq S(N'',P'')$ and $[\![S(N,P)]\!](\phi_2) = \epsilon'$, which concludes that case.
Case $\phi = \phi_1 \vee \phi_2$: similar to the previous one.
Case $\phi = 0$ or $\phi = 1$: trivial. □

By starting from $N = P = \emptyset$, we obtain an algorithm to decide if $S \vDash \phi$, for any formula $\phi$. Indeed, we clearly have $S(\emptyset, \emptyset) = S$. Furthermore, the output of the algorithm gives us two sets $P'$ and $N'$ that can be used in a later run of the algorithm (the larger these sets, the faster the algorithm terminates). Note that if $\phi = t$, then $t$ is in one of the two sets $P'$ or $N'$; this shows that if we have already used the algorithm to compute $[\![S]\!](t)$, a subsequent call with the same argument will terminate immediately.

**Implementation** We have presented a purely functional version of the algorithm. In practice, for efficiency reasons, we can implement the sets $P$ and $N$ using imperative data structures such as hash tables. For the set $P$, this is not a problem, because it only grows during a run of the algorithm. However, we have to be more cautious about set $N$. Indeed, the last case of the definition of $\texttt{eval}(t, N, P)$ outputs the initial value of $N$, and not the one obtained after the

recursive call. Hence we have to be able to backtrack to find the initial value of $N$. This can be done easily, without backing up the entire hash table: we can just keep in a stack the history of items added to $N$; then, to return to the initial value of $N$, we pop elements off the stack until we fall back on $t$, and in parallel we remove all of the popped elements from $N$ . The stack and the hash table continuously keep track of the same objects (those of $N$): the stack is used to backtrack, and the hash table is used to quickly check whether a type is in $N$.

Of course, it is possible to use the same hash table to represent both $P$ and $N$ (meaning only one access is needed for the two checks $t \in P$, $t \in N$).

**Remark 7.15** *Hosoya [Hos01] points out that to implement the subtyping algorithm of XDuce, it is not possible to use an in-place mutable structure, such as a hash table, to represent the intermediate assumptions that may have to be removed after backtracking (which correspond to our set $N$). The technique proposed above, however, allows such a data structure to be used without having to copy it entirely.*

**Optimization** In the definition of $\texttt{eval}(t, N, P)$, we can replace the constraint $\Phi(t)$ by an equivalent constraint modulo the following assumptions: to replace a variable $t'$ such that $t' \in P$ (resp. $t' \in N$) by 1 (resp. 0), and to simplify by using boolean tautologies (such that $\phi \wedge 0 \simeq 0$). For example, if $\Phi(t) = t_1 \wedge t_2$ and $t_2 \in N$, then we can replace $\Phi(t)$ by 0, which avoids "unrolling" $t_1$. The proof of correction of this optimization is easy.

**Backtracking** Outputting $N$ in the last line of case $\phi = t$ comes down to invalidating all the negative assumptions that were added to $N'$ during the recursive call. Indeed, the result $(1, N', P')$ implies that the truth value of the $t' \in N'$ is 0, *by assuming* that the one of $t$ is 0, but that is not true.

**Example 7.16** *Here is an example of that. Let $\sqsupset = \{t_1, t_2\}$, and $\Phi(t_1) = t_2 \vee 1$, $\Phi(t_2) = t_1$. We want to compute the truth value of $t_1$. The algorithm computes $\texttt{eval}(t_2 \vee 1, \{t_1\}, \emptyset) = (1, \{t_1, t_2\}, \emptyset)$, therefore $\texttt{eval}(t_1, \emptyset, \emptyset) = (1, \emptyset, \{t_1\})$. It is necessary to remove not only $t_1$, but also $t_2$ from the intermediate set $N$, because the truth value of $t_2$ is 1 (it is in $N = \{t_1, t_2\}$ because of a false assumption).*

### 7.1.3 Removing backtracking

The algorithm of the previous section, before unrolling the definition of $\Phi(t)$ when it considers a variable $t$, starts by assuming that the truth value of $t$ is 0. During the computation of $\Phi(t)$, the algorithm can establish that the truth value of another variable $t'$ is 0; as this deduction may depend from the assumption that the truth value of $t$ is 0, we have to invalidate it if that assumption turns out to be false. This is the backtracking phenomenon highlighted in Example 7.16. However, in some cases, this systematic backtracking of the $N$ set invalidates facts that do not depend on the assumption that turned out to be false, as shown in the example below.

**Example 7.17** *Let us modify Example 7.16, taking $\Phi(t_2) = 0$. We still obtain $\texttt{eval}(t_2 \vee 1, \{t_1\}, \emptyset) = (1, \{t_1, t_2\}, \emptyset)$, but the fact that $t_2$ ends up in the set*

*$N = \{t_1, t_2\}$ in output does not depend from the fact that $t_1$ was in the set $N = \{t_1\}$ in input. The algorithm of the previous section cannot distinguish this situation from that of Example 7.16, because it forgets the dependencies between assumptions (which can be invalidated), and their conclusions.*

In this section, we present a new algorithm that computes $[\![S]\!]$ while completely avoiding backtracking. The idea is that instead of assuming that a variable $t$ has a truth value 0 during the computation of $\Phi(t)$, we will keep track of the dependencies of $t$, that is, of the constraints that need to be taken into account if the truth value of $t$ is ultimately 1.

**Informal presentation** The algorithm keeps a state for each variable $t$ it has already encountered. It can be a constant 0 or 1, which means that the truth value of this variable is known for sure. It can also be a set of constraints $[C_1; \ldots; C_n]$ that have to be taken into count if we end up finding that the truth value of $t$ is 1, and that we can ignore otherwise.

The overall state of the algorithm, consisting of the states of all variables, is called the current table. When we consider a new variable $t$, we start by giving it the state [] (an empty set of constraints), and we "activate" the constraint $\Phi(t) \Rightarrow t$. In fact, we consider extended constraints, of the form $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$. Activating such a constraint with $n = 0$ comes down to setting the state of $t$ to 1 and successively activating the constraints $C_1, \ldots, C_n$ if the state of $t$ was previously $[C_1; \ldots; C_n]$. If $n > 0$, activating a constraint $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$ is done by considering the form of $\phi_1$. If $\phi_1 = 1$, we activate $\phi_2 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$. If $\phi_1 = 0$, there is nothing to do. If $\phi_1 = t'$, we start by considering $t'$; if its state 0 or 1, then we proceed as in the cases above where $\phi_1 = 0$ or $\phi_1 = 1$; if its state is a set of constraints $[C_1; \ldots; C_n]$, we add to this set an additional constraint $\phi_2 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$. Indeed, if it turns out that the truth value of $t$ is 1, we will have to check this constraint again, although for now it is temporarily suspended. Finally, if $\phi_1 = \phi \wedge \phi'$, we activate $\phi \Rightarrow \phi' \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$, and if $\phi_1 = \phi \vee \phi'$, we successively activate the two constraints $\phi \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$ and $\phi' \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$.

Once we have dealt with all constraints that needed activating, we can replace all the states of the form $[C_1; \ldots; C_n]$ by 0, because we know that the truth value of the corresponding variables cannot be 1. All the possible states for all considered variables are then 0 and 1.

Let us make a few comments, which will result in optimizations for the algorithm.

- When activating a constraint $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$, if we have already established that the truth value of $t$ is 1, then we can immediately ignore this constraint.
- Similarly, a constraint of the form $t \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$ can be ignored, because it is necessarily satisfied.
- When the algorithm considers a new variable $t$, it sets its state to the empty constraint set [] and activates the constraint $\Phi(t) \Rightarrow t$. After doing this, if the state of $t$ is a set of constraints $[C_1; \ldots; C_n]$, we can sometimes replaces it with 0, which will save us from extending this set later. In order to do so, we have to be sure that the truth value of $t$ is indeed 0; that is the case if none of the other suspended constraints in the current table is "pointing" towards $t$.

**Formal presentation** We will give a formal presentation of the algorithm described above. Let us start by formally defining the concepts of extended constraint and table.

**Definition 7.18** *A* **long constraint** *is either a variable $t$, or a term of the form $\phi \Rightarrow C$, where $\phi$ is a formula and $C$ is a long constraint. We can see a long constraint $C$ as a finite list of formulas, followed by a variable, written $v(C)$. The definitions on constraints and systems of constraints are transferred to long constraints and systems of longs constraints, by associating the long constraint $C = \phi_1 \Rightarrow \ldots\ldots \phi_n \Rightarrow v(C)$ with the constraint $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow v(C)$ (if $n = 0$, we take $1 \Rightarrow v(C)$).*

**Definition 7.19** *A* **table** *is a function $\sigma$ which associates to every variable either 0, or 1, or $\bot$, or a finite sequence of long constraints $L = [C_1; \ldots; C_n]$ such that $\sigma(v(C_i)) \notin \{0, \bot\}$.*

We denote by $\sigma\{t \to X\}$ the table $\sigma'$ such that $\sigma'(t) = X$ and $\sigma'(t') = \sigma(t')$ for $t' \neq t$.

**Definition 7.20** *The* **domain** *of a table $\sigma$ is the set $Dom(\sigma)$ of variables $t$ such that $\sigma(t) \neq \bot$. We define a preorder on tables by:*

$$\sigma \leq \sigma' \iff \begin{cases} Dom(\sigma') \subseteq Dom(\sigma) \\ \{t \mid \sigma'(t) = 1\} \subseteq \{t \mid \sigma(t) = 1\} \end{cases}$$

*This preorder is well-founded (because $\sqsupset$ is finite).*

A table with no suspended constraints and that is correct regarding the system $S$ is called a partial solution.

**Definition 7.21** *A* **partial solution** *is a table $\sigma$ such that:*

$$\forall t \in Dom(\sigma).\ \sigma(t) = [\![S]\!](t)$$

Figure 7.2 defines two mutually recursive functions `extend` and `trigger`. They both take a current table as input and output an updated table. The role of function `extend` is to "consider" a new variable $t$, that is, to ensure that the current table is defined over $t$. The `trigger` function activates a constraint; it uses an auxiliary function `trigger'` which is only called when the state of the variable associated with the constraint is not 1. The auxiliary function `may` is used to check that no suspended constraint points to a given variable. Finally, the function `clean` removes all suspended constraints and replaces the corresponding states with 0.

**Theorem 7.22** *Let $\sigma$ be a partial solution and $t$ a variable. Writing $\sigma' =$ `clean`(`extend`$(\sigma, t)$), as defined in Figure 7.2. Then $\sigma'$ is a well-defined partial solution (i.e. the algorithm terminates), with $\sigma' \leq \sigma$ and $t \in Dom(\sigma')$.*

We will give the proof of this theorem below.

By starting from the empty domain table, we obtain an algorithm to decide if $S \vDash t$ for some variable $t$. In output we get a table $\sigma'$ that can be used as input for the next call of the algorithm (global cache).

```
extend(σ, t) :=
    if σ(t) = ⊥ then
        let σ' = trigger(σ{t→[]}, Φ(t)⇒t) in
        if may(σ', t) then σ' else σ'{t→0}
    else
        σ

trigger(σ, C) := if σ(v(C)) = 1 then σ else trigger'(σ, C)
trigger'(σ, t) := let L = σ(t) in trigger*(σ{t→1}, L)
trigger'(σ, t⇒C) :=
    if v(C) = t then σ else
    let σ' = extend(σ, t) in
    match σ'(t) with
    | 0 → σ'
    | 1 → trigger(σ', C)
    | L → σ'{t→C :: L}
trigger'(σ, 0⇒C) := σ
trigger'(σ, 1⇒C) := trigger'(σ, C)
trigger'(σ, φ1∧φ2⇒C) := trigger'(σ, φ1⇒(φ2⇒C))
trigger'(σ, φ1∨φ2⇒C) := trigger(trigger'(σ, φ1⇒C), φ2⇒C)

trigger*(σ, []) := σ
trigger*(σ, C :: L) := trigger*(trigger(σ, C), L)

may(σ, t) := σ(t) = 1 ∨ ∃t'.∃C∈σ(t'). v(C) = t

clean(σ) := (t ↦ if σ(t) ∈ {0, 1, ⊥} then σ(t) else 0)
```

Figure 7.2: Algorithm without backtracking

**Implementation** As with the algorithm with cache, we have presented a functional version of this algorithm. Here, however, the table $\sigma$ is used in a purely linear way (in that sense, it is a non-backtracking algorithm). This makes it easy to implement the algorithm using an imperative data structure, such as a hash table, to represent $\sigma$. To compute function `may`, we can maintain a counter, for each variable $t$, of the number of long constraints $C \in \sigma(t')$ such that $v(C) = t$. This makes it possible to compute this function in constant time. To implement function `clean`, we simply keep a list of all the variables that have been added to the domain of $\sigma$.

**Optimizations** The optimization mentioned for the algorithm with cache can be transposed. In the definition of `extend`$(\sigma, t)$, we can simplify the constraint $\Phi(t)$ by using the following assumptions (if $\sigma(t') \in \{0, 1\}$, we can replace $t'$ by $\sigma(t')$), and boolean tautologies. Similarly, we can simplify a long constraint $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$, and also change the order of the $\phi_i$ (for example, to first consider smaller formulas).

**Examples** Let us give some examples of how the algorithm works. We show each time the tree of recursive calls between functions `trigger`, `extend` and `may`. We denote by $\{\}$ the empty table, and we name all the intermediate tables that appear in the algorithm by order of appearance.

**Example 7.23** *Taking back Example 7.17, which illustrates a useless backtracking step done by the algorithm of the previous section. We have* $\Phi(t_1) = t_2 \vee 1$, $\Phi(t_2) = 0$. *We are trying to compute the truth value of* $t_1$. *Here is how the algorithm works for the call* $\texttt{extend}(\{\}, t_1)$*:*

$$
\begin{array}{l}
\texttt{extend}(\{\}, t_1) = \sigma_4 \\
\quad \texttt{trigger}(\sigma_1, t_2 \vee 1 \Rightarrow t_1) = \sigma_4 \\
\quad\quad \texttt{trigger}(\sigma_1, t_2 \Rightarrow t_1) = \sigma_3 \\
\quad\quad\quad \texttt{extend}(\sigma_1, t_2) = \sigma_3 \\
\quad\quad\quad\quad \texttt{trigger}(\sigma_2, 0 \Rightarrow t_2) = \sigma_2 \\
\quad\quad\quad\quad \texttt{may}(\sigma_2, t_2) = \texttt{false} \\
\quad\quad \texttt{trigger}(\sigma_3, 1 \Rightarrow t_1) = \sigma_4 \\
\quad\quad\quad \texttt{trigger}(\sigma_3, t_1) = \sigma_4 \\
\quad \texttt{may}(\sigma_4, t_1) = \texttt{true}
\end{array}
$$

*where:*

$\sigma_1 = \{t_1 \to []\}$
$\sigma_2 = \{t_1 \to [], t_2 \to []\}$
$\sigma_3 = \{t_1 \to [], t_2 \to 0\}$
$\sigma_4 = \{t_1 \to 1, t_2 \to 0\}$

*The final result is* $\{t_1 \to 1, t_2 \to 0\}$ *(which is invariant by* `clean`*). When we unrolled the definition of* $\Phi(t_2)$*, we saw that there was no way to prove* $t_2$ *(by using* `may`*), therefore we were able to immediately compute its truth value* 0*.*

**Example 7.24** *Taking back Example 7.16:* $\Phi(t_1) = t_2 \vee 1$, $\Phi(t_2) = t_1$. *Here is how the algorithm works for the call* $\texttt{extend}(\{\}, t_1)$*:*

$$
\begin{array}{l}
\texttt{extend}(\{\}, t_1) = \sigma_6 \\
\quad \texttt{trigger}(\sigma_1, t_2 \vee 1 \Rightarrow t_1) = \sigma_6 \\
\quad\quad \texttt{trigger}(\sigma_1, t_2 \Rightarrow t_1) = \sigma_4 \\
\quad\quad\quad \texttt{extend}(\sigma_1, t_2) = \sigma_3 \\
\quad\quad\quad\quad \texttt{trigger}(\sigma_2, t_1 \Rightarrow t_2) = \sigma_3 \\
\quad\quad\quad\quad\quad \texttt{extend}(\sigma_2, t_1) = \sigma_2 \\
\quad\quad\quad\quad \texttt{may}(\sigma_3, t_2) = \texttt{true} \\
\quad\quad \texttt{trigger}(\sigma_4, 1 \Rightarrow t_1) = \sigma_6 \\
\quad\quad\quad \texttt{trigger}(\sigma_4, t_1) = \sigma_6 \\
\quad\quad\quad\quad \texttt{trigger}(\sigma_5, t_2) = \sigma_6 \\
\quad\quad\quad\quad\quad \texttt{trigger}(\sigma_6, t_1) = \sigma_6 \\
\quad \texttt{may}(\sigma_6, t_1) = \texttt{true}
\end{array}
$$

*where:*

$\sigma_1 = \{t_1 \to []\}$
$\sigma_2 = \{t_1 \to [], t_2 \to []\}$
$\sigma_3 = \{t_1 \to [t_2], t_2 \to []\}$
$\sigma_4 = \{t_1 \to [t_2], t_2 \to [t_1]\}$
$\sigma_5 = \{t_1 \to 1, t_2 \to [t_1]\}$
$\sigma_6 = \{t_1 \to 1, t_2 \to 1\}$

*The final result is* $\{t_1 \to 1, t_2 \to 1\}$ *(which is invariant by* `clean`*). During the calculus, the dependency* $t_1 \Rightarrow t_2$ *was stored; hence, when* $t_1$ *proved to be true (because of the* $1$ *in* $\Phi(t_1) = t_2 \vee 1$*), we immediately knew that* $t_2$ *was also true.*

**Example 7.25** *Let us give another example, with* $\Phi(t_1) = t_2$ *and* $\Phi(t_2) = t_1$*. Here is the trace of this algorithm:*

$$
\begin{array}{l}
\texttt{extend}(\{\}, t_1) = \sigma_4 \\
\quad \left| \begin{array}{l} \texttt{trigger}(\sigma_1, t_2 \Rightarrow t_1) = \sigma_4 \\
\quad \left| \begin{array}{l} \texttt{extend}(\sigma_1, t_2) = \sigma_3 \\
\quad \left| \begin{array}{l} \texttt{trigger}(\sigma_2, t_1 \Rightarrow t_2) = \sigma_3 \\
\quad \left| \texttt{extend}(\sigma_2, t_1) = \sigma_2 \right. \end{array} \right. \\
\quad \left| \texttt{may}(\sigma_3, t_2) = \texttt{true} \right. \end{array} \right. \\
\quad \left| \texttt{may}(\sigma_4, t_1) = \texttt{true} \right.
\end{array}
$$

*where:*

$\sigma_1 = \{t_1 \to []\}$
$\sigma_2 = \{t_1 \to [], t_2 \to []\}$
$\sigma_3 = \{t_1 \to [t_2], t_2 \to []\}$
$\sigma_4 = \{t_1 \to [t_2], t_2 \to [t_1]\}$

*The algorithm saves the mutual dependency between* $t_1$ *and* $t_2$*. Because there are no more "implicite" constraints (*$\kappa = \emptyset$ *for the global calls of the algorithm), we can eliminate these dependencies. This is the role of* `clean`*, that transforms table* $\{t_1 \to [t_2], t_2 \to [t_1]\}$ *into* $\{t_1 \to 0, t_2 \to 0\}$*.*

**Proof of Theorem 7.22** The rest of this section is dedicated to the proof of Theorem 7.22. We will introduce some auxiliary concepts.

**Definition 7.26** *The system of long constraints associated with a table* $\sigma$ *is defined by:*

$$
\begin{array}{rl}
S(\sigma) & = \{\Phi(t) \Rightarrow t \mid \sigma(t) = \bot\} \\
& \cup \{t \mid \sigma(t) = 1\} \\
& \cup \{t \Rightarrow C_i \mid \sigma(t) = [C_1; \ldots; C_n], i = 1..n\}
\end{array}
$$

**Definition 7.27** *A* **state** *is a pair* $(\sigma, \kappa)$ *where* $\sigma$ *is a table and* $\kappa$ *is a a set of long constraints, such that:*

$$
\left\{ \begin{array}{l} S \simeq S(\sigma) \cup \kappa \\ \forall C \in \kappa.\ \sigma(v(C)) \notin \{0, \bot\} \end{array} \right.
$$

**Lemma 7.28** *If* $(\sigma, \kappa)$ *is a state and* $\sigma(t) \in \{0, 1\}$*, then* $[\![S]\!](t) = \sigma(t)$*.*

> *Proof*: If $\sigma(t) = 1$, then the constraint $t$ is in $S(\sigma)$, therefore $[\![S]\!](t) = [\![S(\sigma) \cup \kappa]\!](t) = 1$. If $\sigma(t) = 0$, then no constraint in $S(\sigma) \cup \kappa$ points to $t$, therefore $[\![S(\sigma) \cup \kappa]\!](t) = 0$. □

In a state, the set of constraints $\kappa$ represents the constraints implicitly stored in the control flow, waiting to be "integrated" to the table $\sigma$.

Theorem 7.22 is a consequence of the two lemmas and of the theorem below.

**Lemma 7.29** *If* $\sigma$ *is a partial solution, then* $(\sigma, \emptyset)$ *is a state.*

*Proof*: We have to prove that $S \simeq S(\sigma)$. We have:

$$\begin{array}{lcl} S & = & \{\Phi(t) \Rightarrow t \mid t \in \beth\} \\ S(\sigma) & = & \{\Phi(t) \Rightarrow t \mid \sigma(t) = \bot\} \cup \{t \mid \sigma(t) = 1\} \end{array}$$

Proving first that $[\![S]\!] \vDash S(\sigma)$. The constraints of the form $\Phi(t) \Rightarrow t$ with $\sigma(t) = \bot$ are obviously satisfied under $[\![S]\!]$. If $\sigma(t) = 1$, then $[\![S]\!](t) = 1$ by definition of a partial solution, therefore the constraint $t$ is satisfied under $[\![S]\!]$. We deduce that $[\![S]\!] \vDash S(\sigma)$, hence $[\![S(\sigma)]\!] \leq [\![S]\!]$.

Proving now that $[\![S(\sigma)]\!] \vDash S$. Let $\Phi(t) \Rightarrow t$ be a constraint of $S$. If $\sigma(t) = \bot$, then the constraint is in $S(\sigma)$ and we are done. If $\sigma(t) = 1$, then $S(\sigma) \vDash t$, therefore we also have $S(\sigma) \vDash \Phi(t) \Rightarrow t$. If $\sigma(t) = 0$, then $[\![S]\!](t) = 0$, implying $[\![S]\!](\Phi(t)) = 0$. Now we have seen that $[\![S(\sigma)]\!] \leq [\![S(\sigma)]\!]$, which gives $[\![S(\sigma)]\!](\Phi(t)) = 0$, therefore $S(\sigma) \vDash \Phi(t) \Rightarrow t$. The case $\sigma(t) = [C_1; \ldots; C_n]$ is impossible because $\sigma$ is a partial solution. We have proven that $[\![S(\sigma)]\!] \vDash S$, which gives $[\![S]\!] \leq [\![S(\sigma)]\!]$. □

**Lemma 7.30** *If $(\sigma, \emptyset)$ is a state, then* `clean`$(\sigma)$ *is a partial solution $\sigma'$ with $\sigma' \leq \sigma$ (for the preorder on tables).*

*Proof*: Writing $\sigma' =$ `clean`$(\sigma)$. Consider the function $\rho : \beth \to B$ such that $\rho(t) = 1 \iff \sigma(t) \in \{1, \bot\}$. It is easy to see that $\rho \vDash S(\sigma)$: the constraints $\Phi(t) \Rightarrow t$ with $\sigma(t) = \bot$ and $t$ with $\sigma(t) = 1$ are verified because $\rho(t) = 1$, and the constraints $t \Rightarrow C_i$, with $\sigma(t) = [C_1; \ldots; C_n]$ are verified because $\rho(t) = 0$. We deduce $[\![S]\!] = [\![S(\sigma)]\!] \leq \rho$, proving that $\sigma'(t) = 0 \Rightarrow [\![S]\!](t) = 0$. If $\sigma'(t) = 1$, then $\sigma(t) = 1$, therefore $[\![S]\!] = [\![S(\sigma)]\!] \vDash t$ (since $t \in S(\sigma)$), that is, $[\![S]\!](t) = 1$.

The relation $\sigma' \leq \sigma$ is immediate. In fact, we also have $\sigma \leq \sigma'$. □

**Theorem 7.31 (Termination and invariant)** *Let $(\sigma, \kappa)$ be a state and $t \in \beth$. Then* `extend`$(\sigma, t)$ *is well-defined table $\sigma'$, with $t \in Dom(\sigma')$, $\sigma' \leq \sigma$, and such that $(\sigma', \kappa)$ is a state.*

*Let $(\sigma, \kappa \cup \{C\})$ be a state. Then* `trigger`$(\sigma, C)$ *is a well-defined table $\sigma'$, with $\sigma' \leq \sigma$, and such that $(\sigma', \kappa)$ is a state.*

*Proof*: We prove both assertions by mutual induction over $\sigma$ (by using the well-founded preorder $\leq$ on tables).

Starting by the assertion on `extend`. If $t \in \mathrm{Dom}(\sigma)$, the result is $\sigma$, and there is nothing left to prove. Otherwise, we try to apply the induction hypothesis for the call to `trigger`. The table $\sigma\{t \to []\}$ is strictly smaller than $\sigma$. In addition, we find that $(\sigma\{t \to []\}, \kappa \cup \{\Phi(t) \Rightarrow t\})$ is a state because $S(\sigma\{t \to []\}) \cup \{\Phi(t) \Rightarrow t\}) = S(\sigma)$. Hence we can apply the induction hypothesis: the call to `trigger` terminates and outputs a table $\sigma'$ such that $(\sigma', \kappa)$ is a state. If `may`$(\sigma', t)$ is true, then it is over. Otherwise, we write $\sigma'' = \sigma'\{t \to 0\}$, and we have to see that $(\sigma'', \kappa)$ is a

state. If $\sigma'(t) = 0$, there is nothing to prove. The case $\sigma'(t) = \bot$ is impossible because $\sigma' \leq \sigma\{t \to []\}$. The case $\sigma'(t) = 1$ is impossible because $\texttt{may}(\sigma', t)$ is false. The only option remaining is therefore $\sigma'(t) = [C_1; \ldots; C_n]$. No constraints of $S(\sigma') \cup \kappa$ points to $t$ (that is, $v(C) \neq t$ for any constraint $C \in S(\sigma') \cup \kappa$); indeed, if $C \in \kappa$, then $\sigma(v(C)) \notin \{0, \bot\}$ but $\sigma(t) = \bot$, and if $C \in S(\sigma')$, then we conclude by using the fact that $\texttt{may}(\sigma', t)$ is false. Hence, $[\![S(\sigma') \cup \kappa]\!](t) = 0$, and we can remove from this system any constraint of the form $t \Rightarrow C$ without changing the equivalence class for $\simeq$. Then we obtain $S(\sigma'') \cup \kappa$, which indeed gives $S(\sigma'') \cup \kappa \simeq S$. We found that $(\sigma'', \kappa)$ is a state.

Moving on to the assertion over `trigger`. We will do (for a fixed equivalence class of $\sigma$) a second (so-called local) induction over the size of the constraint $C$. We define the size of a constraint $C$ as the pair $(n, m)$ where $n$ is the number of symbols $\wedge$ appearing in $C$, and $m$ is the total number of symbols of $C$. It is easy to see that every recursive call to $\texttt{trigger}'$ from $\texttt{trigger}'$ can be replaced by a call to `trigger` without changing the result; indeed, `trigger'` is only called on pairs $(\sigma, C)$ such that $\sigma(v(C)) \neq 1$.

Let us assume then that $(\sigma, \kappa \cup \{C\})$ is a state. First consider the case $\sigma(v(C)) = 1$. We have to see that $(\sigma, \kappa)$ is also a state, i.e. that $S(\sigma) \cup \kappa \simeq S(\sigma) \cup \kappa \cup \{C\}$; this comes from the fact that $[\![S(\sigma) \cup \kappa]\!] \vDash C$ (because $v(C) \in S(\sigma)$).

From now on, we assume that $\sigma(v(C)) \neq 1$, and we proceed by case disjunction over the form of $C$.

Case $C = t$: Since $(\sigma, \kappa \cup \{C\})$ is a state and $\sigma(v(C)) \neq 1$, we necessarily have $\sigma(v(C)) = \sigma(t)$ of the form $L = [C_1; \ldots; C_n]$. We find that $S(\sigma) \cup \kappa \cup \{t\} = S(\sigma\{t \to 1\}) \cup \kappa \cup \{t \Rightarrow C_1, \ldots, t \Rightarrow C_n\} \simeq S(\sigma\{t \to 1\}) \cup \kappa \cup \{C_1, \ldots, C_n\}$. Hence, we see that $(\sigma\{t \to 1\}, \kappa \cup \{C_1, \ldots, C_n\})$ is a state, and $\sigma\{t \to 1\}$ is strictly smaller than $\sigma$. We conclude this case by applying $n$ times the global induction hypothesis (in $\texttt{trigger}^*$).

Case $C = t \Rightarrow C'$: Consider first the case where $t = v(C')$. The long constraint $C$ is then true for any congruence, hence $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa$, and $(\sigma, \kappa)$ is indeed a state.

Consider now the general case. By using the assertion over `extend`, we obtain that $(\sigma', \kappa \cup \{t \Rightarrow C'\})$ is a state, with $t \in \mathrm{Dom}(\sigma')$ and $\sigma' \leq \sigma$. Hence the pattern matching is exhaustive. If $\sigma'(t) = 0$, then $[\![S]\!](t) = [\![S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\}]\!](t) = 0$, therefore $S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\} \simeq S(\sigma') \cup \kappa$. Hence we find that $(\sigma', \kappa)$ is a state. If $\sigma'(t) = 1$, then $[\![S]\!](t) = [\![S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\}]\!](t) = 1$, therefore $S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\} \simeq S(\sigma') \cup \kappa \cup \{C'\}$. Hence we find that $(\sigma', \kappa \cup \{C'\})$ is a state. We can apply the global or local induction hypothesis, whether $\sigma'$ is strictly smaller than that $\sigma$ or not (and then $C'$ is strictly smaller than $C$). Finally, if $\sigma'(t) = [C_1; \ldots; C_n]$, then $S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\} = S(\sigma'\{t \to C' :: L\}) \cup \kappa$, therefore $(S(\sigma'\{t \to C' :: L\}), \kappa)$ is a state.

Case $C = \phi_1 \wedge \phi_2 \Rightarrow C'$: We find that $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa \cup \{\phi_1 \Rightarrow (\phi_2 \Rightarrow C')\}$. Hence $(\sigma, \kappa \cup \{\phi_1 \Rightarrow (\phi_2 \Rightarrow C')\})$ is a state.

Furthermore, the constraint $\phi_1 \Rightarrow (\phi_2 \Rightarrow C')$ is strictly smaller than $C$, with the order we have chosen over the constraints (the number of symbols $\wedge$ is strictly decreasing). We conclude this case by applying the local induction hypothesis.
Case $C = \phi_1 \vee \phi_2 \Rightarrow C'$: We find that $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa \cup \{\phi_1 \Rightarrow C', \phi_2 \Rightarrow C'\}$. Hence $(\sigma, \kappa \cup \{\phi_1 \Rightarrow C'\} \cup \{\phi_2 \Rightarrow C'\})$ is a state, and we conclude this case by applying twice the induction hypothesis (the first time, it is the local induction; the second time, it is either the local or global induction, whether $\sigma$ is strictly decreasing or not during the first call).
Case $C = 0 \Rightarrow C'$: We find that $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa$, therefore $(\sigma, \kappa)$ is a state.
Case $C = 1 \Rightarrow C'$: We find that $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa \cup \{C'\}$, therefore $(\sigma, \kappa \cup \{C'\})$ is a state. We conclude by local induction hypothesis. □

**Remark 7.32** *In the proof, we see that, as it is presented, the algorithm includes two optimizations that are in fact optional:*
- *in the definition of* `extend`*, we can output* $\sigma'$ *even though* `may` *outputs false;*
- *in the definition of* $\texttt{trigger}'(\sigma, t \Rightarrow C)$*, we can ignore the special case* $t = v(C)$*.*

### 7.1.4 Application: emptiness checking for a tree automaton

We now give a simple example of application of our inductive predicate computation algorithms. Consider binary trees, whose leafs are labeled by symbols from a finite alphabet $\Sigma$ (the nodes have no labels). A tree automaton is a pair $(Q, R)$, where $Q$ is a finite set of states, and $R$ is a finite set of transitions of the form $a \to q$ ($a \in \Sigma$, $q \in Q$) or $(q_1, q_2) \to q$ ($q, q_1, q_2 \in Q$). Each state $q$ defines in the classic way a set of regular trees, which we denote by $[\![q]\!]$. We are interested in whether $[\![q]\!] = \emptyset$ or not, for a given state $q$.

It is easy to see that the predicate $[\![q]\!] \neq \emptyset$ is *inductively* defined by:

$$[\![q]\!] \neq \emptyset \iff \begin{cases} \exists a.\ (a \to q) \in R \\ \vee \\ \exists (q_1, q_2).\ ((q_1, q_2) \to q) \in R \wedge [\![q_1]\!] \neq \emptyset \wedge [\![q_2]\!] \neq \emptyset \end{cases}$$

This fits well into the formalism introduced. We take $\beth = Q$, and:

$$\Phi(q) = \bigvee_{(a \to q) \in R} 1 \vee \bigvee_{(q_1, q_2) \to q \in R} q_1 \wedge q_2$$

If the automaton is given in a descending way (*top-down*), that is, if we can obtain for each $q$ the set of transitions with target $q$ in linear time in the size of the result, then we can prove that the non-backtracking algorithm gives a linear time algorithm (hence optimal) in relation to the size of the automaton (number of states + number of transitions). This is a known result, but it is usually achieved by a bottom-up method (by saturating the set of states $q$ such that $[\![q]\!] \neq \emptyset$); the advantage of using a top-down method is to be able to ignore

certain states: those that are inaccessible, of course, but also, for a transition $(q_1, q_2) \to q$, the state $q_2$ when we find that $[\![q_1]\!] = \emptyset$. This is particularly important when the automaton is not given explicitly and the computation of its states and transitions is costly.

## 7.2 Subtyping algorithm

Now let us see how to apply the algorithms of the previous section to compute the subtyping relation in universal models. Taking a universal model $[\![\_]\!]$, we will denote by $\leq$ the subtyping relation it induces. We obviously reduce the problem to emptiness checking, finding that $t_1 \leq t_2 \iff [\![t_1 \backslash t_2]\!] = \emptyset$. The starting point is Lemma 4.27, that can be rewritten in the following way. Let $t$ be a type and $\beth$ a socle that contains it. Then:

$$[\![t]\!] \neq \emptyset \iff \forall \mathcal{S} \subseteq \beth.\ \mathcal{S} \subseteq \mathbb{E}\mathcal{S} \Rightarrow t \notin \mathcal{S}$$

We associate to a subset $\mathcal{S} \subseteq \beth$ the function $\rho_{\mathcal{S}} : \beth \to \{0, 1\}$ defined by $\rho_{\mathcal{S}}(t) = 1 \iff t \notin \mathcal{S}$ (characteristic function of the complement). For any $t \in \beth$, we can deduce (see below) from the definition of $\mathbb{E}S$ a boolean formula $\Phi(t)$ such that:

$$\rho_{\mathbb{E}\mathcal{S}}(t) = 1 \iff \rho_S \vDash \Phi(t)$$

We then consider the system of constraints:

$$S = \{\Phi(t) \Rightarrow t \mid t \in \beth\}$$

Hence:

$$\mathcal{S} \subseteq \mathbb{E}\mathcal{S} \iff \rho_{\mathcal{S}} \vDash S$$

therefore:

$$[\![t]\!] \neq \emptyset \iff \forall \rho \vDash S.\ \rho(t) = 1$$

therefore:

$$[\![t]\!] \neq \emptyset \iff [\![S]\!](t) = 1$$

We can then use one of the algorithms presented in the previous section to compute this predicate $[\![t]\!] \neq \emptyset$. The set $\beth$ can be very large, but algorithms will generally only consider part of it. Similarly, the formulas $\Phi(t)$ are complex, but since the algorithms do not necessarily consider them entirely, it may be in our best interest to build them lazily.

Now let us explain the definition of the $\Phi(t)$ formulas. We simply follow Definition 4.11. To avoid confusion between logical operators $\vee, \wedge$ and boolean operators on types $\mathsf{V}, \mathsf{\Lambda}$, we will write $[t]$ instead of $t$ to denote the variable associated with the type $t$ (which intuitively corresponds to the predicate stating that the type $t$ is not empty).

$$
\begin{array}{lcl}
\Phi([t]) & ::= & \displaystyle\bigvee_{\substack{(P,N)\in t \\ u \mid P\subseteq T_u}} \Phi_u(P,N) \\[2ex]
\Phi_{\mathbf{basic}}(P,N) & ::= & \begin{cases} 0 & \text{if } \mathcal{C} \cap \displaystyle\bigcap_{b\in P} \mathbb{B}[\![b]\!] \subseteq \bigcup_{b\in N} \mathbb{B}[\![b]\!] \\ 1 & \text{otherwise} \end{cases} \\[2ex]
\Phi_{\mathbf{prod}}(P,N) & ::= & \displaystyle\bigvee_{N'\subseteq N\cap T_{\mathbf{prod}}} \begin{cases} \left[\displaystyle\bigwedge_{t_1\times t_2\in P} t_1 \setminus \bigvee_{t_1\times t_2\in N'} t_1\right] \\ \qquad\qquad \wedge \\ \left[\displaystyle\bigwedge_{t_1\times t_2\in P} t_2 \setminus \bigvee_{t_1\times t_2\in N\setminus N'} t_2\right] \end{cases} \\[2ex]
\Phi_{\mathbf{fun}}(P,N) & ::= & \displaystyle\bigwedge_{t'_1\to t'_2\in N} \bigvee_{P'\subseteq P} \begin{cases} \left[t'_1 \setminus \displaystyle\bigvee_{t_1\times t_2\in P'} t_1\right] \\ \qquad\qquad \wedge \\ \begin{cases} 1 & \text{if } P = P' \\ \left[\left(\displaystyle\bigwedge_{t_1\times t_2\in P\setminus P'} t_2\right) \setminus t'_2\right] & \text{if } P \neq P' \end{cases} \end{cases}
\end{array}
$$

**Implementation** We have chosen to present separately our generic algorithms to compute predicates defined inductively by boolean formulas, and the function $\Phi$ that generates the subtyping predicate (in fact, the negation of emptiness checking). A naive implementation would consist of effectively manipulating syntactic terms to represent boolean formulas $\Phi([t])$, and using generic implementations of the algorithms of the previous section (with or without backtracking). In fact, by specializing these algorithms with the particular definition of $\Phi$, we can avoid producing these terms (which are immediately deconstructed by the algorithms). The formulas then correspond to checkpoints of the implementation. For the algorithm without backtracking, however, we sometimes have to store constraints in the table $\sigma$; in practice, such a constraint may be represented as a closure obtained by the partial application of `trigger` to the constraint.

## 7.3 Variant and optimization

### 7.3.1 Other formulas

The formulas obtained in the previous section, which come from the definition of a simulation, ultimately come from the set-theoretic result stated in Lemma 4.6. We will now show how, starting from a different set-theoretic property, we can get an algorithm that is potentially more efficient.

**Lemma 7.33** *Let $X, X' \subseteq D_1$, $Y, Y' \subseteq D_2$. Then:*

$$(X \times Y)\backslash(X' \times Y') = ((X\backslash X') \times Y) \cup (X \times (Y\backslash Y'))$$

Consider now a type $t$, and $(P, N) \in t$, with $P \subseteq T_{\mathbf{prod}}$. Writing:

$$P = \{t_1^1 \times t_2^1; \ldots; t_1^n \times t_2^n\}$$

$$N \cap T_{\mathbf{prod}} = \{s_1^1 \times s_2^1; \ldots; s_1^m \times s_2^m\}$$

Writing $t_1 = \bigwedge_{i=1..n} t_1^i$ and $t_2 = \bigwedge_{i=1..n} t_2^i$. We study the question of whether the following is true:

$$(*) \quad [\![t_1]\!] \times [\![t_2]\!] \subseteq \bigcup_{i=1..m} [\![s_1^i]\!] \times [\![s_2^i]\!]$$

Clearly, if $[\![t_1]\!] = \emptyset$ or $[\![t_2]\!] = \emptyset$, then the assertion $(*)$ is true. Otherwise, if $m = 0$, the assertion is false, and if $m \neq 0$, we use Lemma 7.33 (by isolating the first term of the union), and we obtain that $(*)$ is true if and only if the two following assertions are true:

$$\begin{cases} [\![t_1']\!] \times [\![t_2]\!] \subseteq \bigcup_{i=2..m} [\![s_1^i]\!] \times [\![s_2^i]\!] \\ [\![t_1]\!] \times [\![t_2']\!] \subseteq \bigcup_{i=2..m} [\![s_1^i]\!] \times [\![s_2^i]\!] \end{cases}$$

where $t_1' = t_1 \backslash s_1^1$ and $t_2' = t_2 \backslash s_2^1$. Each of these two assertions is in the same form as $(*)$. This decomposition can therefore be continued for each of the products $s_1^i \times s_2^i$. This gives an alternative definition of the set $\mathbb{E}\mathcal{S}$, and therefore different formulas $\Phi(t)$:

$$\Phi_{\mathbf{prod}}(P, N) ::= [t_1] \wedge [t_2] \wedge \Phi'_{\mathbf{prod}}(t_1, t_2, N \cap T_{\mathbf{prod}})$$

where:

$$t_1 = \bigwedge_{t_1 \times t_2 \in P} t_1$$

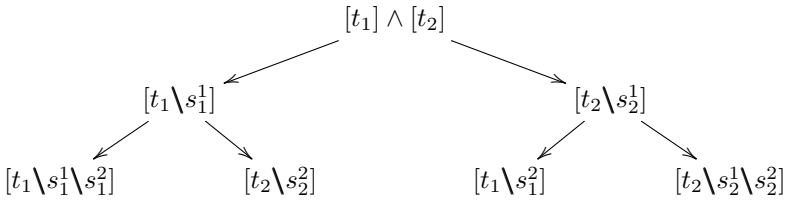
$$t_2 = \bigwedge_{t_1 \times t_2 \in P} t_2$$

and the formulas $\Phi'_{\mathbf{prod}}(t_1, t_2, N)$ are defined by:

$$\begin{array}{lcl} \Phi'_{\mathbf{prod}}(t_1, t_2, \emptyset) & ::= & 1 \\ \Phi'_{\mathbf{prod}}(t_1, t_2, \{s_1 \times s_2\} \cup N') & ::= & \begin{cases} [t_1 \backslash s_1] \wedge \Phi'_{\mathbf{prod}}(t_1 \backslash s_1, t_2, N') \\ \qquad \vee \\ [t_2 \backslash s_2] \wedge \Phi'_{\mathbf{prod}}(t_1, t_2 \backslash s_2, N') \end{cases} \end{array}$$

(the choice of $s_1 \times s_2$ in $N$ can be arbitrary)

We can see these formulas as binary trees. Hence for $N = \{s_1^1 \times s_2^1; s_1^2 \times s_2^2\}$, the tree is:



The corresponding boolean formula can be interpreted as the existence of a branch whose nodes are all true. We immediately see that these new formulas allow for some significant pruning: as soon as a node is false, there is no need to consider the corresponding subtree (and the algorithms of the previous section will take this pruning into account).

We can introduce an additional optimization to this presentation, whose motivation comes from the following set-theoretic property, that completes Lemma 7.33.

**Lemma 7.34** *Let $X, X' \subseteq D_1$, $Y, Y' \subseteq D_2$. If we have $X \cap X' = \emptyset$ or $Y \cap Y' = \emptyset$, then:*

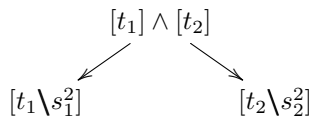
$$(X \times Y) \backslash (X' \times Y') = (X \times Y)$$

This suggests taking:

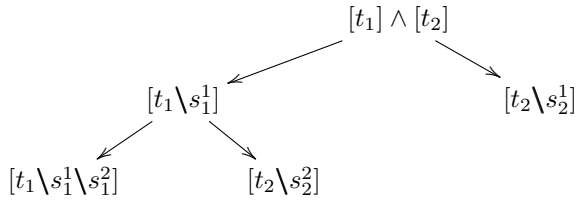
$$\Phi'_{\mathbf{prod}}(t_1, t_2, \{s_1 \times s_2\} \cup N') ::= \Phi'_{\mathbf{prod}}(t_1, t_2, N')$$

instead of the generic formula for $\Phi'_{\mathbf{prod}}(t_1, t_2, \{s_1 \times s_2\} \cup N')$, when we already know that $[\![t_1 \wedge s_1]\!] = \emptyset$ or that $[\![t_2 \wedge s_2]\!] = \emptyset$, based on the current assumptions of the algorithms (for the algorithm with cache, by considering the set of negative assumptions, and for the algorithm without backtracking, by considering the table $\sigma$).

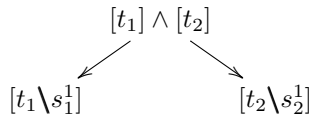
For example, let us take back the example with $N = \{s_1^1 \times s_2^1; s_1^2 \times s_2^2\}$. If we already know (ou are assuming) that, say, $[\![t_1 \wedge s_1^1]\!] = \emptyset$, then the decision tree becomes:




Similarly, if we know that $[\![t_1 \wedge s_1^2]\!] = \emptyset$, or that $[\![(t_2 \backslash s_2^1) \wedge s_2^2]\!] = \emptyset$, then the decision tree becomes:




Note that in the case $[\![t_1 \wedge s_1^2]\!] = \emptyset$, it would have been smarter to consider the product $s_1^2 \times s_2^2$ before $s_1^1 \times s_2^1$, which would have given the decision tree:




Generally, even without this optimization, it is best to first consider the (set-theoretically) "largest" products of $N$, so as to cut the path of the decision trees higher. Hence we could imagine using heuristics (that quickly estimate a certain measure of the "size" of types) to decide in which order to consider the products $N$. We have not obtained any results in this direction.

Finally, let us note that everything that has been said in this section also applies *mutatis mutandis* to the formulas $\Phi_{\mathbf{fun}}$, that also comes from Lemma 4.6 (via Lemma 4.9).

### 7.3.2 Approximations

We have seen how to modify the boolean formulas that decide whether a type is empty or not, in order to optimize the subtyping algorithm. Another approach consists in searching for approximations, that is, necessary conditions and sufficient conditions, for a type to be empty. This can short-circuit the generic algorithm in simple and typical cases. Assuming that we have two systems of constraints over the variables $[t]$ (for $t$ in a given socle), says, $S_0$ and $S_1$ that approximate the subtyping relation:

$$[\![S_0]\!](t) = 0 \Rightarrow [\![t]\!] = \emptyset$$

$$[\![S_1]\!](t) = 1 \Rightarrow [\![t]\!] \neq \emptyset$$

We assume that these systems are given by formulas:

$$S_i = \{\Phi_i(t) \Rightarrow t \mid t \in \beth\}$$

If the predicates $[\![S_0]\!]$ and $[\![S_1]\!]$ are way easier to compute than $[\![S]\!]$, we can replace $\Phi(t)$ by:

$$\Phi'(t) = [\![S_1]\!](t) \vee ([\![S_0]\!](t) \wedge \Phi(t))$$

Therefore, we use complex formulas for $\Phi(t)$ only when the approximation are not enough to check that $[\![t]\!] = \emptyset$.

Eventually, we can nest the computation of $[\![S_0]\!]$, $[\![S_1]\!]$ and $[\![S]\!]$, by using mutually recursive systems (formally, each type $t$ results in three variables $[t], [t]_0, [t]_1$, and we use a unique function from variables to boolean formulas).

Here is a typical example of approximation. Consider this instance of subtyping:

$$t \times s \leq \bigvee_{i=1..n} t_i \times s_i \quad (*)$$

Clearly, to have $(*)$, we only need $t \leq t_{i_0}$ and $s \leq s_{i_0}$ for a certain $i_0$. Conversely, a necessary condition to get $(*)$ is to either have $t \leq \mathbb{0}$, or the conjunction of $t \leq \bigvee_{i=1..n} t_i$ and of $s \leq \bigvee_{i=1..n} s_i$. These approximations induce the definition of systems of constraints $S_0$ and $S_1$, whose smallest fixed points are easier to compute than the one of $S$.

# Chapter 8

# Pattern matching compilation

In this chapter, we study the implementation of pattern matching (Chapter 6). Given a value $v$ and a pattern $p$, it consists in computing the result $v/p$ of matching $v$ to $p$. We study the situation at compile time: we know $p$ in advance, which allows us to perform some possibly expensive computations in order to prepare the efficient evaluation of the pattern match $v/p$ for some values $v$ available at runtime.

Consider the evaluation of the expression $\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2$. Once $e$ has been reduced to a value $v$, we have, according to the semantics of the construction, to compute $v/p_1$, and then eventually $v/p_2$. We will develop techniques to efficiently evaluate several patterns on the same value in parallel.

We are mainly interested in the structural aspect of pattern matching, that is, we assume that we can efficiently evaluate type test expressions for "simple" $t$ types, i.e. such that $t\mathbin{\wedge}\mathbb{1}_{\mathbf{prod}} \simeq \mathbb{0}$ (where $\mathbb{1}_{\mathbf{prod}} = \mathbb{1}\mathbin{\times}\mathbb{1}$). We see values as binary trees whose leaves are constants or abstractions (i.e. elements of $[\![\mathbb{1}\backslash\mathbb{1}_{\mathbf{prod}}]\!]$). Internal nodes (and therefore sub-trees) correspond to sub-values. For any value $v = (v_1, v_2)$, we see $v_1$ (resp. $v_2$) as the left sub-tree (resp. right). A value is said to be simple if it is not a pair.

Every pattern considered in this chapter is assumed to be in normal form, within the meaning of Section 6.5.2. In particular, all the types that appear inside of patterns are simple. We fix a $G'$-basis $B$ (still within the meaning of Section 6.5.2), and we will only study patterns $\sigma(q)$ for $q \in B$, and their sub-patterns. There is only a finite number of such patterns.

We write $v/q$ instead of $v/\sigma(q)$, $\lceil q \rfloor$ instead of $\lceil \sigma(q) \rfloor$, and $\mathrm{Var}(q)$ instead of $\mathrm{Var}(\sigma(q))$.

## 8.1 Naive evaluation

The semantics of pattern matching (Figure 6.1) directly gives a naive algorithm to compute pattern matches, by induction on the lexicographically ordered pair $(v, p)$. To compute $v/p_1|p_2$, we start by computing $v/p_1$, and we only compute $v/p_1$ if the result is $\Omega$. Similarly, to compute $(v_1, v_2)/(q_1, q_2)$, we start, for

example, by computing $v_1/\sigma(q_1)$ and we only compute $v_2/\sigma(q_2)$ if the result is not $\Omega$.

## 8.2 Optimization ideas

**Ignore a sub-tree** A first optimization from the naive algorithm consists in detecting the patterns $p$ such that $\lbag p \rbag \simeq \mathbb{1}$ and $\mathrm{Var}(p) \simeq \emptyset$; when evaluating $v/p$ for such a pattern $p$, we can completely ignore the sub-tree $v$ and directly output the result $\{\}$.

**Ordering computations** When computing $v/p_1|p_2$, the choice of starting with $p_1$ is quite natural given the *first-match* policy (if $p_1$ succeeds, it is not necessary to consider $p_2$). However, it is sometimes better to start with $p_2$, for example if $\lbag p_1 \rbag \wedge \lbag p_2 \rbag \simeq \mathbb{0}$ (because then, if $p_2$ succeeds, we will not have to consider $p_1$) and if computing $v/p_2$ is faster than $v/p_1$ (or if we find that statistically $v/p_2$ succeeds more often than $v/p_1$).

**Memoization** The naive algorithm we sketched in the previous section can be made to compute $v/p$ several times for the same sub-value $v$ and the same pattern $p$ (and the number of times it could do that is unbounded, meaning an exponential complexity in the size of $v$). We can avoid these useless computations with a *memoization* technique (by keeping track of already made computations). This ensures a time complexity linear in the size of the value $v$ (because the number of different patterns is finite).

**Determinization** In fact, we can force the algorithm to go through each sub-value only once at most. All we need to do is to compute in parallel all the $v/q$ for $q \in B$ by starting from the leaves of $v$. Knowing the two families $(v_1/q)_{q\in B}$ and $(v_2/q)_{q\in B}$, we can easily compute $((v_1, v_2)/q)_{q\in B}$. This corresponds to the classic determinization algorithm for bottom-up tree automata. This evaluation technique is of course very costly, because we must go through every node of the tree (we cannot ignore a sub-tree), and we have to do the same computations everywhere, although for some value $(v_1, v_2)$ and pattern $(q_1, q_2)$, we only need $v_1/q_1$ and $v_2/q_2$, but not $v_1/q_2$ or $v_2/q_1$. So we are doing a lot of useless computations.

**Top-down determinization** It is possible to refine the determinization technique, by computing in parallel not all of the $v/q$, but only a certain number. If we have to compute $(v/q)_{q\in R}$ for $R \subseteq B$ and $v = (v_1, v_2)$, then we compute (at compile time) two sets $R_1 \subseteq B$ and $R_2 \subseteq B$ such that $(v/q)_{q\in R}$ can be computed only using $(v_1/q)_{q\in R_1}$ and $(v_2/q)_{q\in R_2}$. To minimize the number of computations that need to be done for every sub-tree, we try to take $R_1$ and $R_2$ as small as possible. For example, if $R = \{q_0\}$, $\sigma(q_0) = (q_1, q_1')|\ldots|(q_n, q_n')$, then we can take $R_1 = \{q_i \mid i = 1..n\}$, and $R_2 = \{q_i' \mid i = 1..n\}$.

**Lateral propagation** Let us continue the idea of the previous paragraph. Since the computations of $(v_1/q)_{q\in R_1}$ and of $(v_2/q)_{q\in R_2}$ will be done sequentially, then can make, say, $R_2$, depend from the result of $(v_1/q)_{q\in R_1}$ (rather than

from the fact that every $v_1/q$ is either $\Omega$ or not). In the previous paragraph example, we could take $R_1 = \{q_i \mid i = 1..n\}$, and $R_2 = \{q'_i \mid v_1/q_i \neq \Omega\}$. Of course, the symmetric choice is also valid (to start with $v_2$, and make $R_1$ depend from the result).

**Determinization ?** Even when applying the ideas of the two previous paragraphs, determinization can induce some useless computations. If $\sigma(q_0) = (q_1, q'_1)|(q_2, q'_2)$, it may be best, instead of evaluating $q_1$ and $q_2$ in parallel on the left sub-tree, to start with $q_1$, and if successful, to continue with $q'_1$ on the right sub-tree. We only consider $q_2$ if one of these computations fail, which can be faster. There is a part of heuristics in choosing between one approach or the other (determinization or sequential evaluation), because the best choice depends on the data actually present during execution.

**Taking types into account** We do not usually need to evaluate $v/p$ for every value $v$. Indeed, the static type system of the language gives a type to the inputs of the pattern match. We can assume that $v$ is in a certain type $t_0$. This static information can help avoid some unnecessary computations. For example, for a leaf, we can avoid a type test for $t$ (a simple type) if the type information $t_0$ at this level is such that $t_0 \leq t$ (then the test will necessarily succeed). This can be generalized to the case of an arbitrary $p$ such that $t_0 \leq \lfloor p \rfloor$ and $\mathrm{Var}(p) = \emptyset$. Similarly, if $t_0 \wedge \lfloor p \rfloor \simeq \mathbb{0}$, we can completely ignore the corresponding sub-tree because the result will necessarily be $\Omega$.

When evaluating $v/p_1|p_2$ with some static information, we can assume, independently from the evaluation order (between $p_1$ and $p_2$) and from a possible computation in parallel (determinization), that when computing $p_2$ we have the static information $t_0 \setminus \lfloor p_1 \rfloor$. This might simplify the computation (compared to the mere static information $t_0$). Indeed, if the additional assumption we made (that the value is in $\neg \lfloor p_1 \rfloor$) is wrong, then the result of this computation will be ignored anyway, hence it is acceptable to output a wrong result for $v/p_2$.

Therefore we interpret the static information $t_0$ not as a guarantee that the value $v$ is in $[\![t_0]\!]$, but as the contract that demands a correct result only when that condition is achieved (and authorizes any outcome otherwise).

To compile the pattern matching expression `match` $e$ `with` $p_1 \to e_1 \mid p_2 \to e_2$ with an input type[1] $t_0$, we can compute $p_1$ and $p_2$ in parallel (or not); for $p_1$, the static information is $t_0$ and for $p_2$, we take $t_0 \setminus \lfloor p_1 \rfloor$.

**Type propagation** The static information gets propagated to the sub-values. So, knowing that $v = (v_1, v_2)$ is in $t_0$, we then also know that $v_1$ is in $\pi_1[t_0 \wedge \mathbb{1}_{\mathbf{prod}}]$ and that $v_2$ is in $\pi_2[t_0 \wedge \mathbb{1}_{\mathbf{prod}}]$.

During the evaluation of pattern matching, the computations made allow to gain information on the type of the value and of its sub-values, and thus to refine the static information. For example, if we compute $v_1/q_1$ on the left sub-tree, and it fails, we then know that the value $v = (v_1, v_2)$ is not only of type $t_0$ (*a priori* information), but also of type $\neg(\lfloor q_1 \rfloor \times \mathbb{1})$, which can be used for further computations over $v$ (and $v_1$). If the computation of $v_1/q_1$ succeeds,

[1] A pattern matching expression can be typed several times by the typing algorithm, if it is inside of an overloaded function. In that case, the type $t_0$ is the union of all the types computed for the expression $e$.

then we know that $v$ is also of type $\lceil q_1 \rfloor \times \mathbb{1}$, which might provide a more accurate information for $v_2$.

**Factorization of captures** In some cases, to compute $v/p$, we can completely ignore $v$. We have seen that it is the case if the static information $t_0$ that we have over $v$ is such that $t_0 \wedge \lceil p \rfloor \simeq \mathbb{0}$ (the result is then $\Omega$), or such that $t_0 \leq \lceil p \rfloor$ and $\mathrm{Var}(p) = \emptyset$ (the result is then $\{\}$). When $t_0 \leq \lceil p \rfloor$, but $\mathrm{Var}(p) \neq \emptyset$, the result can still sometimes be directly computed, for example if $p$ is of the form $x_1 \& x_2 \& \dots x_n \& p_0$, with $\mathrm{Var}(p_0) = \emptyset$ (or by replacing the $x_i$ with $(x_i := c_i)$). It may be necessary to rewrite the pattern $p$ to put it in this form. For example, if $\sigma(q) = (q, q) | x$, then $\sigma(q) \simeq x$. We also may have to consider the type $t_0$. For example, if $p = (q_0, q_1)$ with $\sigma(q_0) = x$, $\sigma(q_1) = (x := c)$ and $t_0 = \mathbb{1} \times b_c$, then we can replace $p$ by $x$.

Even if $t_0 \not\leq \lceil p \rfloor$, we might want to "pull up" the captures inside the patterns, to simplify computations in the sub-trees. For example, if $\sigma(q_0) = (q_1, q_0) | ((x := c) \& b_c)$, $\sigma(q_1) = x \& t_1$, then we can replace $\sigma(q_0)$ by $x \& q_0'$ where $\sigma(q_0') = (q_1', q_0') | b_c$, $\sigma(q_1') = t_1$. This avoids having to rebuild already existing pairs.

**Merging patterns** Consider an alternation pattern $(q_1, q_2) | (q_3, q_4)$. If $q_1$ and $q_3$ are equivalent according to the static information that we have (i.e. for any value in $\pi_1[t_0]$, they output the same result), then we can merge the terms of the alternation into $(q_1, q)$ with $\sigma(q) = q_2 | q_4$. Indeed, computing a result for $q$ might be simpler than computing a result for $q_2$ and for another $q_4$, as we would do in the original pattern. For example, if $\sigma(q_2)$ and $\sigma(q_4)$ do not have any variables and $\lceil \sigma(q_2) \rfloor = \neg \lceil \sigma(q_4) \rfloor$, then computing a result for $q$ is trivial, while computing the same result for $q_2$ alone might be complicated.

## 8.3 Formalization

Our algorithm to compile patterns is not rigid. Indeed, as we have seen in the previous section, some choices can only be done through heuristics. To experiment several heuristics, either dynamically or by using statistical data collected from previous executions (*profiling*), we will set up a formalism that will express the different optimizations and strategies presented above.

We still fix a $G'$-basis $B$ (in the meaning of Section 6.5.2). We also fix a socle $\beth$, that will allow us to express the static information $t_0$. We assume that $\beth$ contains $\mathbb{1}_{\mathbf{prod}}$, as well as all the other $\lceil p \rfloor$ for the sub-patterns $p$ of the $\sigma(q)$ ($q \in B$).

We write $p \xrightarrow{t_0} x$ the predicate:

$$(x \in \mathrm{Var}(p)) \wedge (\forall v \in [\![ t_0 \wedge \lceil p \rfloor ]\!].\ (v/p)(x) = v)$$

and $p \xrightarrow{t_0} (x := c)$ the predicate:

$$(x \in \mathrm{Var}(p)) \wedge (\forall v \in [\![ t_0 \wedge \lceil p \rfloor ]\!].\ (v/p)(x) = c)$$

We will give algorithms to compute these predicates in Section 8.4 , which are used to express the fact that a capture variable (or a pattern constant) can be factorized inside of a pattern.

### 8.3.1 Queries

A query is the specification of a computation to be performed on a value. In the case of the naive evaluation, we can identify a query to a pattern node $q$. To be able to do several computations in parallel (determinization), we must have queries with several nodes. To be able to simplify the computations by taking into account an assumption on the type of the input vale, we associate to each node $q$ of the query a type $t_0$. It is possible to have different types for each node of the query. Indeed, as we have seen earlier, this type $t_0$ is not a strong constraint on the input value: the value might not be in $t_0$, but then we would be free to output any result for the corresponding computation Finally, in order to factorize captures, we can take the restriction of a node $q$ to a certain set of variables $X \subseteq \mathrm{Var}(q)$.

**Definition 8.1 (Queries)** *A* **good triplet** *is a triplet $(p, t_0, X)$ where $p$ is a pattern built over $B$, $t_0 \in \sqsupset$ and $X \subseteq Var(p)$.*

*An* **atomic query** *$r$ is a triplet $(q, t_0, X)$ with $q \in B$, $t_0 \in \sqsupset$, and $X \subseteq Var(q)$. We write $Var(r) = X$ and $\sigma(r)$ the good triplet $(\sigma(q), t_0, X)$.*

*A* **query** *$R$ is a finite set of atomic queries.*

**Lemma 8.2** *There is only a finite number of queries.*

*Proof*: The sets $B$ and $\sqsupset$ are finite. □

### 8.3.2 Results

**Definition 8.3** *An* **atomic result** *$\Gamma$ for some finite set of variables $X$ is either a function from $X$ to values, or $\Omega$. We sometimes write $\Gamma^X$ to recall the support of $\Gamma$.*

*If $\Gamma$ is a result for $X$ and $X' \subseteq X$, we write $\Gamma_{|X'}$ the restriction of $\Gamma$ to $X'$ ($\Omega_{|X'} = \Omega$, $\gamma_{|X'} = \{x \mapsto \gamma(x) \mid x \in X'\}$).*

*An atomic result for some atomic query $r$ is an atomic result for $Var(r)$.*

*A* **result** *$\mathbb{R}$ for some query $R$ is a family $\mathbb{R} = (\Gamma_r)_{r \in R}$ of atomic results for each of the components. We also write $Dom(\mathbb{R}) = R$.*

The only atomic result for $R = \emptyset$ is written $\mathbb{R}^\emptyset$.

**Definition 8.4 (Correct results)** *An atomic result $\Gamma$ for $X$ is said* **correct** *for some good triplet $(p, t_0, X)$ if:*

$$v \in [\![t_0]\!] \Rightarrow \Gamma = (v/p)_{|X}$$

*We then write $v \Vdash \Gamma : (p, t_0, X)$.*

*We define the correction of an atomic result $\Gamma$ for some query $r$ and some value $v$ by:*

$$v \Vdash \Gamma : r \iff v \Vdash \Gamma : \sigma(r)$$

*We define the correction of a result $\mathbb{R} = (\Gamma_r)_{r \in R}$ for some query $R$ and some value $v$ by:*

$$v \Vdash \mathbb{R} \iff \forall r \in R.\ v \Vdash \Gamma_r : r$$

The following lemma is trivial to establish.

**Lemma 8.5** *For any value $v$ and any atomic query $r$ or any query $R$, there exists (at least one) result correct for $v$.*

We are trying to efficiently compute *one* result correct for some given query $R$ and value $v$.

**Definition 8.6** *If $\mathbb{R}_1$ and $\mathbb{R}_2$ are two results, we write $\mathbb{R}_1 \cup \mathbb{R}_2$ the result of domain $Dom(\mathbb{R}_1) \cup Dom(\mathbb{R}_2)$ defined by:*

$$(\mathbb{R}_1 \cup \mathbb{R}_2)_r = \begin{cases} (\mathbb{R}_1)_r & \text{if } r \in Dom(\mathbb{R}_1) \backslash Dom(\mathbb{R}_2) \\ (\mathbb{R}_2)_r & \text{if } r \in Dom(\mathbb{R}_2) \end{cases}$$

### 8.3.3 Query compilation

The finiteness property of the number of queries allows us to place ourselves in a compilation situation: each query $R$ will give rise, in the generated code, to a function that associates to a value $v$ a correct result for $R$ over $v$. To do so, it can apply queries a finite number of times on the *sub-values* $v_1$ and $v_2$ when $v = (v_1, v_2)$. This corresponds to recursive calls between the different functions associated with the requests, and termination is guaranteed (because the values are finite trees). Overall, we can see the evaluation of pattern matching on a value as the traversal of a tree (value) by a tree automaton, whose states correspond to queries.

Of course, the number of requests, though finite, is gigantic (exponential in the product of the cardinal of $B$, times the cardinal of $\sqsupset$, times the exponential of the number of variables), but in practice, we only generate the code corresponding to queries that might be used, and this number is generally reasonable.

The code in charge of evaluating the queries has the following form (in pseudo-ML):

```
let rec eval_{R_1} v = ...
    and eval_{R_2} v = ...
    ...
    and eval_{R_n} v = ...
```

We can introduce some amount of memoization for the functions associated with queries, or for some of them (it is a heuristic choice). We only have to keep track of already-computed results, and if a query must be evaluated again on a previously considered value, then it can output the memoized result.

We still have to define the ... in the pseudo-code above, that is, the way to evaluate a query $R$ over a value $v$. There are two cases. First, it may be possible to compute a correct result without considering the form of the value $v$. That is the case if for every atomic queries $r = (q, t_0, X)$ in $R$, we have :

— either $t_0 \wedge \wr q \int \simeq \mathbb{0}$;

— or $t_0 \leq \wr q \int$ and all the variables $x \in \mathrm{Var}(q)$ can be factorized ($\sigma(q) \xrightarrow{t_0} x$ or $\sigma(q) \xrightarrow{t_0} (x := c)$ for some constant $c$).

In both cases, we can compute a correct result for $r$ without considering $v$.

Otherwise, we have to distinguish according to the form of $v$: constant, abstraction or pair. The pseudo-code of the function evaluating the query $R$ will then have the form:

```
eval_R v = match v with
  | (v_1,v_2) -> ... (* Code for the pairs *)
  | c -> ... (* Code for the constants *)
  | f -> ... (* Code for the abstractions *)
```

If $v$ is a simple value (constant or abstraction), we have to do a certain amount of type tests over $v$, according to the patterns $t$ that appear in the $\sigma(q)$ (for $(q, t_0, X) \in R$). We see $t$ as a boolean combination of atoms: $t \simeq \bigvee_{(P,N)\in t}(\bigwedge_{a\in P} a)\wedge(\bigwedge_{a\in B}\neg a)$. By using the fact that $v \in [\![t_1 \vee t_2]\!] \iff (v \in [\![t_1]\!]) \vee (v \in [\![t_2]\!])$ and some similar properties for $t_1 \wedge t_2$ and $\neg t$, it all comes down to type tests with atomic types $a$. It is possible to take into account the static information of atomic queries to optimize these tests (we can know whether they will succeed or fail, only by looking at the static information $t_0$). If $v$ and $a$ do not have the same kind, the test fails. If $v$ is a constant and $a$ is an atomic base type, we can use several *dispatch* techniques, according to the base types considered (for example, research trees, *tries*, ...). If $v$ is an abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$ and $a$ is an arrow type, we use the equivalence:

$$v \in [\![a]\!] \iff \bigwedge_{i=1..n} t_i \to s_i \leq a \qquad (*)$$

Hence, we only need to extract the interface of the function. If we assume to be working in a closed world, then we know the full set of abstractions of the program, and all the type tests that could be done on the interfaces of these abstraction. It is possible to precompute the tests $(*)$ for all the abstractions and all the type tests on arrow types. This allows, for example, to keep only an integer at runtime to represent the interface of a function and to avoid subtyping tests.

In the rest of this chapter, we only study the case of a pair value $v = (v_1, v_2)$. We then have to do some computations on the sub-values $v_1$ and $v_2$, that is, to apply some $R'$ sub-queries to them (by calling the functions $\texttt{eval}_{R'}$). Each sub-query collects information that is relevant to the computation to be done. When enough information is available, we can finally compute the result of the query $R$. The selection of sub-queries to be evaluated on the sub-values is sequential: if the first sub-query can only depend on $R$, the choice of the next sub-queries can depend on the results of the sub-queries previously evaluated. Specifically, this choice may depend on the success or failure of each atomic request of the sub-requests evaluated. We have to choose a strategy to make that choice. In the following sections, we formalize a target language used to formalize these sub-query choices.

### 8.3.4 Target language: syntax

**Definition 8.7 (Static results)** *The set of* **static atomic results** *is* $\{\Omega, \checkmark\}$. *We use the letter* $\mathtt{r}$ *to denote a static result.*

*A* **static query result** *is a partial function from the set of atomic queries to* $\{\Omega, \checkmark\}^R$. *We denote by* $\mathtt{R}$ *a static query result* $(\mathtt{r}_r)_{r\in R}$. *We set* $Dom(\mathtt{R}) = R$, *and we write* $\mathtt{R} : R$ *to denote that* $Dom(\mathtt{R}) = R$. *We denote by* $\mathtt{R}^{\emptyset}$ *the unique static result for the empty query.*

*The function* $|\_|$ *associates to an (atomic) result a static (atomic) result defined by:* $|\Omega| = \Omega$, $|\gamma| = \checkmark$, $|(\mathbb{r}_r)_{r\in R}| = (|\mathbb{r}_r|)_{r\in R}$.

We will represent by a pair $I = (R_1, R_2)$ the sub-queries evaluated on the sub-values. We write $I^\emptyset = (\emptyset, \emptyset)$. We write $I \cup (1 : R) = (R_1 \cup R, R_2)$ and $I \cup (2 : R) = (R_1, R_2 \cup R)$.

**Definition 8.8 (Target language)** *A* **target expression** *for a set of variables $X$ and a pair of sub-queries $I = (R_1, R_2)$ is a term generated by the productions:*

$$\begin{array}{rcll} \mathtt{e}^{X,I} & := & x & \text{where } X = \{x\} \\ & | & \mathbb{r}^X & \\ & | & (i, r_{|X}) & \text{where } i = 1..2, r \in R_i, X \subseteq Var(r) \\ & | & \mathtt{e}_1^{X_1,I} \oplus \mathtt{e}_2^{X_2,I} & \text{where } X = X_1 \cup X_2 \end{array}$$

*where $\mathbb{r}^X$ is an atomic result for $X$, $i = 1..2$, $r$ is an atomic query, $X$ a finite set of variables.*

*A* strategy *for some query $R_0$ and a pair of sub-queries $I$ is a finite term generated by the productions:*

$$\begin{array}{rcl} \mathtt{s}^{R_0,I} & := & (\mathtt{e}_r^{Var(r),I})_{r \in R_0} \\ & | & (i, R) \mapsto (\mathtt{s}_{\mathtt{R}}^{R_0, I \cup (i:R)})_{\mathtt{R}:R} \end{array}$$

*where $i = 1..2$, $R$ is a query, and the $\mathtt{e}_r$ are target expressions.*

The exponents are there to ensure well-formedness constraints over the introduced objects. We sometimes omit them when there is no ambiguity.

A strategy abstractly represents a piece of code responsible for calculating a correct result for a query $R_0$ on an input value $v = (v_1, v_2)$. This code has the form of a finite tree. An internal node $(i, R) \mapsto (\mathtt{s}_\mathtt{R})_{\mathtt{R}:R}$ corresponds to the evaluation of the sub-query $R$ over the value $v_i$. The family $(\mathtt{s}_\mathtt{R})_{\mathtt{R}:R}$ tells how to continue the computation, depending on the success or failure of each atomic query in $R$. The leaves of the tree are reached when enough information is available to compute a correct result for $R_0$. The result for an atomic query $r \in R_0$ is computed by a target expression. The target expression $\mathbb{r}^X$ outputs the denoted result. The target expression $x$ outputs the environment $\{x \mapsto v\}$ where $v$ is the current value. The target expression $\mathtt{e}_1 \oplus \mathtt{e}_2$ combines two results. The target expression $(i, r_{|X})$ uses the result of an atomic sub-query $r$ over $v_i$, by restricting it to the variables of $X$. This result must be available, that is, it must have been computed by the strategy during the call of a sub-query $R$ over $v_i$, with $r \in R$.

### 8.3.5 Target language: semantics

We are going to formalize the semantics of the target language. The results of the sub-requests evaluated during a strategy should be stored. For this we define the concept of information packet.

**Definition 8.9 (Packets)** *An* **information packet** $\mathbb{I}$ *is a pair of query results $(\mathbb{R}_1, \mathbb{R}_2)$. We write $Dom(\mathbb{I}) = (Dom(\mathbb{R}_1), Dom(\mathbb{R}_2))$. We write $\mathbb{I}^\emptyset$ the information packet $(\mathbb{R}^\emptyset, \mathbb{R}^\emptyset)$.*

**Definition 8.10** *We define the correction of an information packet* $(\mathbb{R}_1, \mathbb{R}_2)$ *for some value* $v = (v_1, v_2)$ *by:*

$$v \Vdash (\mathbb{R}_1, \mathbb{R}_2) \iff \forall i.\ v_i \Vdash \mathbb{R}_i$$

If $\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2)$ is an information packet, we write $\mathbb{I} \cup (1 : \mathbb{R}) = (\mathbb{R}_1 \cup \mathbb{R}, \mathbb{R}_2)$ and $\mathbb{I} \cup (2 : \mathbb{R}) = (\mathbb{R}_1, \mathbb{R}_2 \cup \mathbb{R})$.

**Lemma 8.11** *Let* $v = (v_1, v_2)$ *be a value and* $i = 1..2$. *If* $v_i \vDash \mathbb{R}$ *and* $v \vDash \mathbb{I}$, *then* $v \vDash \mathbb{I} \cup (i : \mathbb{R})$.

Figure 8.1 introduces two judgments that formalize the semantics of the target language:

— Evaluation of a strategy: $\mathtt{s}^{R_0,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0$, where $v$ is a value of the form $(v_1, v_2)$, $\mathrm{Dom}(\mathbb{I}) = I$, and $\mathrm{Dom}(\mathbb{R}_0) = R_0$.

— Evaluation of a target expression: $\mathtt{e}^{X,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}$ where $\mathbb{r}$ is a result for $X$.

$$\frac{(\forall r \in R_0)\ \mathtt{e}_r \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_r}{(\mathtt{e}_r)_{r \in R_0} \overset{v,\mathbb{I}}{\rightsquigarrow} (\mathbb{r}_r)_{r \in R_0}}\ (assemble)$$

$$\frac{v_i \Vdash \mathbb{R} \quad \mathrm{Dom}(\mathbb{R}) = R \quad \mathtt{s}_{|\mathbb{R}|} \overset{v,\mathbb{I} \cup (i:\mathbb{R})}{\rightsquigarrow} \mathbb{R}_0}{(i, R) \mapsto (\mathtt{s}_{\mathtt{R}})_{\mathtt{R}:R} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0}\ (subreq)$$

$$\frac{}{x \overset{v,\mathbb{I}}{\rightsquigarrow} \{x \mapsto v\}}\ (capture)$$

$$\frac{}{\mathbb{r}^X \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}^X}\ (const)$$

$$\frac{\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2) \quad \mathbb{r} = (\mathbb{R}_i)_r}{(i, r_{|X}) \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_{|X}}\ (fetch)$$

$$\frac{(\forall i)\ \mathtt{e}_i \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_i}{\mathtt{e}_1 \oplus \mathtt{e}_2 \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_1 \oplus \mathbb{r}_2}\ (compose)$$

Figure 8.1: Semantics of the target language

The choice of a correct result for the sub-query $R$ in rule $(subreq)$ is arbitrary. The non-determinism of the semantics comes solely from these choices, and the well-formedness conditions in the syntax of the target language ensures the absence of execution errors (that is, it is always possible to build a derivation, for any choice of $R$ in rule $(subreq)$). In particular, in rule $(fetch)$, the result $\mathbb{R}_i$ is well-defined over the atomic query $r$.

**Definition 8.12 (Correction)** *We define the correction of a strategy* $\mathtt{s}^{R_0,I}$ *for some information packet* $\mathbb{I}$ *(with* $\mathrm{Dom}(\mathbb{I}) = I$*) by:*

$$\mathbb{I} \Vdash \mathtt{s}^{R_0,I} \iff \forall v. \forall \mathbb{R}_0. \left\{ \begin{array}{ll} \mathtt{s}^{R_0,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0 & \\ v \Vdash \mathbb{I} & \end{array} \right. \Rightarrow \quad v \Vdash \mathbb{R}_0$$

*When $\mathbb{I} = \mathbb{I}^{\emptyset}$, we allows ourselves to omit it when writing this judgement.*

Intuitively, the predicate $\mathbb{I} \Vdash \mathtt{s}^{R_0,I}$ means that we can use strategy $\mathtt{s}$ to compute a result correct for the query $R_0$ on any value of the form $v = (v_1, v_2)$, provided we already have the information packet $\mathbb{I}$ (correct for $v$). Since the packet $\mathbb{I}^{\emptyset}$ is correct for every value $v$, the predicate $\Vdash \mathtt{s}^{R_0,I^{\emptyset}}$ means that we can use strategy $\mathtt{s}$ to implement the calculation of a result correct for $R_0$ (for pair values).

### 8.3.6 Static information

**Definition 8.13** *A* **static information packet** $\mathtt{I}$ *is a pair of static query results* $(\mathtt{R}_1, \mathtt{R}_2)$*. We set* $Dom(\mathtt{I}) = (Dom(\mathtt{R}_1), Dom(\mathtt{R}_2))$*.*

We write $\mathtt{I}^{\emptyset}$ for the static information packet $(\mathtt{R}^{\emptyset}, \mathtt{R}^{\emptyset})$. If $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$ is a static information packet, we write $\mathtt{I} \cup (1 : \mathtt{R}) = (\mathtt{R}_1 \cup \mathtt{R}, \mathtt{R}_2)$ and $\mathtt{I} \cup (2 : \mathtt{R}) = (\mathtt{R}_1, \mathtt{R}_2 \cup \mathtt{R})$.

For some information packet $\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2)$, we denote by $|\mathbb{I}|$ the static information packet $(|\mathbb{R}_1|, |\mathbb{R}_2|)$.

**Lemma 8.14** *We have:*

$$|\mathbb{I} \cup (i : \mathbb{R})| = |\mathbb{I}| \cup (i : |\mathbb{R}|)$$

A result $\mathtt{r}$ correct for a value $v$ on a good triplet $(p, t_0, X)$ gives some information about the type of $v$. Indeed, if the result is $\Omega$, the value is in $\neg \lbag p \rbag$, otherwise the value is in $\lbag p \rbag$. We see that it only depends on the static result $|\mathtt{r}|$. In fact, this is only true if the value is in $t_0$, because otherwise the correct result can be anything.

**Definition 8.15** *For a good triplet $(p, t_0, X)$ and a static atomic result $\mathtt{r}$, we write:*

$$\lbag \mathtt{r} : (p, t_0, X) \rbag = \begin{cases} \lbag p \rbag \vee \neg t_0 & \text{if } \mathtt{r} = \checkmark \\ (\neg \lbag p \rbag) \vee \neg t_0 & \text{if } \mathtt{r} = \Omega \end{cases}$$

*for an atomic query $r = (q, t_0, X)$:*

$$\lbag \mathtt{r} : r \rbag = \lbag \mathtt{r} : \sigma(r) \rbag$$

*The type associated with the static result $\mathtt{R} = (\mathtt{r}_r)_{r \in R}$ of a query $R$ is defined by:*

$$\lbag \mathtt{R} \rbag = \bigwedge_{r \in R} \lbag \mathtt{r}_r : r \rbag$$

*The type associated with the static packet $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$ is defined by:*

$$\lbag \mathtt{I} \rbag = \lbag \mathtt{R}_1 \rbag \times \lbag \mathtt{R}_2 \rbag$$

**Remark 8.16** *The types $\lbag \mathtt{R} \rbag$ are in $\sqsupset$. This is not necessarily the case for the types $\lbag \mathtt{I} \rbag$ (socles are not stable by constructor $\times$).*

**Lemma 8.17**

$$\begin{array}{lcl} (v \Vdash \mathtt{r} : (p, t_0, X)) & \Longrightarrow & v \in [\![ \zeta | \mathtt{r} : (p, t_0, X) | \rbag ]\!] \\ (v \Vdash \mathtt{r} : r) & \Longrightarrow & v \in [\![ \zeta | \mathtt{r} : r | \rbag ]\!] \\ (v \Vdash \mathbb{R}) & \Longrightarrow & v \in [\![ \zeta | \mathbb{R} | \rbag ]\!] \\ ((v_1, v_2) \Vdash \mathbb{I}) & \Longrightarrow & (v_1, v_2) \in [\![ \zeta | \mathbb{I} | \rbag ]\!] \end{array}$$

A static information $t_0$ on a value $v = (v_1, v_2)$ gives information over $v_1$ and $v_2$, that is, the types $\pi_1[t_0]$ and $\pi_2[t_0]$. If we take into account a static information packet $\mathtt{I}$, we can refine the information $t_0$ into $t'_0 = t_0 \wedge \lceil \mathtt{I} \rfloor$. This type is not necessarily in $\beth$ (Remark 8.16). However, the projections $\pi_1[t'_0]$ and $\pi_2[t'_0]$ can be represented by types in $\beth$.

**Lemma 8.18** *If $t_0 \in \beth$, $\mathtt{I}$ is a static information packet, and $i = 1..2$, then we can compute a type $\pi_i[t_0; \mathtt{I}] \in \beth$ such that $\pi_i[t_0 \wedge \lceil \mathtt{I} \rfloor] \simeq \pi_i[t_0; \mathtt{I}]$*

> *Proof*: Let us take, for example, $i = 1$. We can assume $t_0 \leq \mathbb{1}_{\mathbf{prod}}$ (because $\mathbb{1}_{\mathbf{prod}} \in \beth$). We write:
>
> $$t_0 \simeq \bigvee_{(t_1,t_2)\in\pi(t_0)} t_1 \times t_2$$
>
> If $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$, and if we write $t'_i = \lceil \mathtt{R}_i \rfloor$, then $\lceil \mathtt{I} \rfloor = t'_1 \times t'_2$. We have $t'_i \in \beth$, and:
>
> $$t'_0 := t_0 \wedge \lceil \mathtt{I} \rfloor \simeq \bigvee_{(t_1,t_2)\in\pi(t_0)} (t_1 \wedge t'_1) \times (t_2 \wedge t'_2)$$
>
> Hence:
>
> $$\pi_1[t'_0] \simeq \left( \bigvee_{(t_1,t_2)\in\pi(t_0) \mid (*)} t_1 \right) \wedge t'_1 \quad \in \beth$$
>
> where $(*)$ is the condition $t_2 \wedge t'_2 \not\simeq \mathbb{0}$. □

**Remark 8.19** *We easily find that $\pi_i[t_0; \mathtt{I}]$ is a sub-type (sometimes strict) of $\pi_i[t_0] \wedge \lceil \mathtt{R}_i \rfloor$, where $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$.*

### 8.3.7 Strategy development

We study how to produce correct strategies for a given query. First of all, let us consider the construction of target expressions.

Figure 8.2 gives a method to compute, given a good triplet $(p, t_0, X)$ and a static information packet $\mathtt{I}$, a target expression $\mathsf{e}^{X,I}$ that computes a result correct for $(p, t_0, X)$. The figure defines a judgement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathsf{e}^{X,I}$ where $I = \mathrm{Dom}(\mathtt{I})$. It uses an auxiliary judgement $\mathtt{I} \vdash_i (q, t_0, X) \Rightarrow \mathsf{e}^{X,I}$.

Before we state and prove the correction of the expressions produced by this judgment, let us comment on the rules. For a good triplet $(p, t_0, X)$, the type $\lceil \mathtt{I} \rfloor \wedge t_0$ is the static information under which we have to compute. If that type does not intersect with $\lceil p \rfloor$, then the result $\Omega$ is correct (if the value is in $\lceil \mathtt{I} \rfloor \wedge t_0$, it is the real result, and otherwise any result is correct). Rules (*first*) and (*second*) treat the case of a disjunctive pattern $p_1 | p_2$. To apply them, we have to *a priori* know (that is, by knowing $\mathtt{I}$) whether the first pattern succeeds or fails. The two (*factor*) rules treat in particular the case of patterns $x \& p$ and $(x := c) \& p$, when $x \in X$ (we then use (*skip*) when we have eliminated $x$ from $X$). They are more general because they can factorize capture variables (or binders $(x := c)$). Rule (*prod*) treats the case of a pattern $(q_1, q_2)$ We use the auxiliary judgement $\vdash_i$ on each side, while propagating the static information

$$\frac{t_0 \wedge \lceil \mathtt{I} \rfloor \wedge \lceil p \rfloor \simeq \mathbb{0}}{\mathtt{I} \vdash (p, t_0, X) \Rightarrow \Omega}\ (fail)$$

$$\frac{\mathtt{I} \vdash (p_1, t_0, X) \Rightarrow \mathtt{e} \quad t_0 \wedge \lceil \mathtt{I} \rfloor \leq \lceil p_1 \rfloor}{\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}}\ (first)$$

$$\frac{\mathtt{I} \vdash (p_2, t_0, X) \Rightarrow \mathtt{e} \quad t_0 \wedge \lceil \mathtt{I} \rfloor \wedge \lceil p_1 \rfloor \simeq \mathbb{0}}{\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}}\ (second)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \wedge \lceil \mathtt{I} \rfloor} x \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\}) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash (p, t_0, X) \Rightarrow x \oplus \mathtt{e}}\ (factor)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \wedge \lceil \mathtt{I} \rfloor} (x := c) \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\}) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash (p, t_0, X) \Rightarrow \{x \mapsto c\} \oplus \mathtt{e}}\ (factor)$$

$$\frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash (x \& p, t_0, X) \Rightarrow \mathtt{e}}\ (skip) \qquad \frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash ((x := c) \& p, t_0, X) \Rightarrow \mathtt{e}}\ (skip)$$

$$\frac{(\forall i = 1..2)\ \mathtt{I} \vdash_i (q_i, \pi_i[t_0; \lceil \mathtt{I} \rfloor], X \cap \mathrm{Var}(q_i)) \Rightarrow \mathtt{e}_i}{\mathtt{I} \vdash ((q_1, q_2), t_0, X) \Rightarrow \mathtt{e}_1 \oplus \mathtt{e}_2}\ (prod)$$

$$\frac{t_0 \leq \lceil q \rfloor}{\mathtt{I} \vdash_i (q, t_0, \emptyset) \Rightarrow \{\}}\ (succeed)$$

$$\frac{r = (q, t_0', X') \in \mathrm{Dom}(\mathtt{R}_i) \quad t_0 \leq t_0' \quad X \subseteq X'}{(\mathtt{R}_1, \mathtt{R}_2) \vdash_i (q, t_0, X) \Rightarrow (i, r_{|X})}\ (fetch)$$

Figure 8.2: Production of target expressions

on each component. This judgement $\vdash_i$ can output $\{\}$ for an atomic query $(q, t_0, X)$ if $q$ necessarily succeeds for any value in $t_0$, and if, in addition, we have $X = \emptyset$ (which may be a consequence of the use of the $(factor)$ rules). If one of these conditions is not true, we have to use one of the atomic sub-queries $(q', t_0', X')$ previously evaluated over $v_i$. We allow ourselves to have $X \subseteq X'$ and $t_0 \leq t_0'$; this means that the sub-query that we are using can do more work than necessary for $(q, t_0, X)$, but that does not matter.

The following theorem states the correction of target expressions $\mathtt{e}$ produced by the judgement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$.

**Theorem 8.20** *For every value $v = (v_1, v_2)$ and every information packet $\mathbb{I}$, we have the implication:*

$$\begin{cases} v \Vdash \mathbb{I} \\ |\mathbb{I}| \vdash (p, t_0, X) \Rightarrow \mathtt{e} \\ \mathtt{e} \overset{v, \mathbb{I}}{\rightsquigarrow} \mathbb{r} \end{cases} \implies v \Vdash \mathbb{r} : (p, t_0, X)$$

*Proof*: Let $v = (v_1, v_2)$ and $\mathbb{I}$ such that $v \Vdash \mathbb{I}$. Writing $\mathtt{I} = |\mathbb{I}|$. Hence we have $v \in [\![\lceil \mathtt{I} \rfloor]\!]$.

Then we reason by induction on the derivation of $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$. If $v \notin [\![t_0]\!]$, the assertion $v \Vdash \mathtt{r} : (p, t_0, X)$ comes immediately. So we assume $v \in [\![t_0]\!]$, and we are going to prove that $\mathtt{e} \overset{v,\mathbb{I}}{\rightsquigarrow} (v/p)_{|X}$ (which proves the result, since the semantics of the target expressions is deterministic).

Rule (*fail*): since the value $v$ is in $\wr \mathtt{I} \int \wedge t_0$, it is not in $\wr p \int$, therefore $v/p = \Omega$.

Rule (*first*): according to the premise, we obtain that $v$ is in $\wr p_1 \int$, therefore $v/p_1|p_2 = v/p_1$, which allows us to conclude by using the induction hypothesis.

Rule (*second*): we obtain that $v$ is not in $\wr p_1 \int$, therefore $v/p_1|p_2 = v/p_2$.

Rules (*factor*): as an example, let us treat the case $p \overset{t_0 \wedge \wr \mathbb{I} \int}{\longrightarrow} x$. We then have $(v/p)_{|X} = \{x \mapsto v\} \oplus (v/p)_{|X \setminus \{x\}}$.

Rules (*skip*): If $x \notin X$, we have $(v/x\&p)_{|X} = (v/(x := c)\&p)_{|X} = (v/p)_{|X}$.

Rule (*prod*): we have $(v/(q_1, q_2))_{|X} = (v_1/q_1)_{|X \cap \mathrm{Var}(q_1)} \oplus (v_2/q_2)_{|X \cap \mathrm{Var}(q_2)}$, and then we know that that $v_i$ is in $\pi_i[t_0; \wr \mathtt{I} \int]$.

Finally we only have to see that if $\mathtt{I} \vdash_i (q, t_0, X) \Rightarrow \mathtt{e}$ with $v_i$ in $t_0$, then $\mathtt{e} \overset{v,\mathbb{I}}{\rightsquigarrow} (v_i/\sigma(q))_{|X}$.

Rule (*succeed*): We have $\mathtt{e} = \{\}$ and $X = \emptyset$. However $v_i/\sigma(q)$ is not $\Omega$, because $v_i$ is in $t_0 \leq \wr q \int$. Therefore $(v_i/\sigma(q))_{|X} = \{\}$.

Rule (*fetch*): Assume $\mathtt{e} = (i, r_{|X})$ with $r = (q', t_0', X') \in \mathrm{Dom}(\mathtt{R}_i)$, $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$, and $\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2)$. We then have $\mathtt{e} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathtt{r}_{|X}$ where $\mathtt{r} = (\mathbb{R}_i)_r$. But we assumed that $v \Vdash \mathbb{I}$, which implies $v_i \Vdash \mathtt{r} : r$. Now $v$ is in $t_0$, therefore it is *a fortiori* in $t_0'$, hence we obtain $\mathtt{r} = (v_i/\sigma(q))_{|X'}$. But $X \subseteq X'$, therefore $\mathtt{r}_{|X} = (v_i/\sigma(q))_{|X}$. □

We now consider the construction of correct strategies for a given query $R_0$. Figure 8.3 defines a judgement $\mathtt{I} \vdash R_0 \Rightarrow \mathtt{s}^{R_0, I}$ where $I = \mathrm{Dom}(\mathtt{I})$.

$$\frac{(\forall r \in R_0)\ \mathtt{I} \vdash \sigma(r) \Rightarrow \mathtt{e}_r}{\mathtt{I} \vdash R_0 \Rightarrow (\mathtt{e}_r)_{r \in R_0}}\ (\mathit{assemble})$$

$$\frac{(\forall \mathtt{R} : R)\ \mathtt{I} \cup \mathtt{I}(i, \mathtt{R}) \vdash R_0 \Rightarrow \mathtt{s}_\mathtt{R}}{\mathtt{I} \vdash R_0 \Rightarrow (i, R) \mapsto (\mathtt{s}_\mathtt{R})_{\mathtt{R}:R}}\ (\mathit{subreq})$$

Figure 8.3: Strategy production

There are two ways to build a strategy. Either there is enough information in $\mathtt{I}$ to compute a correct result for all atomic queries in $R_0$, and in this case we can apply the (*assemble*) rule, or we have to apply a sub-query $R$ (to be chosen) to $v_i$ which allows us to obtain more information (rule (*subreq*)), depending on the static result that emerges.

**Theorem 8.21** *If* $\mathtt{I} \vdash R_0 \Rightarrow \mathtt{s}^{R_0, I}$*, then, for any information packet* $\mathbb{I}$ *such*

*that* $\mathtt{I} = |\mathbb{I}|$*, we have:*

$$\mathbb{I} \Vdash \mathtt{s}^{R_0,I}$$

*In particular, if* $\mathtt{I}^{\emptyset} \vdash R_0 \Rightarrow \mathtt{s}^{R_0,I}$*, then:*

$$\Vdash \mathtt{s}^{R_0,I}$$

> *Proof*: By induction on the derivation of $\mathtt{I} \vdash R_0 \Rightarrow \mathtt{s}^{R_0,I}$. Let $v = (v_1, v_2)$, and $\mathbb{I}$ such that $\mathtt{I} = |\mathbb{I}|$ and $v \Vdash \mathbb{I}$. We just have to see that if $\mathtt{s}^{R_0,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0$, then $v \Vdash R_0$.
>
> Rule (*assemble*): The semantics gives $\mathbb{R}_0 = (\mathbb{r}_r)_{r \in R_0}$, with $\mathtt{e}_r \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_r$ where $\mathtt{I} \vdash \sigma(r) \Rightarrow \mathtt{e}_r$ (for every $r \in R_0$). Theorem 8.20 gives $v \Vdash \mathbb{r}_r : r$, therefore $v \Vdash \mathbb{R}_0$.
>
> Rule (*subreq*): The semantics gives $\mathtt{s}_{|\mathbb{R}|} \overset{v,\mathbb{I}\cup(i:\mathbb{R})}{\rightsquigarrow} \mathbb{R}_0$ for a certain result $v_i \Vdash \mathbb{R}$ with $\mathrm{Dom}(\mathbb{R}) = R$. Lemma 8.11 gives $v \Vdash \mathbb{I} \cup (i : \mathbb{R})$. Let us take $\mathtt{R} = |\mathbb{R}|$. We find that $|\mathbb{I} \cup (i : \mathbb{R})| = \mathtt{I} \cup (i : \mathtt{R})$, and the premise of (*subreq*) which corresponds to $\mathtt{R}$ does give, by induction: $v \Vdash \mathbb{R}_0$. □

**Strategy development** The systems defined in figures 8.2 and 8.3 do not give actual constructions for the target expressions or the strategies. A strategy development scheme is defined by:

— the way to build a target expression for some given static information packet and good triplet $(p, t_0, X)$, when possible;
— the way to choose the sub-query $(i, R)$ to be applied when it is not possible to build a target expression for all the atomic queries of $R$ with the given static information packet.

The way to build a target expression when possible is to specify a way to build a derivation for the judgement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$. A natural approach is to apply the rules $(fail)$, $(factor)$ and (*succeed*) as soon as possible.

We will now present a tool that makes it possible to choose the sub-queries to apply. To do this, we will make explicit the obstacles that prevent us from building a target expression for a good triplet $(p, t_0, X)$, and show how a particular choice of sub-queries removes these obstructions. Consider the judgement $\mathtt{I} \vdash (p, t_0, X)$ defined in Figure 8.4. The main element is rule (*daemon*), which replaces the $(fetch)$ rule when it cannot be applied.

**Lemma 8.22** *If* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$ *and* $t_0' \wedge \wr\, \mathtt{I} \int \leq t_0$*, then* $\mathtt{I} \vdash (p, t_0', X) \Rightarrow \mathtt{e}$.

> *Proof*: By induction on the derivation of $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$. □

**Lemma 8.23** *Consider a derivation of* $\mathtt{I} \vdash (p, t_0, X)$ *that does not use rule* (*daemon*)*. Then:*

— *either* $t_0 \wedge \wr\, \mathtt{I} \int \leq \wr p \int$*, or* $t_0 \wedge \wr\, \mathtt{I} \int \wedge \wr\, p \int \simeq \mathbb{0}$*;*
— *we can compute a target expression* $\mathtt{e}$ *such that* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$.

> *Proof*: The proof is done by induction on the derivation of $\mathtt{I} \vdash (p, t_0, X)$. Rules $(fail)$, $(factor)$ and (*skip*) are immediate.
>
> Consider rule (*alt*). By induction, we build two expressions $\mathtt{e}_i$ such that $\mathtt{I} \vdash (p_1, t_0, X) \Rightarrow \mathtt{e}_1$ and $\mathtt{I} \vdash (p_2, t_0 \setminus \wr\, p_1 \int, X) \Rightarrow \mathtt{e}_2$.

$$\frac{t_0 \wedge \lfloor \mathtt{I} \rfloor \wedge \lfloor p \rfloor \simeq \mathbb{0}}{\mathtt{I} \vdash (p, t_0, X)}\ (fail)$$

$$\frac{\mathtt{I} \vdash (p_1, t_0, X) \quad \mathtt{I} \vdash (p_2, t_0 \setminus \lfloor p_1 \rfloor, X)}{\mathtt{I} \vdash (p_1 | p_2, t_0, X)}\ (alt)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \wedge \lfloor \mathtt{I} \rfloor} x \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\})}{\mathtt{I} \vdash (p, t_0, X)}\ (factor)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \wedge \lfloor \mathtt{I} \rfloor} (x := c) \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\})}{\mathtt{I} \vdash (p, t_0, X)}\ (factor)$$

$$\frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X)}{\mathtt{I} \vdash (x \& p, t_0, X)}\ (skip) \qquad \frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X)}{\mathtt{I} \vdash ((x := c) \& p, t_0, X)}\ (skip)$$

$$\frac{(\forall i = 1..2)\ \mathtt{I} \vdash_i (q_i, \pi_i[t_0; \lfloor \mathtt{I} \rfloor], X \cap \mathrm{Var}(q_i))}{\mathtt{I} \vdash ((q_1, q_2), t_0, X)}\ (prod)$$

$$\frac{t_0 \leq \lfloor q \rfloor}{\mathtt{I} \vdash_i (q, t_0, \emptyset)}\ (succeed)$$

$$\frac{r = (q, t_0', X') \in \mathrm{Dom}(\mathtt{R}_i) \quad t_0 \leq t_0' \quad X \subseteq X'}{(\mathtt{R}_1, \mathtt{R}_2) \vdash_i (q, t_0, X)}\ (fetch)$$

$$\frac{}{\mathtt{I} \vdash_i (q, t_0, X)}\ (daemon)$$

Figure 8.4: Target expression production skeleton

If $t_0 \wedge \lfloor \mathtt{I} \rfloor \leq \lfloor p_1 \rfloor$, then $t_0 \wedge \lfloor \mathtt{I} \rfloor \leq \lfloor p_1 | p_2 \rfloor$, therefore $\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}_1$. Otherwise, we have $t_0 \wedge \lfloor \mathtt{I} \rfloor \wedge \lfloor p_1 \rfloor \simeq \mathbb{0}$, therefore $t_0 \wedge \lfloor \mathtt{I} \rfloor \leq t_0 \setminus \lfloor p_1 \rfloor$. Lemma 8.22 then gives $\mathtt{I} \vdash (p_2, t_0, X) \Rightarrow \mathtt{e}_2$, and rule (*second*) can be applied: $\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}_2$.

Consider rule (*prod*), and take $t_i = \pi_i[t_0; \mathtt{I}] \simeq \pi_i[t_0 \wedge \lfloor \mathtt{I} \rfloor]$. First, we find that if $\mathtt{I} \vdash_i (q_i, t_i, X_i)$ with one of the rules (*succeed*) or $(fetch)$ (but not (*daemon*)), then either $t_i \wedge \lfloor q_i \rfloor \simeq \mathbb{0}$, or $t_i \leq \lfloor q_i \rfloor$. It is trivial for rule (*succeed*). For rule $(fetch)$, we take $r = (q_i, t_i', X_i') \in \mathrm{Dom}(\mathtt{R}_i)$ such that $t_i' \leq t_i'$, $X_i \subseteq X_i'$. If $(\mathtt{R}_i)_r = \checkmark$, then $\lfloor \mathtt{R}_i \rfloor \leq \lfloor q_i \rfloor \vee \neg t_i'$; or $t_i \leq \lfloor \mathtt{R}_i \rfloor$, therefore $t_i \leq \lfloor q_i \rfloor$. If $(\mathtt{R}_i)_r = \Omega$, then $\lfloor \mathtt{R}_i \rfloor \leq (\neg \lfloor q_i \rfloor) \vee \neg t_i'$, therefore $t_i \leq \neg \lfloor q_i \rfloor$, hence finally $t_i \wedge \lfloor q_i \rfloor \simeq \mathbb{0}$.

We have $t_0 \wedge \lfloor \mathtt{I} \rfloor \leq t_1 \times t_2$ and $\lfloor (q_1, q_2) \rfloor \simeq \lfloor q_1 \rfloor \times \lfloor q_2 \rfloor$. Therefore if $t_1 \leq \lfloor q_1 \rfloor$ and $t_2 \leq \lfloor q_2 \rfloor$, then $t_0 \wedge \lfloor \mathtt{I} \rfloor \leq \lfloor (q_1, q_2) \rfloor$. Otherwise, for example if $t_1 \leq \neg \lfloor q_1 \rfloor$, then $t_0 \wedge \lfloor \mathtt{I} \rfloor \wedge \lfloor (q_1, q_2) \rfloor \leq ((\neg \lfloor q_1 \rfloor) \times t_2) \wedge (\lfloor q_1 \rfloor \times \lfloor q_2 \rfloor) \simeq \mathbb{0}$. □

**Remark 8.24** *The converse result is also true: if* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$*, then we can build a derivation of* $\mathtt{I} \vdash (p, t_0, X)$ *without using rule* (*daemon*)*. We mostly have to just erase the* $\Rightarrow \mathtt{e}$ *in the derivation of* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$*. Each*

*instance of rule* $(first)$ *or* $(second)$ *is turned into an instance of rule* $(alt)$*, and the missing premise is obtained with rule* $(fail)$*.*

**Lemma 8.25** *For every good triplet* $(p, t_0, X)$ *and every static information packet* $\mathtt{I}$*, we can build we can build a derivation of* $\mathtt{I} \vdash (p, t_0, X)$*.*

Indeed, there is no "obstruction" thanks to rule (*daemon*). As for the judgement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$, a natural choice to build a derivation of $\mathtt{I} \vdash (p, t_0, X)$ consists in applying rules $(fail)$,$(factor)$ and (*succeed*) as much as possible.

To build a strategy for $R_0$ from some static information packet $\mathtt{I}$, we start by building derivations for the $\sigma(r)$ (for $r \in R_0$), with as few rules (*daemon*) as possible (this is not a hard constraint: it does not matter if we "miss" some rules that would prevent us from using (*daemon*), such as factorization, it will just produce a less efficient strategy). If we can do this without ever using the (*daemon*) rule, we have succeeded, and we can build a strategy with rule (*assemble*), that is, without doing any sub-queries. We just have to apply Lemma 8.23 to build a target expression for each atomic query of $R_0$.

Otherwise, we need to choose $i = 1..2$ and a query $R$ to be applied over $v_i$. Let us look at what happens to the instances of (*daemon*) if we work not with $\mathtt{I}$, but with $\mathtt{I}' = \mathtt{I} \cup (i : \mathtt{R})$ instead, where $\mathtt{R}$ is a static result for $R$. We refine the static information: $\wr\mathtt{I}'\int \leq \wr\mathtt{I}\int$. We see that the derivations of the $\mathtt{I} \vdash \sigma(r)$ trivially give derivations for $\mathtt{I}' \vdash \sigma(r)$. In fact, we can replace a (*daemon*) rule proving $\mathtt{I}' \vdash_i (q, t_0, X)$ by an instance of $(fetch)$, as long as the atomic query $(q, t_0, X)$ is in $R$ (or if a query $(q', t'_0, X')$ is in $R$ with the conditions of rule $(fetch)$). Indeed, if $\mathtt{I} = (\mathtt{R}'_1, \mathtt{R}'_2)$, we have: $\mathrm{Dom}(\mathtt{R}'_i) = \mathrm{Dom}(\mathtt{R}_i) \cup R$.

But the more accurate static information $\mathtt{I}'$ makes it possible to simplify even more the derivations of the $\mathtt{I} \vdash \sigma(r)$, by allowing more uses of the rules $(factor)$, $(fail)$, (*succeed*). In general, among the $2^n$ static results $\mathtt{R}$ (where $n$ is the cardinal of $R$), there are many which are "impossible", i.e. such that $\wr\mathtt{I}'\int \simeq \mathbb{0}$, making it possible to immediately apply the rule $(fail)$ for all the atomic queries of $R_0$.

In any case, if we choose $i$ and $R$ such that at least one of the (*daemon*) rules can be turned into $(fetch)$, we will be able to reduce the number (*daemon*) rules used. By induction on this number, we can build a strategy for $R_0$.

The following section gives examples of constructions.

### 8.3.8 Examples of constructions

**Left-right strategy** In a left-right strategy, we can only apply one query on each sub-tree (left and right), and we have to start with the left sub-tree (of course, the symmetrical choice is possible). This implements the ideas of paragraphs "Top-down determinization" and "Lateral Propagation" from Section 8.2. Since we only visit each sub-tree once, we do not have to use *memoization*.

Left-right strategies have the form $(1, R_1) \mapsto (\mathtt{s}_\mathtt{R})_{\mathtt{R}:R_1}$ with $\mathtt{s}_\mathtt{R} = (2, R_2(\mathtt{R})) \mapsto ((\mathtt{e}_{\mathtt{R},\mathtt{R}',r})_{r\in R_0})_{\mathtt{R}':R_2(\mathtt{R})}$, where $R_1$ is the query to be computed over $v_1$ and $R_2(\mathtt{R})$ is the query to be computed over $v_2$ (it can depend on the result of $R_1$).

To build $\mathtt{s}$, we start from a derivation $\mathtt{I}^{\emptyset} \vdash \sigma(r)$ for each $r \in R_0$. We consider all the (*daemon*) rules used in this derivation, with $i = 1$. We take as $R_1$ the set of all corresponding atomic queries $r$. For each $\mathtt{R} : R_1$, we simplify the derivation obtained. All the (*daemon*) rules that remain are of the form

$\mathtt{I}' \vdash_2 r$. We take for $R_2(\mathtt{R})$ the set of these atomic queries $r$. For each $\mathtt{R}' : R_2(\mathtt{R})$, after simplification, we obtain derivations that do not use rule (*daemon*), and we can then build the target expressions.

**Iterative strategy** An iterative strategy consists in only applying sub-queries that are made of a single atomic query. Given derivations of $\mathtt{I} \vdash \sigma(r)$ for the $r \in R_0$, we choose an instance of rule (*daemon*), i.e. $\mathtt{I} \vdash_i r$, and we build strategy $(i, \{r\}) \mapsto (\mathtt{s}_\mathtt{R})$. Strategies $\mathtt{s}_\mathtt{R}$ are built the same, starting from $\mathtt{I} \cup (i : \mathtt{R})$, after simplifying the derivation, and so on. At each step, we decrease by at least 1 the number of (*daemon*) rules used, which ensures the termination of the process.

A natural choice to find rule (*daemon*) consists in searching it first in the derivation for $p_1$, before the one for $p_2$, in rule (*alt*). Indeed, if $p_1$ succeeds, the static information will be enough to ignore $p_2$, while the converse is not true (of course, if $\wr p_1 \int \wedge \wr\, p_2 \int \simeq \mathbb{0}$, the argument does not hold).

**A concrete example** Let us give a concrete example of a construction strategy. This example does not involve capture variables, which simplifies things a bit (in particular, you can always do away with the $(fetch)$ rule, and use only $(fail)$ and (*succeed*)). Let us take $t_0 = (t_1 \times t_2) \vee (\mathbb{1} \times t_3)$ where $t_2 \wedge t_3 \simeq \mathbb{0}$, and $q$ such that $\sigma(q) = (q_1, q_2)$, with $\mathrm{Var}(q_i) = \mathbb{0}$ and $\wr q_i \int = t_i$ (we can take $\sigma(q_i) = t_i$ if the $t_i$ are base types, and otherwise, we can apply the normal form decomposition from Section 6.5.2). We study the query $R_0 = \{(q, t_0, \emptyset)\}$.

Here is a derivation obtained with the rules of Figure 8.4:

$$\dfrac{\dfrac{}{\mathtt{I}^\emptyset \vdash_1 (q_1, \mathbb{1}, \emptyset)}\,(daemon) \quad \dfrac{}{\mathtt{I}^\emptyset \vdash_2 (q_2, t_2 \vee t_3, \emptyset)}\,(daemon)}{\mathtt{I}^\emptyset \vdash ((q_1, q_2), t_0, \emptyset)}\,(prod)$$

We are forced to use the (*daemon*) rules, because there are no queries in the initially empty static information packet, which prevents the use of the rule $(fetch)$, while the (*succeed*) rule does not apply either. To continue building the strategy, we now have to choose between applying query $R_1 = \{(q_1, \mathbb{1}, \emptyset)\}$ to the left sub-tree, and applying query $R_2 = \{(q_2, t_2 \vee t_3, \emptyset)\}$ to the right sub-tree.

Let us look at both possibilities one after the other. We choose to start the strategy with $(1, R_1) \mapsto (\ldots)$. We have to complete the $\ldots$. There are two possible static results $\mathtt{R}_1$ for $R_1$, which correspond to the success or failure of its unique atomic query $r_1$. In case of failure: $\mathtt{R}_1 = \{r_1 \mapsto \Omega\}$. We then have: $\wr \mathtt{I}(1, \mathtt{R}_1) \int = (\neg t_1 \mid \neg \mathbb{1}) \times \mathbb{1} \simeq (\neg t_1) \times \mathbb{1}$. This allows us to directly use the rule $(fail)$:

$$\dfrac{t_0 \wedge \wr\, \mathtt{I}(1, \mathtt{R}_1) \int \wedge \wr\, (q_1, q_2) \int \simeq t_0 \wedge ((\neg t_1) \times \mathbb{1}) \wedge (t_1 \times t_2) \simeq \mathbb{0}}{\mathtt{I}(1, \mathtt{R}_1) \vdash ((q_1, q_2), t_0, \emptyset)}\,(fail)$$

and we can use this derivation (with the rules of Figure 8.2) to obtain the target expression $\Omega$.

In case of success of $r_1$, that is, if $\mathtt{R}_1 = \{r_1 \mapsto \checkmark\}$, we obtain $\wr \mathtt{I}(1, \mathtt{R}_1) \int = (t_1 \mid \neg \mathbb{1}) \times \mathbb{1} \simeq t_1 \times \mathbb{1}$. An instance of rule (*daemon*) can then be eliminated, but the second one necessarily remains:

$$\dfrac{\dfrac{t_1 \leq \wr q_1 \int}{\mathtt{I}(1, \mathtt{R}_1) \vdash_1 (q_1, t_1, \emptyset)}\,(succeed) \quad \dfrac{}{\mathtt{I}(1, \mathtt{R}_1) \vdash_2 (q_2, t_2 \vee t_3, \emptyset)}\,(daemon)}{\mathtt{I}(1, \mathtt{R}_1) \vdash ((q_1, q_2), t_0, \emptyset)}\,(prod)$$

We then apply the query $R_2$ on the right sub-tree, i.e. we continue the strategy with $(2, R_2) \mapsto (\ldots)$. We once again have to distinguish between the success or failure of the unique atomic query $r_2$ of $R_2$. If $\mathtt{R}_2 = \{r_2 \mapsto \Omega\}$, we obtain $\lceil \mathtt{R}_2 \rfloor = (\neg t_2) \mathbf{\vee} \neg (t_2 \mathbf{\vee} t_3) \simeq \neg t_2$, therefore, by writing $\mathtt{I}_2 = \mathtt{I}(1, \mathtt{R}_1) \cup \mathtt{I}(2, \mathtt{R}_2)$: $\lceil \mathtt{I}_2 \rfloor = t_1 \mathbf{\times} (\neg t_2)$, which allows us to directly use the $(fail)$ rule:

$$\frac{t_0 \mathbf{\wedge} \lceil \mathtt{I}_2 \rfloor \mathbf{\wedge} \lceil (q_1, q_2) \rfloor \simeq t_0 \mathbf{\wedge} (t_1 \mathbf{\times} (\neg t_2)) \mathbf{\wedge} (t_1 \mathbf{\times} t_2) \simeq \mathbb{0}}{\mathtt{I}_2 \vdash ((q_1, q_2), t_0, \emptyset)} \ (fail)$$

If $\mathtt{R}_2 = \{r_2 \mapsto \checkmark\}$, we obtain $\lceil \mathtt{R}_2 \rfloor = t_2 \mathbf{\vee} \neg (t_2 \mathbf{\vee} t_3) \simeq \neg t_3$, therefore, by writing $\mathtt{I}_2 = \mathtt{I}(1, \mathtt{R}_1) \cup \mathtt{I}(2, \mathtt{R}_2)$: $\lceil \mathtt{I}_2 \rfloor = t_1 \mathbf{\times} (\neg t_3)$, which allows us to replace $(daemon)$ by $(succeed)$:

$$\frac{\dfrac{t_1 \leq \lceil q_1 \rfloor}{\mathtt{I}_2 \vdash_1 (q_1, t_1, \emptyset)} \ (succeed) \quad \dfrac{t_2 \leq \lceil q_2 \rfloor}{\mathtt{I}_2 \vdash_2 (q_2, t_2, \emptyset)} \ (succeed)}{\mathtt{I}_2 \vdash ((q_1, q_2), t_0, \emptyset)} \ (prod)$$

which corresponds to the target expression $\{\} \oplus \{\}$, that is equivalent to $\{\}$.

Overall, by adopting the left-right approach, we have obtained the strategy:

$$\mathtt{s}_1 = (1, R_1) \mapsto \begin{cases} \{r_1 \mapsto \Omega\} & \mapsto & \Omega \\ \{r_1 \mapsto \checkmark\} & \mapsto & (2, R_2) \mapsto \begin{cases} \{r_2 \mapsto \Omega\} & \mapsto & \Omega \\ \{r_2 \mapsto \checkmark\} & \mapsto & \{\} \end{cases} \end{cases}$$

Now let us consider the other possible choice, namely to start with the sub-query $R_2$. In case of failure, we obtain $\lceil \mathtt{I}(2, \mathtt{R}_2) \rfloor \simeq \mathbb{1} \mathbf{\times} \neg t_2$, hence directly:

$$\frac{t_0 \mathbf{\wedge} \lceil \mathtt{I}(2, \mathtt{R}_2) \rfloor \mathbf{\wedge} \lceil (q_1, q_2) \rfloor \simeq \mathbb{0}}{\mathtt{I}(2, \mathtt{R}_2) \vdash ((q_1, q_2), t_0, \emptyset)} \ (fail)$$

In case of success, if $\mathtt{R}_2 = \{r_2 \mapsto \checkmark\}$, we obtain $\lceil \mathtt{R}_2 \rfloor \simeq \neg t_3$, therefore, $t_0 \mathbf{\wedge} \lceil \mathtt{I}(2, \mathtt{R}_2) \rfloor = t_1 \mathbf{\times} t_2$, hence we can replace the *two* $(daemon)$ rules by $(succeed)$:

$$\frac{\dfrac{t_1 \leq \lceil q_1 \rfloor}{\mathtt{I}(2, \mathtt{R}_2) \vdash_1 (q_1, t_1, \emptyset)} \ (succeed) \quad \dfrac{t_2 \leq \lceil q_2 \rfloor}{\mathtt{I}(2, \mathtt{R}_2) \vdash_2 (q_2, t_2, \emptyset)} \ (succeed)}{\mathtt{I}(2, \mathtt{R}_2) \vdash ((q_1, q_2), t_0, \emptyset)} \ (prod)$$

which again corresponds to the target expression $\{\} \oplus \{\}$, that simplifies into $\{\}$.

The strategy we have just built is:

$$\mathtt{s}_2 = (2, R_2) \mapsto \begin{cases} \{r_2 \mapsto \Omega\} & \mapsto & \Omega \\ \{r_2 \mapsto \checkmark\} & \mapsto & \{\} \end{cases}$$

The question now is what strategy should be chosen, between $\mathtt{s}_1$ and $\mathtt{s}_2$. There is no better choice *a priori*. The strategy $\mathtt{s}_2$ seems simpler, since it ignores the left sub-tree in all cases, and must anyway be used in the event of a failure of $r_1$ if we use $\mathtt{s}_1$ ($\mathtt{s}_2$ appears as a sub-strategy of $\mathtt{s}_1$). But it is not enough to say that $\mathtt{s}_2$ is necessarily better.

If the sub-query $r_1 = (q_1, \mathbb{1}, \emptyset)$ is much easier to evaluate than $r_2$ (for example if $t_1$ is a base type, and in order to distinguish between types $t_2$ and $t_3$ it is necessary to go deep into the value), and if $r_1$, does fail sometimes during

execution, then it is better to use $\mathtt{s}_1$, which will sometimes avoid evaluating $r_2$ (compensating for the fact that when $r_1$ succeeds, we have done a useless computation). Another case in which it is better to use $\mathtt{s}_1$ is when $r_1$ is only a little easier to evaluate than $r_2$, and $r_1$ fails most of the time during execution. Indeed, in this situation, choosing $\mathtt{s}_1$ makes it possible to avoid computing $r_2$ most of the time, and even if the gain is small, it is often realized. We give a simple probabilistic model to support these intuitions. Let us assume that the cost of $r_1$ is constant, denoted by $C_1$, and that the probability of success of $r_1$ is $0 \leq \alpha_1 \leq 1$. Similarly, we introduce $C_2$, the cost of $r_2$. By considering only the cost of sub-queries, the average cost of $\mathtt{s}_1$ is then $\alpha_1(C_1 + C_2) + (1 - \alpha_1)C_1 = C_1 + \alpha_1 C_2$ (if $r_1$ succeeds, $r_2$ also needs be evaluated). The average cost of $\mathtt{s}_2$ is simply $C_2$. Choosing $\mathtt{s}_1$ is therefore best as soon as:

$$C_1 \leq (1 - \alpha_1)C_2$$

which justifies the different cases of the informal discussion above. Of course, the value of $\alpha_1$ is unknown to the compiler (it depends on the values manipulated during execution), and having assumed that the cost of the sub-queries is unfounded (it also depends on the values actually encountered). This analysis, however, suggests that it is not possible to rely on a hypothetical concept of optimality to choose between the various possible strategies: it is, indeed, a heuristic choice.

**Remark 8.26** *Levin and Pierce [LP04] suggest a "maximum connection factor" heuristic, which, reworded in our formalism, would suggest choosing* $\mathtt{s}_2$ *in this example (inspired by the example presented in Figure 3 of their article).*

## 8.4 Factorization of captures

In this section, we show how to compute the predicates $p \xrightarrow{t_0} x$ and $p \xrightarrow{t_0} (x := c)$ introduced in Section 8.3.

The predicate $p \xrightarrow{t_0} (x := c)$ is the easiest, since we can simply use the typing algorithm for pattern matching. Indeed, by using Theorem 6.12, we find that:

$$p \xrightarrow{t_0} (x := c) \iff ((t\mathbf{\wedge} \lfloor p \rfloor)/p)(x) \leq b_c$$

and we have seen how to compute the environment $((t\mathbf{\wedge} \lfloor p \rfloor)/p)$.

Let us move on to the predicate $p \xrightarrow{t_0} x$. We are actually going to inductively axiomatize its negation. Consider the judgement $\vdash (p, t_0, x)$ defined by the following system of rules:

$$\frac{x \notin \mathrm{Var}(p)}{\vdash (p, t_0, x)}$$

$$\frac{\vdash (p_1, t_0 \mathbf{\wedge} \lfloor p_1 \rfloor, x)}{\vdash (p_1 | p_2, t_0, x)} \quad \frac{\vdash (p_2, t_0 \mathbf{\setminus} \lfloor p_1 \rfloor, x)}{\vdash (p_1 | p_2, t_0, x)}$$

$$\frac{\exists i.\ \vdash (\sigma(q_i), \pi_i[t_0], x)}{\vdash ((q_1, q_2), t_0, x)}\ (prod)$$

$$\frac{\forall i.\ \vdash (p_i, t_0, x)}{\vdash (p_1 \mathbf{\&} p_2, t_0, x)}$$

$$\frac{t_0 \not\leq b_c}{\vdash ((x := c), t_0, x)}$$

**Lemma 8.27** *Let $p$ be a well-formed pattern, $v \in \lfloor p \rceil$, and $x \in Var(p)$. Then the value $(v/p)(x)$, seen as a binary tree, has at most as many nodes as $v$.*

> *Proof*: By induction on the pair $(v, p)$, by following the definition of the pattern matching semantics. □

**Lemma 8.28** *Let $q_1, q_2$ be two pattern nodes, $v$ a value, and $x \in Var(q_1) \cup Var(q_2)$. Then:*

$$(v/(q_1, q_2))(x) = v \iff v = (v_1, v_2) \wedge \forall i.\ (x \in Var(q_i) \wedge v_i = (v_i/\sigma(q_i))(x))$$

> *Proof*: The implication $\Leftarrow$ is trivial. Let us prove the implication $\Rightarrow$. First of all, if $v$ is not a pair, then $(v/(q_1, q_2))(x) = \Omega$. So let us assume that $v = (v_1, v_2)$. If $x \in \mathrm{Var}(q_1) \cap \mathrm{Var}(q_2)$, we have $(v_1, v_2) = v = (v/(q_1, q_2))(x) = ((v_1/\sigma(q_1))(x), (v_1/\sigma(q_2))(x))$, therefore $v_i = (v_i/\sigma(q_i))(x)$. Let us show that it is impossible to have $x \in \mathrm{Var}(q_1) \backslash \mathrm{Var}(q_2)$, the symmetrical case being similar. If $x \in \mathrm{Var}(q_1) \backslash \mathrm{Var}(q_2)$, then $(v_1, v_2) = (v_1/\sigma(q_2))(x)$, which contradicts Lemma 8.27. □

**Theorem 8.29** *We have: $p \xrightarrow{t_0} x \iff \neg(\vdash (p, t_0 \mathbf{\wedge} \lfloor p \rceil, x))$*

> *Proof*: Even if it means replacing $t_0$ by $t_0 \mathbf{\wedge} \lfloor p \rceil$, we can obviously assume that $t_0 \leq \lfloor p \rceil$.
>
> We first show, by induction on the derivation, the implication $(\vdash (p, t_0, x)) \Rightarrow \neg(p \xrightarrow{t_0} x)$, assuming that $t_0 \leq \lfloor p \rceil$. Specifically, we associate a derivation of $\vdash (p, t_0 \mathbf{\wedge} \lfloor p \rceil, x)$ with a value $v \in [\![t_0]\!]$ such that $v/p \neq v$. Every case is easy. The (*prod*) rule is dealt with by Lemma 8.28.
>
> To prove the implication $\neg(p \xrightarrow{t_0} x) \Rightarrow (\vdash (p, t_0, x))$, we show the following assertion by induction on the pair $(v, p)$:
>
> $$(v \in [\![t_0]\!] \wedge x \in \mathrm{Var}(p) \wedge (v/p)(x) \neq x) \Rightarrow (\vdash (p, t_0, x))$$
>
> The induction follows the semantics of pattern matching. Every case is easy. The (*prod*) rule is dealt with by Lemma 8.28. □

Therefore, to know if the variable $x$ can be factorized in pattern $p$ for some input type $t_0$, we have to compute the inductive predicate defined by the rules given above. This comes down directly to the formalism of Section 7.1, and we can then use the algorithm without backtracking to decide if the judgement $\vdash (p, t_0 \mathbf{\wedge} \lfloor p \rceil, x)$ is derivable or not.

# Part III

# The CDuce language

# Chapter 9

# Records

In this chapter, we will extend the calculus of Chapter 5 with a new type, values and operations: records. We show how to adapt the definitions and the results of the previous chapters.

Let $\mathcal{L}$ be an infinite set of labels. Record values are functions from a finite set of labels to values. To simplify the discussion, we will in fact assume that the records are defined on the whole set $\mathcal{L}$ but they are constant on almost all labels (i.e. on a cofinite set). We will see in Section 9.5 how to code partial functions.

**Quasi constant functions** For any $Z$ set, a function $r : \mathcal{L} \to Z$ is called quasi constant if there is an element $z \in Z$ such that the set $\{l \in \mathcal{L} \mid f(l) \neq z\}$ is finite. The $z$ element is uniquely identified. It is denoted $\mathtt{def}(r)$. We define: $\mathrm{Dom}(r) := \{l \mid f(l) \neq \mathtt{def}(r)\}$. The set of quasi constant functions from $\mathcal{L}$ to $Z$ is denoted $\mathcal{L} \xrightarrow{c} Z$.

The notation $\{l_1 = z_1; \ldots; l_n = z_n; \_ = z\}$ refers to the function $r \in \mathcal{L} \xrightarrow{c} Z$ defined by $r(l_i) = z_i$ for $i = 1..n$ and $r(l) = z$ for $l \notin \{l_1, \ldots, l_n\}$. Any element $r \in \mathcal{L} \xrightarrow{c} Z$ can be written in this form, and one can even choose the set $\{l_1, \ldots, l_n\}$ to be arbitrarily large, but this form is not unique (it becomes unique if we require $z_i \neq z$).

If $(Z_l)_{l \in \mathcal{L}}$ is a family of subsets of $Z$, we denote $\prod_{l \in \mathcal{L}}^{c} Z_l$ to be the subset of $\mathcal{L} \xrightarrow{c} Z$ consisting of $r$ functions such that $r(l) \in Z_l$ for any label $l$.

## 9.1 Types

### 9.1.1 Algebra of types

**Atoms** We first introduce the record types in the minimal algebra defined in Section 3.3. The functor $F$ gives the signature of the constructors. We declare:

$$a \in FX ::= x \to x \mid x \times x \mid b$$

We now take:

$$a \in FX ::= x \to x \mid x \times x \mid b \mid r$$

where $r$ represents an element of $\mathcal{L} \xrightarrow{c} X$. We have a new universe of atoms **rec**, as well as **fun**, **prod**, **basic**. The set of atoms in that universe is:

$$T_{\mathbf{rec}} = \mathcal{L} \xrightarrow{c} T = \{\{l_1 = \theta_1; \ldots; l_n = \theta_n; \_ = \theta_0\} \mid \theta_i \in T\}$$

**Socle** We extend the definition of $\sqsupset$ by imposing that if $r \in T_{\mathbf{rec}}$ is a record atom present in $P \cup N$ (avec $(P, N)$ in the type of $\sqsupset$), then $\tau(r(l)) \in \sqsupset$ for all $l \in \mathcal{L}$. Putting it differently:

$$\{l_1 = \theta_1; \ldots; l_n = \theta_n; \_ = \theta_0\} \in P \cup N \Rightarrow \forall i = 0..n.\ \tau(\theta_i) \in \sqsupset$$

### 9.1.2 Models

We modify the definition of the extensional interpretation of types associated with the set-theoretic interpretation $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ (Definition 4.2). We define:

$$\mathbb{E}D := \mathcal{C} + D \times D + \mathcal{P}(D \times D_\Omega) + \mathcal{L} \xrightarrow{c} D$$

and, for $r \in \mathcal{L} \xrightarrow{c} T$:

$$\mathbb{E}[\![r]\!] = \prod_{l \in \mathcal{L}}^{c} [\![\tau(r(l))]\!] \subseteq \mathcal{L} \xrightarrow{c} D$$

We denote $\mathbb{E}_{\mathbf{rec}} D = \mathcal{L} \xrightarrow{c} D$.

In the definition of a well-founded model (Definition 4.3), we add the condition:

$$d' \in \mathcal{L} \xrightarrow{c} D \Rightarrow \forall l \in \mathcal{L}.\ d'(l) \lhd d$$

In the definition of a structural model (Definition 4.4), we require that $D$ is the initial solution to an equation of the form:

$$D = \mathcal{C} + D \times D + \mathcal{P}(D \times D_\Omega) + \mathcal{L} \xrightarrow{c} D$$

and that, for $r \in \mathcal{L} \xrightarrow{c} T$:

$$[\![r]\!] = \mathbb{E}[\![r]\!]$$

### 9.1.3 Subtyping

We modify the approach of Section 4.3. The analogue of Lemmas 4.6 and 4.9 is given by:

**Lemma 9.1** *Let $(X_i)_{i \in P}$ and $(X_i)_{i \in N}$ be two families of elements of $\mathcal{L} \xrightarrow{c}$*

$\mathcal{P}(D)$. *Let* $L \supseteq \bigcup_{i\in P\cup N} Dom(X_i)$. *Then:*

$$\bigcap_{i\in P}\prod_{l\in\mathcal{L}}^{c} X_i(l) \subseteq \bigcup_{i\in N}\prod_{l\in\mathcal{L}}^{c} X_i(l)$$

$$\Longleftrightarrow$$

$$\forall \iota : N \to L \cup \{\_\}. \begin{cases} \exists l \in L.\ \bigcap_{i\in P} X_i(l) \subseteq \bigcup_{i\in N \mid \iota(i)=l} X_i(l) & (i) \\ \vee \\ \exists i_0 \in N.\ (\iota(i_0) = \_) \wedge \bigcap_{i\in P} \mathtt{def}(X_i) \subseteq \mathtt{def}(X_{i_0}) & (ii) \\ \vee \\ \bigcap_{i\in P} \mathtt{def}(X_i) = \emptyset & (iii) \end{cases}$$

*(with the convention* $\bigcap_{i\in\emptyset}\prod_{l\in\mathcal{L}}^{c} X_i(l) = \mathcal{L} \xrightarrow{c} D$*)*

*Proof*: Lets first show the contradiction of the implication $\Rightarrow$. Lets suppose the existence of the function $\iota$ which verifies none of the three conditions $(i)$, $(ii)$, $(iii)$ of the right side. We are going to construct an element $\rho$ which invalidates the set inclusion of the left side. In $L$, the values of $\rho$ are given by $(i)$. We choose $\rho(l)$ in the set:

$$\bigcap_{i\in P} X_i(l) \setminus \bigcup_{i\in N \mid \iota(i)=l} X_i(l)$$

this guarantees already that $\rho$ is not in $\prod_{l\in\mathcal{L}}^{c} X_i(l)$ si $\iota(i) \in L$.

For each $i_0$ such that $\iota(i_0) = \_$, we choose $l_{i_0} \notin L$, all distinct, we take for $\rho(l_{i_0})$ an element of

$$\bigcap_{i\in P} \mathtt{def}(X_i) \setminus \mathtt{def}(X_{i_0})$$

which is not empty, according to $(ii)$, this guarantees that $\rho$ is not in $\prod_{l\in\mathcal{L}}^{c} X_{i_0}(l)$ if $\iota(i_0) = \_$.

We only need to complete $\rho$ on the labels $l \in \mathcal{L}\setminus L$ which are not on the form $l_{i_0}$ for $\iota(i_0) = \_$. We select an element $d$ of

$$\bigcap_{i\in P} \mathtt{def}(X_i)$$

which is not empty according to $(iii)$ and we define $\rho(l) = d$ for those labels. We have then constructed the quasi-function $\rho$, which invalidates the set inclusion of the left side.

Lets now tackle the proof of the implication $\Leftarrow$, again by contradiction. We take a function $\rho$ which invalidates the set inclusion

of the left member. For all $i_0 \in N$, we can then find a label $l_{i_0}$ such that:

$$\rho(l_{i_0}) \in \bigcap_{i \in P} X_i(l_{i_0}) \backslash X_{i_0}(l_{i_0})$$

We define a function $\iota : N \to L \cup \{\_\}$ by $\iota(i_0) = l_{i_0}$ if $l_{i_0} \in L$ and $\iota(i_0) = \_$ otherwise. Let's check that the function $\iota$ invalidates the assertions $(i)$,$(ii)$ and $(iii)$. If $(i)$ was true, we could find $l \in L$ and $i_0 \in N$ with $\iota(i_0) = l$ and $\rho(l) \in X_{i_0}(l)$ (because $\rho(l) \in \bigcap_{i \in P} X_i(l)$). But $\iota(i_0) = l$ gives $l = l_0$, and $\rho(l_{i_0}) \notin X_{i_0}(l_{i_0})$. Let's now consider $(ii)$. Let's take $i_0 \in N$ with $\iota(i_0) = \_$, i.e. $l_{i_0} \notin L$. As a consequence $\rho(l_{i_0}) = \mathtt{def}(\rho)$, and $X_i(l_{i_0}) = \mathtt{def}(X_i)$ for $i \in P \cup N$, which gives: $\mathtt{def}(\rho) \in \bigcap_{i \in P} \mathtt{def}(X_i) \backslash \mathtt{def}(X_{i_0})$, and is enough to invalidate $(ii)$. As to $(iii)$, it suffices to consider $\mathtt{def}(\rho)$, which is indeed in $\bigcap_{i \in P} \mathtt{def}(X_i)$. □

**Remark 9.2** *The result comes from the following observation. We can identify the set* $\mathcal{L} \xrightarrow{c} Z$ *with:*

$$\prod_{l \in L} Z \times \left( (\mathcal{L} \backslash L) \xrightarrow{c} Z \right)$$

*We are then brought back to a problem of inclusion between an intersection and a union of finite cartesian products, a problem that we have already met in the two of products with two elements (Lemma 4.6) and which can be easily generalized. For the last component, we must deal with the problem of the inclusion between an intersection and the union of infinite products with all identical components (by limiting oneself to quasi-constant families). This problem is quite close to the one considered in Lemma 4.8, which explains the point* $(ii)$*, except that an infinite products of empty sets is empty, whereas* $\mathcal{P}(\emptyset)$ *is not empty (which explains point* $(iii)$*).*

In this result, we derive the way to extend the definition of the set $\mathbb{E}\mathcal{S}$ (Definition 4.11). We take:

$$C_{\mathbf{rec}} ::= \forall \iota : N_{\mathbf{rec}} \to L \cup \{\_\}.$$
$$\begin{cases} \quad \exists l \in L.\ \left( \bigwedge_{r \in P} \tau(r(l)) \backslash \bigvee_{r \in N_{\mathbf{rec}} \mid \iota(r) = l} \tau(r(l)) \right) \in \mathcal{S} \\ \vee \quad \exists r' \in N_{\mathbf{rec}}.\ (\iota(r') = \_) \wedge \left( \bigwedge_{r \in P} \tau(\mathtt{def}(r)) \backslash \tau(\mathtt{def}(r')) \right) \in \mathcal{S} \\ \vee \quad \left( \bigwedge_{r \in P} \tau(\mathtt{def}(r)) \right) \in \mathcal{S} \end{cases}$$

where $N_{\mathbf{rec}} = N \cap T_{\mathbf{rec}}$ et $L = \bigcup_{r \in P \cup N_{\mathbf{rec}}} \mathrm{Dom}(r)$ (or a superset).

The construction of a universal model (Section 4.5) is not problematic. An algorithm to compute subtyping of the universal models is obtained by extending the algorithm of Chapter 7. We can also apply the technique in Section 7.3 for records. Rather than consider binary trees, we use a tree where branching is indexed by $L \cup \{\_\}$.

### 9.1.4 Decomposition, projection

We denote $\mathbb{1}_{\mathbf{rec}}$ the type of all records, which is $\{\_ = \mathbb{1}\}$.

We define the set of labels of a type $t$ by:

$$\mathcal{L}(t) = \bigcup_{(P,N)\in t} \bigcup_{r\in(P\cup N)\cap T_{\mathbf{rec}}} \mathrm{Dom}(r)$$

Let $L = \{l_1; \ldots; l_n\}$ be a finite set of labels. If $(t_l)_{l\in L}$ is a family of types, $E$ is a finite set of types, and $t$ a type, we define:

$$R((t_l)_{l\in L}; t_0; E) := \{l_1 = t_{l_1}; \ldots; l_n = t_{l_n}; \_ = t_0\} \setminus \bigvee_{s\in E} \{l_1 = \mathbb{1}; \ldots; l_n = \mathbb{1}; \_ = \neg s\}$$

Each type $s \in E$ adds the constraint which can be interpreted by "there exists a label outside of $L$ whose value is in $s$".

We can now state the equivalent of Lemmas 4.34 with 4.35 this representation.

**Lemma 9.3**

$$R((t_l)_{l\in L}; t_0; E) \wedge R((t'_l)_{l\in L}; t'_0; E') \simeq R((t_l \wedge t'_l)_{l\in L}; t_0 \wedge t'_0; E \cup E')$$

**Lemma 9.4**

$$R((t_l)_{l\in L}; t_0; E) \setminus R((t'_l)_{l\in L}; t'_0; \emptyset) \simeq R((t_l)_{l\in L}; t; E \cup \{\neg t'_0\}) \vee \bigvee_{l_0\in L} R((t''_{l,l_0})_{l\in L}; t_0; E)$$

*where* $t''_{l,l_0} = t_l \setminus t'_l$ *if* $l = l_0$ *and* $t''_{l,l_0} = t_l$ *otherwise.*

By using those two facts a proof similar to the one for Theorem 4.36, we can establish:

**Theorem 9.5** *Let $t$ be a type such that $t \leq \mathbb{1}_{\mathbf{rec}}$ and $\mathcal{L}(t) \subseteq L$. Then there exists a finite set $\pi^L_{\mathbf{rec}}(t)$ of triplets $((t_l)_{l\in L}, t_0, E) \in \widehat{T}^L \times \widehat{T} \times \mathcal{P}_f(\widehat{T})$ such that:*

- $t \simeq \bigvee_{((t_l),t_0,E)\in\pi^L_{\mathbf{rec}}(t)} R((t_l); t_0; E)$
- $\forall((t_l), t_0, E) \in \pi^L_{\mathbf{rec}}(t).\ R((t_l); t_0; E) \not\simeq \mathbb{0}$
- *if $\mathbb{\sqsupset}$ is a socle containing $t$, then: $\pi^L_{\mathbf{rec}}(t) \subseteq \mathbb{\sqsupset}^L \times \mathbb{\sqsupset} \times \mathcal{P}_f(\mathbb{\sqsupset})$*

*We see $\pi^L_{\mathbf{rec}}$ as a partial function, which is defined on the types $t$ such that $t \leq \mathbb{1}_{\mathbf{rec}}$ and $\mathcal{L}(t) \subseteq L$.*

Note that the condition $R((t_l); t_0; E) \not\simeq \mathbb{0}$ is equivalent to:

$$\begin{cases} \forall l \in L.\ t_l \not\simeq \mathbb{0} \\ t_0 \not\simeq \mathbb{0} \\ \forall s \in E.\ t_0 \wedge s \not\simeq \mathbb{0} \end{cases}$$

**Projection on the label** For a label $l_0$ and a subtype $t$ of $\mathbb{1}_{\mathbf{rec}}$, we want to define a type $\pi_{l_0}[t]$ such that, in a structural model:

$$[\![\pi_{l_0}[t]]\!] = \{\rho(l_0) \mid \rho \in [\![t]\!]\}$$

We can see it as the smallest solution $s$ of the inequality:

$$t \leq \{l_0 = s; \_ = \mathbb{1}\}$$

The Theorem 9.5 gives us the answer, by converting to types of the form $R((t_l)_{l\in L}; t_0; E)$ where we took $L \supseteq \mathcal{L}(t) \cup \{l_0\}$. The set of possible values of labels $\rho(l_0)$ when $\rho$ inhabits $[\![R((t_l)_{l\in L}; t_0; E)]\!]$ is $[\![t_{l_0}]\!]$. We can then define:

$$\pi_{l_0}[t] = \bigvee_{((t_l)_{l\in L}, t_0, E) \in \pi^L_{\mathbf{rec}}(t)} t_{l_0}$$

Actually, we could be satisfies with the simpler set $L \supseteq \mathcal{L}(t)$, by considering, if $l_0 \notin L$:

$$\pi_{l_0}[t] = \bigvee_{((t_l)_{l\in L}, t_0, E) \in \pi^L_{\mathbf{rec}}(t)} t_0$$

Indeed, the assertion "there exists at least a label not in $L$ whose value is in $s$", for $s \in E$, does not give any information (more precise than $t_0$) on the set of possible values for the label $l_0$.

## 9.2 Patterns

We add the records to the pattern algebra (Chapter 6). We add one case in the definition of the image $GY$ of the set $Y$ by the functor $G$:

$$\begin{array}{rcll} p \in GY & ::= & \ldots & \\ & | & r & r \in \mathcal{L} \xrightarrow{c} Y \end{array}$$

In the definition of a stable set of patterns (Definition 6.1), we add the condition $(\sigma \circ r) \in \mathcal{L} \xrightarrow{c} S$, which means:

$$\forall r \in S.\ \forall l \in \mathcal{L}.\ \sigma(r(l)) \in S$$

A more effective version of that condition is, for a pattern $p = \{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}$:

$$\forall i = 0..n.\ \sigma(q_i) \in S$$

The condition of well-formedness associated to a record pattern $r$ is:

$$\forall l_1 \neq l_2.\ \mathrm{Var}(r(l_1)) \cap \mathrm{Var}(r(l_2)) = \emptyset$$

A more effective version of that condition, for a pattern $p = \{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}$:

$$(\forall i \neq j.\ \mathrm{Var}(q_i) \cap \mathrm{Var}(q_j) = \emptyset) \wedge (\mathrm{Var}(q_0) = \emptyset)$$

So, the pattern node $q$ repeated infinitely often in $r$ can only perform a type test.

**Semantics** The semantics of pattern-matching (in a structural model supported by $D$) is given by:

$$d/r = \begin{cases} \bigoplus_{l \in \mathcal{L}} d(l)/\sigma(r(l)) & \text{si } d \in \mathcal{L} \xrightarrow{c} D \\ \Omega & \text{sinon} \end{cases}$$

A more effective version is, for a pattern $p = \{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}$ and an element $d = \{l_1 = d_1; \ldots; l_n = d_n; \_ = d_0\}$:

$$d/p = d_0/\sigma(q_0) \oplus \ldots \oplus d_n/\sigma(q_n)$$

Note that $d_0/\sigma(q_0)$ can only be $\Omega$ or $\{\}$, because $\mathrm{Var}(\sigma(q_0)) = \emptyset$ (well-formed pattern).

**Typing** We easily extend the operator $\wr\_\int$. In particular, for a record pattern $r$, we have:

$$\wr r \int = \wr\_\int \circ \sigma \circ r$$

where the right member is a record atom. More explicitly:

$$\wr\{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}\int = \{l_1 = \wr\sigma(q_1)\int; \ldots; l_n = \wr(\sigma(q_n)\int; \_ = \wr\sigma(q_0)\int\}$$

For the typing of pattern matching, we can extend the Lemma 6.11 with:

$$(t /\!\!/ r)(x) = (\pi_l[t] /\!\!/ \sigma(r(l)))(x)$$

where the label $l$ is defined by: $x \in \mathrm{Var}(\sigma(r(l)))$.

The theorem 6.12 can easily be extended: you only need to introduce an extension of the notion of a system $\mathsf{V}\times$ which considers records. Such a system is a function $\phi : V \to \mathcal{P}_f(V + V \times V + (\mathcal{L} \xrightarrow{c} V) + \widehat{T})$.

The elimination of the intersection (Section 6.5.2) in the patterns is done by observing that, similar to the product constructor $\times$, the record constructor commutes with the intersection (the intersection distributes over components).

**Compilation** The implementation technique of pattern matching based on strategies for the case of pairs (Chapter 8) can be adapted to the case of record values (meaning elements of $\mathcal{L} \xrightarrow{c} \mathcal{V}$). One strategy for the case of records can make two types of sub-queries. As in the case of products , it can apply a query $R$ on a sub-value $v(l)$; we denote $(l, R) \mapsto (\mathtt{s_R})_{\mathtt{R}:R}$ such a strategy. It can also apply a query $R$ on all values $v(l)$ for $l$ in a cofinite set $\mathcal{L}\backslash L$ ($L$ finite). Actually there is only a finite number of different values $v(l)$, which makes this computation possible. We denote $(\mathcal{L}\backslash L, R) \mapsto (\mathtt{s_R})_{\mathtt{R}:R}$ this strategy. In this case, all the atomic queries $(q, t_0, X)$ in $R$ are such that $\mathrm{Var}(q) = \emptyset$ (and then $X = \emptyset$), and the result, for each atomic query is logical conjunction of all the results for the sub-values $v(l)$ ($l \in \mathcal{L}\backslash L$), by identifying $\Omega$ with false and $\{\}$ with true.

## 9.3 Calculus

**Calculus** The calculus in Chapter 5 can be naturally extended with a record constructor in the expressions:

$$\begin{array}{rcll} e \in \mathcal{E} & ::= & \ldots & \\ & | & \rho & \rho \in \mathcal{L} \xrightarrow{c} \mathcal{E} \end{array}$$

**Type system** The associated typing rule is:

$$\frac{r \in \mathcal{L} \xrightarrow{c} \widehat{T} \quad (\forall l)\ \Gamma \vdash \rho(l) : r(l)}{\Gamma \vdash \rho : r}$$

A record expression whose components are values is itself a value. All the type system meta-theoretic properties are still valid.

The inversion lemma (Lemme 5.12) is extended with:

$$[\![r]\!]_{\mathcal{V}} = \{\rho \in \mathcal{L} \xrightarrow{c} \mathcal{V} \mid \forall l.\ \vdash \rho(l) : r(l)\}$$

**Semantics** To define the semantics, we add to the definition of evaluation contexts some contexts to reduce a given label $\{l_1 = []; l_2 = e_2; \ldots; l_n = e_n; \_ = e\}$ or a cofinite set of labels with the same expression $\{l_1 = e_1; \ldots; l_n = e_n; \_ = []\}$.

**Schemas, type inference** To setup the type inference algorithm, we add the records to the syntax of schemas. A record schema takes the form $R((\mathbb{t}_l)_{l\in L}; \mathbb{t}_0; E)$ where $L$ is a finite set of labels and $E$ is a finite set of types. Its semantics is given by:

$$\mathbf{\{}R((\mathbb{t}_l)_{l\in L}; \mathbb{t}_0; E)\mathbf{\}} = \{s \mid \exists (t_l) \in \prod_{l\in L} \mathbf{\{}\mathbb{t}_l\mathbf{\}}.\exists t_0 \in \mathbf{\{}\mathbb{t}_0\mathbf{\}}.R((t_L)_{l\in L}; t_0; E) \leq s\}$$

We can extend the proof of the Lemma 5.26 by getting some inspiration from the Lemmas 9.3 and 9.4. The property on the number of nested constructors $\otimes$ can be extended to the number of nested constructors for records. To to prove the lemma 5.27, we use:

$$\mathbb{0} \in \mathbf{\{}R((\mathbb{t}_l)_{l\in L}; \mathbb{t}_0; E)\mathbf{\}} \iff (\exists l.\mathbb{0} \in \mathbf{\{}\mathbb{t}_l\mathbf{\}}) \vee (\mathbb{0} \in \mathbf{\{}\mathbb{t}_0\mathbf{\}}) \vee (\exists s \in E.\mathbb{0} \in \mathbf{\{}s \oslash \mathbb{t}_0\mathbf{\}})$$

and an induction on the number of nested constructors for records.

The rule for the records in the typing algorithm is given by:

$$\mathbb{\Gamma}[\rho] := R((\mathbb{\Gamma}[\rho(l)])_{l\in \mathrm{Dom}(\rho)}; \mathbb{\Gamma}[\mathtt{def}(\rho)]; \emptyset)$$

## 9.4 Concatenation operator

We define a concatenation (or fusion) operator for records. It is a binary operator $\rho \oplus_t \rho'$ where the type $t$ allows the selection of which of the two argument provides a value for each label. We allow the expression $\rho \oplus_t \rho'$ to be written instead of $\oplus_t((\rho, \rho'))$.

The semantics is given by:

$$(\rho, \rho') \overset{\oplus_t}{\rightsquigarrow} \rho''$$

where $\rho''(l) = \rho(l)$ si $\rho(l) \notin [\![t]\!]_{\mathcal{V}}$ and $\rho''(l) = \rho'(l)$ si $\rho(l) \in [\![t]\!]_{\mathcal{V}}$. In other words, the $\rho$ fields which are of type $t$ are replaced by the corresponding fields in $\rho'$. The error cases are given by:

$$v \overset{\oplus_t}{\rightsquigarrow} \Omega$$

when $v$ is not a pair of records.

**Theorem 9.6** *There does not exist, in general, an exact type for the operator* $\oplus_t$.

> *Proof*: Let $c_0, c_1, c_2, c_3, c_4$ be five different constants. Define:
>
> $$t_0 = R(\emptyset; b_{c_0} \vee b_{c_1}; \emptyset) \times R(\emptyset; b_{c_2} \vee b_{c_3} \vee b_{c_4}; \{b_{c_3}, b_{c_4}\})$$
>
> The values of $t_0$ are then pairs of records $(\rho, \rho')$ where $\rho$ only contains $c_0$ or $c_1$ as values and $\rho'$ only contains $c_2$,$c_3$ or $c_4$, with at least a $c_3$ and at least a $c_4$. Let $t = b_{c_1}$. If $\oplus_t$ had an exact type, we could find a type $s_0$ representing exactly all the records that we can find by combining with $\oplus_t$ a pair $(\rho, \rho')$ as above. Let's take $l_0$ and $l_1$ two labels out of $\mathcal{L}(s_0)$. The record $\rho_1'' = \{l_0 = c_0; l_1 = c_0; \_ = c_2\}$ is in $s_0$ because we can get it with $\rho = \{l_0 = c_0; l_1 = c_0; \_ = c_1\}$ and $\rho' = \{l_0 = c_3; l_1 = c_4; \_ = c_2\}$. On the other hand, the record $\rho_2'' = \{l_0 = c_0; \_ = c_2\}$ is not in $s_0$. Indeed, if we could get it from $\rho$ and $\rho'$, we would have $\rho'(l) = c_2$ unless maybe for $l = l_0$, which prevents to find there $c_3$ and $c_4$ as expected. The fact that $\rho_1''$ is in $s_0$ and $\rho_2''$ is not in $s_0$ yields a contradiction with the lemma above. □

**Lemma 9.7** *Let $t$ be a type, and $L \supseteq \mathcal{L}(t)$. Let $\rho, \rho'$ be two record values which coincide on $L$ and which have the same direct image for $\mathcal{L}\backslash L$ ($\rho(\mathcal{L}\backslash L) = \rho'(\mathcal{L}\backslash L)$). Then: $\rho \in [\![t]\!] \iff \rho' \in [\![t]\!]$.*

> *Proof*: It is trivial when $t$ is an atomic record type and the property is stable by boolean combinations. □

We have to give up an exact type, and introduce an approximation in order to type this operator. We observe, in the proof of Theorem 9.6 that the impossibility to obtain an exact type comes from the existential constraints $E$ in the $R((t_l)_{l\in L}; t_0; E)$. Insofar as these constraints do not give any information when we extract a field value, it seems reasonable to forget about then when we apply the operator $\oplus_t$. A natural idea to define this approximation is to associate to a type $t \leq \mathbb{1}_{\mathbf{rec}}$ the type $\widetilde{t}$ defined by:

$$\widetilde{t} = \bigvee_{((t_l)_{l\in L}, t_0, E) \in \pi^L_{\mathbf{rec}}(t)} R((t_l)_{l\in L}; t_0; \emptyset)$$

where $L = \mathcal{L}(t)$, meaning that we remove the existential constraints $E$. Actually, each term in that union can be seen as a record atom. We denote $P(t)$ the set of these atoms:

$$\widetilde{t} = \bigvee_{r \in P(t)} r$$

However, such a definition is hard to use. In particular, it seems difficult to prove directly with it some simple properties of the operator $t \mapsto \widetilde{t}$, like its monotonicity (and even the invariance by type equivalence). In order to reason in a similar fashion as in the Theorem 4.42, we will adopt a semantic approach by describing the approximation as a saturating operator on the types semantics.

**Definition 9.8 (Adherence)** *Let $X$ be a set of record values. We define its* **adherence** *$\widetilde{X}$ as the set of record values which coincident with a specific element of $X$ on an arbitrary finite set of labels:*

$$\widetilde{X} = \{\rho \mid \forall L \in \mathcal{P}_f(\mathcal{L}).\exists \rho' \in X.\forall l \in L.\ \rho(l) = \rho'(l)\}$$

**Lemma 9.9** *If $X$ and $Y$ are two sets of record value, then:*

- $X \subseteq \widetilde{X}$
- $X \subseteq Y \Longrightarrow \widetilde{X} \subseteq \widetilde{Y}$
- $\widetilde{\widetilde{X}} = \widetilde{X}$
- $\widetilde{\emptyset} = \emptyset$
- $\widetilde{X \cup Y} = \widetilde{X} \cup \widetilde{Y}$

> *Proof*: The only point which is not immediate is the inclusion $\subseteq$ in the last inequality. Let $\rho$ a value in $\widetilde{X \cup Y}$. Suppose that $\rho$ is not in $\widetilde{X}$. We then have $L_0 \in \mathcal{P}_f(\mathcal{L})$ for which it is impossible to find $\rho' \in X$ coinciding with $\rho$ on $L_0$. We can show then that $\rho$ in is $\widetilde{Y}$. Let $L \in \mathcal{P}_f(\mathcal{L})$. Since $\rho$ is in $\widetilde{X \cup Y}$, we can find $\rho' \in X \cup Y$ which coincides with $\rho$ on $L \cup L_0$, and then *a fortiori* on $L_0$, which guarantees that $\rho'$ is not in $X$, which means that it must be in $Y$. □

**Lemma 9.10** *Let $t = R((t_l)_{l\in L}; t_0; E)$ and $t' = R((t_l)_{l\in L}; t_0; \emptyset)$. If $t \not\simeq \mathbb{0}$, then:*

$$\widetilde{[\![t]\!]_{\mathcal{V}}} = [\![t']\!]_{\mathcal{V}}$$

> *Proof*: To prove the inclusion $\subseteq$, given $t \leq t'$, it is enough to observe that $[\![t']\!]_{\mathcal{V}}$ is closed by adherence. Indeed, if $\rho$ is in the adherence of that set, then, for an arbitrary label $l$, by taking $L' = \{l\}$ in the definition of the adherence, we can see that $\rho(l) \in [\![t_l]\!]$ (for $l \in L$) or $\rho(l) \in [\![t_0]\!]$ (for $l \notin L$), which proves that $\rho$ is in $t'$.
>
> Let's prove now that any record value $\rho$ of type $t'$ is in the adherence of type $t$. Let $L'$ be a finite set of labels. We can then suppose that $L \subseteq L'$. For each type $s \in E$, we select a label $l_s \notin L'$ and a value $v_s$ of type $s\wedge t_0$ (it is possible since $t \not\simeq \mathbb{0}$). We suppose that the labels $l_s$ are pairwise distinct. We define then $\rho'$ as the record value which coincides with $\rho$ except on the $l_s$, and $\rho'(l_s) = v_s$. This record is in $[\![t]\!]$. □

With the lemmas 9.9 and 9.10, we can see that the syntactic definition of $\widetilde{t}$ coincides, at a semantical level, with the adherence operator.

**Corollary 9.11** *Let $t \leq \mathbb{1}_{\mathbf{rec}}$. Then $[\![\widetilde{t}]\!] = \widetilde{[\![t]\!]}$.*

This corollary provides, with the Lemma 9.9, some properties of the operator $t \mapsto \widetilde{t}$ such as its monotonicity with respect to the sub-typing relation.

We can see the semantics of the $\oplus_t$ operator as a function $\mathcal{V} \to \mathcal{V}_\Omega$. For a set $X$ of values, let $\oplus_t(X)$ be its direct image by this, meaning $\{v' \mid v' \in \mathcal{V}_\Omega, \exists v \in X.\ v \overset{\oplus_t}{\rightsquigarrow} v'\}$.

Let's study this function when $X$ is a product of atomic record types. For two types $t_1, t_2$, we write $f_t(t_1, t_2) = t_1$ si $t_1\wedge t \simeq \mathbb{0}$, and $f_t(t_1, t_2) = (t_1 \setminus t)\vee t_2$ otherwise. We extend this function point-to-point to record atoms $f_t(r_1, r_2) = (l \mapsto f_t(r_1(l), r_2(l)))$.

**Lemma 9.12** *Let $t_1, t_2$ be two non-empty types and $v$ a value of type $f_t(t_1, t_2)$. Then there exists two values $v_1 \in [\![t_1]\!], v_2 \in [\![t_2]\!]$ such that $v_1 \in [\![t]\!] \Rightarrow v = v_1$ and $v_1 \notin [\![t]\!] \Rightarrow v = v_2$.*

**Lemma 9.13** *If $r_1 \not\simeq \mathbb{0}, r_2 \not\simeq \mathbb{0}$, then:*

$$\oplus_t([\![r_1]\!] \times [\![r_2]\!]) = [\![f_t(r_1, r_2)]\!]$$

> *Proof*: Let's show first the inclusion $\subseteq$. Let $v$ be a value in $[\![r_1]\!] \times [\![r_2]\!]$. It is a pair $(\rho_1, \rho_2)$, with, for all $l$, $\rho_i(l)$ of type $r_i(l)$. When we apply the operator $\oplus_t$, we get a record $\rho$, and then we need to verify that $\rho(l)$ est de type $r(l) = f_t(r_1(l), r_2(l))$. Si $\rho_1(l)$ is of type $t$, then $r_1(l) \wedge t \not\simeq \mathbb{0}$, so $r_2(l) \leq r(l)$, and furthermore we have $\rho(l) = \rho_2(l)$, which is indeed of type $r(l)$. If $\rho_1(l)$ is not of type $t$, we have $\rho(l) = \rho_1(l)$, which is of type $r_1(l)$, but not of type $t$, but $r_1(l) \setminus t$ is necessarily a subtype of $r(l)$.
>
> Let's turn to the proof of the inclusion $\supseteq$. Let $\rho$ be a record of type $f_t(r_1, r_2)$. The lemma above allows the definition of two records $\rho_1$ and $\rho_2$, respectively of type $r_1$ et $r_2$, such that $(\rho_1, \rho_2) \overset{\oplus_t}{\rightsquigarrow} \rho$. □

Let's study now the set $\oplus_t([\![t_1]\!] \times [\![t_2]\!])$ for two sub-types $t_1$,$t_2$ of $\mathbb{1}_{\mathbf{rec}}$. The idea is to get an exact result modulo adherence. We will see that we can make this approximation before or after the application of the operator $\oplus_t$ withouth change the result.

**Lemma 9.14** *Let $X_1, X_2$ be two sets of records. Then:*

$$\oplus_t(\widetilde{X_1} \times \widetilde{X_2}) \subseteq \oplus_t(\widetilde{X_1 \times X_2})$$

> *Proof*: Let $\rho = \oplus_t(\rho_1, \rho_2)$ be an element of the left member, with $\rho_i \in \widetilde{X_i}$. Let's show that it is in the adherence of $\oplus_t(X_1 \times X_2)$. Let $L$ be a finite set of labels. We can find $\rho'_i \in X_i$ coinciding with $\rho_i$ on $L$. We then state $\rho' = \oplus_t(\rho'_1, \rho'_2)$, which coincides with $\rho$ sur $L$. □

**Lemma 9.15** *Let $t_1, t_2$ be two sub-types of $\mathbb{1}_{\mathbf{rec}}$ and $X = \oplus_t([\![t_1]\!] \times [\![t_2]\!])$. Then $\widetilde{X} = \oplus_t(\widetilde{[\![t_1]\!]} \times \widetilde{[\![t_2]\!]})$.*

> *Proof*: Let $X_i = [\![t_i]\!]$. The inclusion $\supseteq$ is a consequence of the above lemma. Given that $X_1 \times X_2 \subseteq \widetilde{X_1} \times \widetilde{X_2}$, to prove the inclusion $\subseteq$, it is enough to observe that the right member is closed by adherence. This property is stable by union on $t_1$ on on $t_2$, we can get back to the case where $t_1 = R((t_l^1)_{l \in L}; t_0^1; E^1)$ et $t_2 = R((t_l^2)_{l \in L}; t_0^2; E^2)$ for a specific $L$ (with $t_i \not\simeq \mathbb{0}$) and then the adherence of $t_i$ is represented by a record atom $r_i = R((t_l^i)_{l \in L}; t_0^i; \emptyset)$ (Lemma 9.10), and we have seen that $\oplus_t([\![r_1]\!] \times [\![r_2]\!])$ is itself represented by a record atom (Lemme 9.13), and it is then stable by adherence. □

We can now propose a type (non exact) for the operator $\oplus_t$. The relation $\oplus_t : \_ \rightarrow \_$ is defined by the inductive system of Figure 9.1.

$$\frac{}{\oplus_t : (r_1 \times r_2) \to f_t(r_1, r_2)}\ (\oplus_t\{\})$$

$$\frac{\oplus_t : t_1 \to s_1 \quad \oplus_t : t_2 \to s_2}{\oplus_t : (t_1 \wedge t_2) \to (s_1 \wedge s_2)}\ (\oplus_t \wedge)$$

$$\frac{\oplus_t : t_1 \to s_1 \quad \oplus_t : t_2 \to s_2}{\oplus_t : (t_1 \vee t_2) \to (s_1 \vee s_2)}\ (\oplus_t \vee)$$

$$\frac{\oplus_t : t' \to s' \quad t \leq t' \quad s' \leq s}{\oplus_t : t \to s}\ (\oplus_t \leq)$$

Figure 9.1: Axiomatisation of the type of the operator $\oplus_t$

**Lemma 9.16** *The operator $\oplus_t$ is well-typed.*

*Proof*: Like the proof for Lemma 5.35, it is enough to consider the rule $(\oplus_t\{\})$. This case is given by the inclusion $\subseteq$ of Lemma 9.13. □

**Lemma 9.17** *Let $t_0$ et $s_0$ be two types. The following assertions are equivalent:*

(*i*) $\oplus_t : t_0 \to s_0$

(*ii*) $\widetilde{\oplus_t(\llbracket t_0 \rrbracket)} \subseteq \llbracket s_0 \rrbracket$

(*iii*) $\begin{cases} t_0 \leq \mathbb{1}_{\mathbf{rec}} \times \mathbb{1}_{\mathbf{rec}} \\ \forall (t_1, t_2) \in \pi(t_0).\ \forall r_1 \in P(t_1), r_2 \in P(t_2).\ f_t(r_1, r_2) \leq s_0 \end{cases}$

*Proof*: Let's prove first $(i) \Rightarrow (ii)$ by induction on the derivation of $\oplus_t : t_0 \to s_0$. For the rule $(\oplus_t\{\})$, we get in reality an equality with the Lemma 9.13. For the rule $(\oplus_t \vee)$, we use the fact that the left member of $(ii)$ commutes with the union. For the two other rules, we use the monotonicity of the left member.

Let's prove $(ii) \Rightarrow (iii)$. If we didn't have $t_0 \leq \mathbb{1}_{\mathbf{rec}} \times \mathbb{1}_{\mathbf{rec}}$, then $\oplus_t(\llbracket t_0 \rrbracket)$ would contain $\Omega$, which is not the case. Let $(t_1, t_2) \in \pi(t_0)$. We have $t_1 \times t_2 \leq t_0$, and then $\widetilde{\oplus_t(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)} \subseteq \llbracket s_0 \rrbracket$. According to the Lemma 9.15 and the Corollary 9.11, the left member is $\oplus_t(\llbracket \widetilde{t_1} \times \widetilde{t_2} \rrbracket)$. If $r_i \in P(t_i)$, then $r_i \leq \widetilde{t_i}$, and then $\oplus_t(\llbracket r_1 \rrbracket \times \llbracket r_2 \rrbracket) \leq \llbracket s_0 \rrbracket$ and the left member is $\llbracket f_t(r_1, r_2) \rrbracket$, from which we deduce indeed that $f_t(r_1, r_2) \leq s_0$.

Finally let's prove $(iii) \Rightarrow (i)$. We write $t = \bigvee_{(t_1,t_2)\in\pi(t_0)} t_1 \times t_2$. By writing

$$t_i \leq \widetilde{t_i} = \bigvee_{r_i \in P(t_i)} r_i$$

we can see that:

$$t \leq \bigvee_{\substack{(t_1,t_2)\in\pi(t) \\ r_1 \in P(t_1) \\ r_2 \in P(t_2)}} r_1 \times r_2$$

With the rule $(\oplus_t\{\})$ and the rule $(\oplus_t \leq)$, we get $\oplus_t : (r_1 \times_2) \to s_0$,

and we conclude with the rule ($\oplus_t\vee$). The case $t_0 \leq \mathbb{0}$ is dealt with separately, with the rule ($\oplus_t\{\}$) applied to empty record atoms. □

As in $\pi_1$, the assertion (*iii*) allows us to construct a typing function for $\oplus_t$ on schemas. We extend the function $\pi$ to schemas $\mathbb{t}$ such that $\mathbb{t} \leq \mathbb{1}_{\mathbf{prod}}$ by

$$\begin{array}{lcl} \pi(\mathbb{t}_1 \between \mathbb{t}_2) & = & \pi(\mathbb{t}_1) \cup \pi(\mathbb{t}_2) \\ \pi(\mathbb{t}_1 \otimes \mathbb{t}_2) & = & \{(\mathbb{t}_1, \mathbb{t}_2)\} \end{array}$$

The function $P$ associates to every type $t \leq \mathbb{1}_{\mathbf{rec}}$ a finite set of atomic record types, which means quasi-constant functions $\mathcal{L} \to \widehat{T}$. We extend it to define $P(\mathbb{t})$, a finite set of quasi-constant functions $\mathbb{r}$ of labels in the schemas (we identify such a function to a $R((\mathbb{t}_l)_{l\in L'}; \mathbb{t}_0; \emptyset)$):

$$\begin{array}{lcl} P(\mathbb{t}_1 \between \mathbb{t}_2) & = & P(\mathbb{t}_1) \cup P(\mathbb{t}_2) \\ P(R((\mathbb{t}_l)_{l\in L'}; \mathbb{t}_0; E) & = & \{R((\mathbb{t}_l)_{l\in L'}; \mathbb{t}_0; \emptyset)\} \end{array}$$

We also extend the function $f_t$ to pairs of record schemas $(\mathbb{r}_1, \mathbb{r}_2)$, in an obvious fashion. We can then give a definition for $\oplus_t[\mathbb{t}]$:

$$\oplus_t[\mathbb{t}] = \begin{cases} \bigvee\limits_{\substack{(\mathbb{t}_1,\mathbb{t}_2)\in\pi(\mathbb{t}) \\ \mathbb{r}_1\in P(\mathbb{t}_1) \\ \mathbb{r}_2\in P(\mathbb{t}_2)}} f_t(\mathbb{r}_1, \mathbb{r}_2) & \text{si } \mathbb{t}_0 \leq \mathbb{1}_{\mathbf{rec}} \times \mathbb{1}_{\mathbf{rec}} \\ \Omega & \text{sinon} \end{cases}$$

## 9.5 Partial functions

The formalism that we adopted to present records is that of quasi-constant functions. Compared to the formalism of the functions on a finite domain, it has the advantage of being uniform and to simplify the assertions. In this section, we show how to encode some functions on a finite domain of records.

We choose a constant $\mu \in \mathcal{C}$ which represents the value of record fields which are actually not defined. For the expressions, we simply write $\{l_1 = e_1; \ldots; l_n = e_n\}$ instead of $\{l_1 = e_1; \ldots; l_n = e_n; \_ = \mu\}$. For the type, we write $\{l_1 = t_1; \ldots; l_n = t_n\}$ for $\{l_1 = t_1; \ldots; l_n = t_n; \_ = \mathbb{1}\}$ (open record type and $\{|l_1 = t_1; \ldots; l_n = t_n|\}$ for $\{l_1 = t_1; \ldots; l_n = t_n; \_ = b_\mu\}$ (closed record type), and similarly for patterns. Instead of a field specification in a record type, we also allow to write $l_i : t_i$ fo $l_i = t_i \backslash b_\mu$, and $l_i :? t_i$ for $l_i = t_i \vee b_\mu$, and similarly for patterns: $l_i : p_i$ means that $l_i = p_i \& \neg b_\mu$. This way, the pattern $\{|l_1 : x;\ l_2 = ((y \& \neg b_\mu) \,|\, (y := c))|\}$ accepts any record defined on $l_1$, eventually on $l_2$, and nowhere else, and captures the value for $l_1$ in $x$ and the one for $l_2$ in $y$ if this field is defined, and otherwise associates the constant $c$ to $y$.

**Derived Operators** We can also defined some number of operators derived from $\oplus_t$. For a given constant $c$, we write $\oplus_c$ instead of $\oplus_{b_c}$. We select a constant $c_0$, different from $\mu$ (it is only used to provide some semantics to the new operators, and this semantics does not depend on the choice of $c_0$). Let $L = \{l_1, \ldots, l_n\}$ be a finite set of labels. The restriction operator $_{|L}$ is defined by:

$$e_{|L} := \{l_1 = c_0; \ldots; l_n = c_0; \_ = \mu\} \oplus_{c_0} e$$

The suppression operator $_{\setminus L}$ is defined by:

$$e_{\setminus L} := \{l_1 = \mu; \ldots; l_n = \mu; \_ = c_0\} \oplus_{c_0} e$$

For a type $t$, the restriction operator on $t$ is defined by:

$$e_{|t} := r \oplus_{\neg t} \{\_ = \mu\}$$

This operator removes the fields which are not of type $t$. Finally, we simply write $r_1 \oplus r_2$ fo $r_2 \oplus_\mu r_1$. This is a concatenation operator, which takes, in the case of a label conflict (if the two arguments are defined for that label), the value of the second argument.

**Finite domain** If we only restrict ourselves to writing record expression of the form $\{l_1 = e_1; \ldots; l_n = e_n\}$, meaning that they are records with finite domain (which evaluate to $\mu$ almost everywhere), then the subtyping relation is not complete with respect to the model of values (formally: the set of values is not a model anymore). Indeed, the type $\{\_ = t\}$ contains no value as soon as $t \wedge b_\mu \simeq \mathbb{0}$. It is very easy to modify the definition of a model to fix this: we can take, instead of $\mathcal{L} \xrightarrow{c} D$, the set of functions which evaluate to $\mu$ almost everywhere and this changes slightly the subtyping relation (in the definition of the set $\mathbb{E}\mathcal{S}$, we replace $(\bigwedge_{r \in P} \tau(\mathtt{def}(r))) \in \mathcal{S}$ with $(b_\mu \wedge \bigwedge_{r \in P} \tau(\mathtt{def}(r))) \in \mathcal{S}$). The type $\{l_1 = t_1; \ldots; l_n = t_n; \_ :? t\}$ means then that excepted for the labels $l_i$, the record can be defined on a specific finite number of other labels, and that the values of these fields are necessarily in $t$.

# Chapter 10

# Presentation of the language

In this chapter, we present ℂDuce, a programming language built on the theoretical foundations presented in the previous chapters. ℂDuce has the traits of a general-purpose functional programming language, but it has been designed specifically to develop applications that securely handle XML documents.

The goal of this thesis is not to be used as a reference manual or a user manual for ℂDuce, and since the language keeps evolving, we will not formally specify its concrete syntax or its semantics, but we will present its characteristics via an encoding in the calculus of Chapter 5, extended with pattern matching (Chapter 6) and records (Chapter 9).

In ℂDuce, the syntax for functions combines abstraction and pattern matching:

$$\texttt{fun}\ f(t_1 \texttt{→} s_1; \ldots; t_n \texttt{→} s_n)\ p_1 \to e_1 \mid \ldots \mid p_m \to e_m$$

The identifier $f$ can be omitted if the function is not recursive.

## 10.1 Base types, constants

First of all, we want to introduce the general frame by specifying the base types and the constants. There are three kinds of constants: integers, atoms, and characters.

**Integers** We use arbitrary precision integers. Therefore, the set of integer constants is the set of relative integers $\mathbb{Z}$ . We write `Int` the base type that denotes all those integers: $\mathbb{B}[\![\texttt{Int}]\!] = \mathbb{Z}$. We also introduce, as a base type, closed intervals $i\text{—}j$ where $i, j \in \mathbb{Z}$, with $\mathbb{B}[\![i\text{—}j]\!] = \{n \mid i \leq n \leq j\}$, as well as open intervals $i\text{—}\infty$ with $\mathbb{B}[\![i\text{—}\infty]\!] = \{n \mid i \leq n\}$ and $\infty\text{—}j$ with $\mathbb{B}[\![\infty\text{—}j]\!] = \{n \mid n \leq j\}$. The singleton type $i\text{—}i$ is simply written $i$.

Any sub-type $t$ of `Int` can be written as a finite union of closed interval types. This decomposition is not unique. However, there exists a canonical decomposition, that can be characterized in two different ways:

— the intervals in the decomposition are maximal among the sub-type intervals of $t$;
— the number of intervals in the decomposition is minimal.

In practice, starting from a decomposition, we can obtain this canonical decomposition by (iteratively) merging two intervals that have a non-empty intersection, or that "touch" each other (like $i$—$j$ and $(j+1)$—$k$).

The language has the classic arithmetic operators. The addition operator, for example, has an exact typing, whereas the multiplication operator does not. For example, the multiplication of a an integer of type `Int` with an integer of type 2 gives, semantically, the set of even integers, which is not representable by a type. Of course, we can systematically give the `Int` type to the result, or take a more specific type. For example, we could do interval arithmetic, by specifying that we use the decomposition in maximal intervals.

**Characters** The character constants represent elements of the Unicode character set. Each constant represents a *code point* of that specification, and can therefore be associated to an integer. We denote by `Char` the type of all characters. Its interpretation is the set of all Unicode *code points*. We also introduce interval base types that each represent a segment of the Unicode codespace. The singleton base type associated with an atom constant is an interval type. Characters are written with simple quotes ($'\texttt{a}', '\texttt{b}', \ldots$).

**Atoms** Atoms are used, among other things, to represent the labels (*tags*) of XML elements. An atom is a pair $`ns : ln$ where $ns$ is an XML *namespace* (that is, a possibly empty Unicode string), and $ln$ is a local name, in the meaning of the XML Namespaces [Nam] specification. When $ns$ is the empty *namespace*, we simply write $`ln$ for the atom $`ns : ln$. We denote by `Atom` the type of all atoms, $`ns : *$ denotes the type of atoms in the *namespace* $ns$, and $`ns : ln$ denotes the unique atom $`ns : ln$.

In ℂDuce programs, *namespaces* are not written directly in the atom constants. The language uses a mechanism similar to the XML Namespaces specification to define prefixes which compactly denote *namespaces*. These prefixes are then used to actually write the atom constants in ℂDuce programs.

## 10.2 Types, patterns

The type algebra (Chapter 3) and the pattern algebra (Chapter 6) have many similar constructors: union, intersection, product, record. A pattern $p$ with no capture variables ($\mathrm{Var}(p)$) can be naturally identified with the type $\lceil p \rfloor$, and every type $t$ can be seen as a pattern. It is natural to syntactically unify these two algebras, and the language does use a common external syntax to represent these two algebras. The union, intersection and difference operators are written respectively `|`,`&` and `\`. The product constructor is written `(_,_)` (instead of `_×_` for types). The types $\mathbb{0}$ and $\mathbb{1}$ are denoted by `Empty` and `Any`. We also use the notation `_` instead of `Any` (especially in patterns).

A term that does not contain capture variables can be used as a type. To use a term of this algebra as a pattern, it has to verify the well-formedness conditions on capture variables. A sub-term of the form $p_1 \to p_2$ cannot contain

any variables, which in fact makes it possible to see it as a type constrain pattern. Similarly, in $p_1 \backslash p_2$, the term $p_2$ should not contain any variable.

A term like (`Int`, `Int`), used as a pattern, can be interpreted in two different ways: either as a type constraint pattern (the type being a product type), or as a product pattern of which each component is a type constraint pattern. This kind of ambiguity between types and patterns is not a problem (the different interpretations have the same semantics, and therefore the same typing).

The ℂDuce language uses the subtyping relation $\leq$ induced by a universal model (Section 4.5). This subtyping relation can be computed using the algorithms of Chapter 7.

**Records** We use the formalism of Section 9.5 [1] . The only record expressions allowed are of the form $\{l_1 = e_1; \ldots; l_n = e_n\}$. For the constant $\mu$, that represents undefined record fields, we take the ‘`missing` atom. For the set of labels $\mathcal{L}$, we take the set of atom constants (written without the ‘).

We denote by $e.l$ the access to field $l$. It is syntactic sugar for:

$$\texttt{match}\ e\ \texttt{with}\ \{l = x\} \rightarrow x$$

**XML elements** In addition to functions, pairs, records and constants, the language manipulates another kind of values: XML elements. An expression that denotes such a value has the form $< (e_1)\ e_2 > e_3$. It behaves like a simple three-term Cartesian product. It could be identified with the nested pair $(e_1, (e_2, e_3))$, but we prefer to introduce a new category (of values, types and patterns) to avoid ambiguities. When $e_1$ is an atom constant ‘$ns : ln$, we can write it without the initial ‘: $< ns : ln\ e_2 > e_3$. When $e_2$ is a record expression $\{\ldots\}$, we can omit the { and }: $< (e_1)\ \ldots > e_3$, as well as the semicolons that separate the different fields. Of course, we have types and patterns that correspond to this new category of values and expressions: $< (p_1)\ p_2 > p_3$, with the same simplifications. Subtyping, typing and pattern matching for this new category of values can easily be deduced from the theory introduced for products.

**Recursion** Recursive types and patterns can be written in the form $t_0$ `where` $\texttt{X}_1 = t_1$ `and` ... `and` $\texttt{X}_n = t_n$ where the $\texttt{X}_i$ are recursion binders that can be used inside within $t_0$ and the $t_i$. For types, the language also allows for mutually recursive global statements:

`type` $\texttt{X}_1$ `=` $t_1$ ... `type` $\texttt{X}_n$ `=` $t_n$

Recursions must be well-formed (every recursion cycle must go through at least one arrow, product, XML or record constructor).

## 10.3 Sequences, regular expressions

A number of derived syntactic constructions and operations are introduced to facilitate the manipulation of value sequences.

[1] In fact the implementation is slightly different: in records, an empty field is not represented by a value of the language (records are partial functions).

### 10.3.1 Sequences

Sequences are encoded by nested pairs, and we use the atom constant `'nil` to represent the empty sequence. A sequence expression $[e_1 \dots e_n]$ is therefore translated into the expression $(e_1, \dots, (e_n, \text{`}\texttt{nil}) \dots)$.

### 10.3.2 Type and pattern regular expressions

To accurately describe the type of sequences, and to be able to perform extractions by pattern matching, we introduce regular expressions in the type and pattern algebras. These terms are denoted by $[R]$ where $R$ is a regular expression, that is, a term generated by:

$$\begin{array}{rcl} R & := & p \\ & | & R_1 R_2 \\ & | & R_1 | R_2 \\ & | & R* \\ & | & R? \\ & | & R*? \\ & | & R?? \\ & | & x :: R \end{array}$$

It is only syntactic sugar, and we will later show how to translate a term $[R]$ into a type or a pattern.

The $x :: R$ notation captures the sub-sequence recognized by $R$ in $x$. We say that $x$ is a sequence capture variable. When the same sequence capture variable is used several times in the same regular expression, or when it occurs under a repetition operator, the expected semantics is to concatenate all the corresponding sub-sequences (nested occurrences are not allowed).

The two constructors $*$ and $*?$ are Kleene stars. The difference between the two is that the first has a greedy *greedy* behavior - it tries to do as many iterations as possible - whereas the second has a lazy behavior (it tries to do as few iterations as possible). Similarly, the two constructors ? and ?? optionally capture their argument: the first tries to capture it first, while the second avoids doing so if possible.

**Remark 10.1** *Vansummeren [Van04] has noticed that the greedy semantics for the Kleene star does not simulate a longest-match semantics, as one might naively believe. Consider for example the pattern* $p = [x :: (1|(1\ 2)) *\ 2?]$ *and the value* $v = [1\ 2]$. *The result for* $x$ *with the greedy semantics is* $[1]$, *because during the first iteration in the star, the first branch of the disjunction* $(1|(1\ 2))$ *is tried, and succeeds, which prevents a second iteration without preventing the pattern match from succeeding overall. A longest-match semantics for the star would capture the whole sequence* $[1\ 2]$ *in* $x$. *Vansummeren studies such semantic variants, and the corresponding type inference problems.*

### 10.3.3 Translation of regular expressions

To describe the translation, we will generalize the syntax of $[R]$ and use terms of the form $[R; p_1; p_2]$. The form $[R]$ is translated into $[R; \text{`}\texttt{nil}; \text{`}\texttt{nil}]$. Intuitively, the patterns $p_1$ and $p_2$ represent "continuations", i.e. the pattern to be applied to

the rest of the sequence once $R$ has been recognized on a prefix of the sequence. The pattern $p_1$ is used when $R$ has consumed at least one item, and the pattern $p_2$ is used when $R$ has not recognized any element. We will explain below why we distinguish between these two continuations.

The first step in translation is to eliminate sequence capture variables. This is done by replacing $[R_0; p_1; p_2]$ with $[R'_0; p'_1; p'_2]$. The regular expression $R'_0$ is obtained in the following way. If a sub-expression $x :: R$ appears in $R_0$, it is replaced by $R'$ where $R'$ is obtained from $R$ by replacing all the patterns $p$ in it with $(x\&p)$. If $x_1, \ldots, x_n$ are all the sequence capture variables that appear in $R_0$, the continuations $p'_1$ and $p'_2$ are defined by $p'_i = (x_1 := \text{`}\texttt{nil})\& \ldots \&(x_n := \text{`}\texttt{nil})\&p_i$.

For example, the translation of $[x :: (p_1 \;\; y :: (p_2?)) \;\; y :: p_3; \text{`}\texttt{nil}; \text{`}\texttt{nil}]$ is $[(x\&p_1)(x\&y\&p_2)?(y\&p_3); p'; p']$ with $p' = (x := \text{`}\texttt{nil})\&(y := \text{`}\texttt{nil})\&\text{`}\texttt{nil}$ (we could take $x\&y\&\text{`}\texttt{nil}$ which is equivalent to $p'$).

Let us see now how to translate $[R_0; p_1; p_2]$ when $R_0$ does not contain any sequence capture variable. We define this translation with the function $\Psi$, by induction on the structure of $R_0$:

$$\begin{array}{lcl}
\Psi[p; p_1; p_2] & = & (p, p_1) \\
\Psi[R_1 R_2; p_1; p_2] & = & \Psi[R_1; \Psi[R_2; p_1; p_1]; \Psi[R_2; p_1; p_2]] \\
\Psi[R_1|R_2; p_1; p_2] & = & \Psi[R_1; p_1; p_2]|\Psi[R_2; p_1; p_2] \\
\Psi[R*; p_1; p_2] & = & \texttt{X}|p_2 \text{ } \texttt{where X} = \Psi[R; \texttt{X}|p_1; \texttt{Empty}] \\
\Psi[R*?; p_1; p_2] & = & p_2|\texttt{X} \text{ } \texttt{where X} = \Psi[R; p_1|\texttt{X}; \texttt{Empty}] \\
\Psi[R?; p_1; p_2] & = & \Psi[R; p_1; p_2]|p_2 \\
\Psi[R??; p_1; p_2] & = & p_2|\Psi[R; p_1; p_2]
\end{array}$$

In these definitions, the binding $X$ is fresh. In view of the *first-match* policy for pattern matching, we see that the star $*$ has indeed a greedy behavior (it captures as many elements has possible before moving on to the rest), and that $*?$ has a lazy behavior. Similarly, the regular expression $R?$ first tries to recognize $R$ before giving up if that is impossible, and the regular expression $R??$ tries to move on to the rest, before recognizing $R$ if that is necessary.

The construction is quite classic. The only subtlety comes from the fact that one must produce well-formed recursions, which pass through a constructor. That is why we distinguish between the two continuations $p_1$ and $p_2$. If we had only one continuation $[R; p]$, we would have taken:

$$\Psi[R*; p] = \texttt{X where X} = \Psi[R; X]|p$$

But that would have resulted in an ill-formed recursion when $R$ accepts the empty sequence. For example, for $R = [(p_1 * \;\; p_2*)*]$, we would have (after rearranging a bit):

$$\Psi[R; p] = \texttt{X}_0 \text{ } \texttt{where} \left\{ \begin{array}{lcl} \texttt{X}_0 & = & \texttt{X}_1|p \\ \texttt{X}_1 & = & (p_1, \texttt{X}_1)|\texttt{X}_2 \\ \texttt{X}_2 & = & (p_2, \texttt{X}_2)|\texttt{X}_0 \end{array} \right.$$

which gives a recursion cycle that does not pass through any constructor. Having two different continuations permet allows each iteration of $R$ in $[R*]$ to capture at least one element (in terms of automatons, this means eliminating the $\epsilon$-transition cycles). Our definition of $\Psi$ does not introduce any ill-formed

recursion (because the continuation $p_1$ is only used under a constructor). Using the example below, we obtain (after some cosmetic arrangements):

$$\Psi[R;p;p] = \mathtt{X}_0 \texttt{ where } \begin{cases} \mathtt{X}_0 & = & \mathtt{X}_1|\mathtt{X}_2|p \\ \mathtt{X}_1 & = & (p_1, \mathtt{X}_0) \\ \mathtt{X}_2 & = & (p_2, \mathtt{X}_2|\mathtt{X}_0) \end{cases}$$

This recursive definition is well-formed: the recursion cycles all pass through a constructor.

**Remark 10.2** *This presentation of the translation, with two continuations, is inspired by an in-depth study [FC04] of the problem posed by "dangerous" regular expressions, i.e. with a sub-term of the form $R*$, where $R$ accepts the empty sequence.*

**Other constructors** We also introduce constructors for regular expression derivatives $R+$ and $R+?$ defined by $R+ = R\ R*$ and $R+? = R\ R*?$. We can also introduce a predicate constructor $/p$, which applies a certain pattern to the current sequence without consuming any element. Its translation is given by:

$$\Psi[/p;p_1;p_2] = p\&p_2$$

This predicate can be used, for example, to position inside a variable an indicator that indicates the choice that was made in an alternative:

$$[(p_1 * /(x := 1))|(p_2 * /(x := 2))]$$

In this example, the variable $x$ will be bound to 1 (resp. 2) if it is the first (resp. second) branch that was chosen.

**Example** We are going to illustrate by an example how the semantics chosen for product patterns $(q_1, q_2)$ is able to capture sub-sequences (not necessarily consecutive) when the same variable appear in both the $q_i$.

Let us take $p = [((x :: t)|\_)*]$. Applied to a a sequence, this pattern captures in $x$ the sub-sequence formed of all elements of type $t$. The translation of $p$ is

$$\mathtt{X} \texttt{ where } \mathtt{X} = (x\&t, \mathtt{X})|(\_, \mathtt{X})|(x := \texttt{`nil})\&\texttt{`nil}$$

If we apply this pattern to a value $[v_1\ v_2\ v_3] = (v_1, (v_2, (v_3, \texttt{`nil})))$ with, for example $v_1, v_3$ of type $t$ and $v_2$ of type $\neg t$, we get for $x$ the result $[v_1\ v_3] = (v_1, (v_3, \texttt{`nil}))$.

Note that the regular expression patterns automatically benefit from the exactness of pattern matching, including for sequence capture variables. They also benefit from the efficient compilation of pattern matching. For example, if the above pattern is applied to an expression of type $[(\neg t)\ t*]$, and if the factorization of the capture variables (Section 8.4) is used during compilation, this pattern can be compiled as $(\_, x)$ (i.e. the second projection). If we apply it to an expression of type $[t_1\ (\neg t)\ t_2+]$, with $t_1 \leq t$, $t_2 \leq t$, then it can be compiled as $(x, (\_, x))$, and the type system will know that $x$ has the type $[t_1\ t_2+]$.

### 10.3.4 Decompilation of sequence types

**Definition 10.3** *An* **automaton** *is a quadruplet $\mathcal{A} = (Q, \delta, \mathbb{q}_0, F)$ where $Q$ is a finite set of states, $\delta \subseteq\in \mathcal{P}_f(Q \times \widehat{T} \times Q)$ is the transition relation, $\mathbb{q}_0$ is the initial state and $F \subseteq Q$ is the set of final states.*

*The semantics $[\![\mathcal{A}]\!]_{\mathbb{q}}$ of a state $\mathbb{q} \in Q$ is a set of sequence values, defined by:*

$$\begin{array}{lcl} \text{`}\mathtt{nil} \in [\![\mathcal{A}]\!]_{\mathbb{q}} & \iff & \mathbb{q} \in F \\ (v_1, v_2) \in [\![\mathcal{A}]\!]_{\mathbb{q}} & \iff & \exists (t, \mathbb{q}').\ (\mathbb{q}, t, \mathbb{q}') \in \delta \wedge v_1 \in [\![t]\!] \wedge v_2 \in [\![\mathcal{A}]\!]_{\mathbb{q}'} \end{array}$$

*(This definition is well-founded on the structure of values.) We simply write $[\![\mathcal{A}]\!]$ for $[\![\mathcal{A}]\!]_{\mathbb{q}_0}$.*

We denote by $\mathbb{1}_{\mathbf{seq}}$ the type $[\mathtt{Any}*]$.

**Theorem 10.4** *Let $X$ be a set of values. The following assertions are equivalent:*

*(i) There exists a type $t \leq \mathbb{1}_{\mathbf{seq}}$ such that $[\![t]\!] = X$.*
*(ii) There exists an automaton $\mathcal{A}$ such that $[\![\mathcal{A}]\!] = X$.*
*(iii) There exists a regular expression $R$, generated by the production below, such that $[\![[R]]\!] = X$.*

$$\begin{array}{rcl} R & := & t \\ & | & R_1 R_2 \\ & | & R_1 | R_2 \\ & | & R* \end{array}$$

*Proof*: The implication $(iii) \Rightarrow (i)$ is trivial. The implication $(ii) \Rightarrow (iii)$ is a classic result on word automatons (to formally prove this implication, we would have to give a direct semantics to types of the form $[R]$, and prove that this translation preserves that semantics). Note that the empty language is obtained with $R = \mathtt{Empty}$, and the language reduced to the empty sequence is obtained with $R = \mathtt{Empty}*$.

Let us prove the implication $(i) \Rightarrow (ii)$. Let $\beth$ be a socle that contains $t$ (and $b_{\text{`}\mathtt{nil}}$). Considering the automaton $(Q, \delta, t, F)$, where:

$$\begin{array}{rcl} Q & = & \{t_0 \in \beth \mid t_0 \leq \mathbb{1}_{\mathbf{seq}}\} \\ \delta & = & \{(t_0, t_1, t_2) \in Q \times \beth \times Q \mid (t_1, t_2) \in \pi(t_0 \backslash b_{\text{`}\mathtt{nil}})\} \\ F & = & \{t_0 \in \beth \mid b_{\text{`}\mathtt{nil}} \leq t_0\} \end{array}$$

It is easy to show that $[\![\mathcal{A}]\!] = [\![t]\!]$. Note that if $t_0 \leq \mathbb{1}_{\mathbf{seq}}$ and if $(t_1, t_2) \in \pi(t_0 \backslash b_{\text{`}\mathtt{nil}})$, then $t_2 \leq \mathbb{1}_{\mathbf{seq}}$. □

### 10.3.5 Operators

**Concatenation** We introduce a sequence concatenation operator, denoted @. The semantics of this operator is given by:

$$([v_1\ \ldots\ v_n], [v_{n+1}\ \ldots\ v_m]) \overset{@}{\rightsquigarrow} [v_1\ \ldots\ v_m]$$

and

$$v \overset{@}{\rightsquigarrow} \Omega$$

when $v$ is not a pair of sequences.

Given Theorem 10.4, we obtain an exact typing by taking @ : $[R_1]\times[R_2]\rightarrow[R_1\ R_2]$ and some inference rules similar to the ones that define $\pi_1 : \_\rightarrow\_$ (Figure 5.4). To compute the result type given the input type, we can choose arbitrary representatives for $R_1$ and $R_2$. In practice, starting from $t_1$ and $t_2$, we can compute the result type for $t_1\times t_2$ without explicitly building the two regular expressions $R_1$,$R_2$ with $t_i \simeq [R_i]$. The idea is to construct the automaton associated with $t_1$, and to produce a system of equations that defines the result type result. This system of equations can easily be deduced from the automaton (we "put back" $t_2$ in the final states). This algorithm can be generalized to define the typing function for @ over schemas.

**Flattening** We also introduce a flattening operator for sequences `flatten`.

The semantics of this operator is given by:

$$[[v_1\ \ldots\ v_n]\ \ldots\ [v_m\ \ldots\ v_l]] \overset{\texttt{flatten}}{\rightsquigarrow} [v_1\ \ldots\ v_l]$$

and

$$v \overset{\texttt{flatten}}{\rightsquigarrow} \Omega$$

when $v$ is not a sequence of sequences.

An exact typing is obtained from rule `flatten` : $[R]\rightarrow[\overline{R}]$ where $\overline{R}$ is defined by replacing in $R$ (a regular expression in the meaning of Theorem 10.4) a type $t$ by $R'$ where $R'$ is chosen such that $t \simeq [R']$.

Note that the operator @ could be defined from `flatten` by writing @$((v_1, v_2)) =$ `flatten`$([v_1\ v_2])$.

**Map** The `map` operator applies a function to each element of a sequence. Its semantics is given by:

$$(f, [v_1\ \ldots\ v_m]) \overset{\texttt{map}}{\rightsquigarrow} [(f\ v_1)\ \ldots\ (f\ v_m)]$$

and

$$v \overset{\texttt{map}}{\rightsquigarrow} \Omega$$

when $v$ is not a pair formed of an abstraction and a sequence.

The typing is obtained from rule `map` : $t_f\times[R]\rightarrow[R']$ where $R'$ is obtained from $R$ by replacing each type $t$ with a type $s$ such that **app** : $t_f\times t\rightarrow s$. As with the concatenation operator, we can choose an arbitrary representative $R$ without changing the result, and even avoid computing it, by unfolding the recursive types.

This typing is exact, provided that a function is allowed, in one way or another, not to return the same value when applied to the same argument (i.e. with a non-deterministic choice operator, a random generator, edge effects, ...). To see why that is necessary, consider the type $(b_c \rightarrow \mathbb{1})\times[b_c*]$ where $b_c$ is the singleton type associated with the constant $c$. The type computed for the result of `map` applied to a value of that type is $[\mathbb{1}*]$. However, if the functions always return the same result for the same argument, it is not possible to reach all the

values of this type with an argument of `map` of type $(b_c \to \mathbb{1})\mathbf{\times}[b_c*]$. It is easy to see how to do this, for example with a non-deterministic choice mechanism, or with a counter (incremented with each call to the function).

**Remark 10.5** *The fact that functions are seen as non-deterministic functions is natural in a programming language, and this assumption was made in the model definition (see the comments that follow Definition 4.2).*

In a type system with ML-like parametric polymorphism, the operator `map` can usually be defined as a recursive function. The reason why ℂDuce has a specific operator is not only the lack of parametric polymorphism in the type system, but also the lack of precision of an ML-like typing scheme for `map`. Indeed, the specific typing for operator `map` gives for example:

$$\texttt{map} : ((t_1 \mathbf{\to} s_1)\mathbf{\wedge}(t_2 \mathbf{\to} s_2))\mathbf{\times}[t_1* \quad t_2?] \to [s_1* \quad s_2?]$$

In other words, when an overloaded function is applied to every element of a non-uniform sequence, it acts differently on the different types present in the sequence. Therefore, the type of the result reflects the order (and arity) of the elements of the input sequence, whereas an ML-like type scheme of the form $\forall\alpha.\forall\beta.(\alpha\mathbf{\to}\beta)\mathbf{\times}[\alpha*] \to [\beta*]$ would impose an approximation on the input type by a uniform sequence type ($[(t_1|t_2)*]$ in the above example), therefore we would lose in precision for the result ($[(s_1|s_2)*]$). The gain in precision obtained with our exact typing has proved itself to be useful in practice for writing XML transformations.

**Remark 10.6** *In fact, the ℂDuce language has a specific construction for* `map`*:*

$$\texttt{map}\ e\ \texttt{with}\ B$$

*where $B$ is a set of pattern matching branches $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$. This avoids having to explicitly write a function (and a pattern match), especially its interface. To type this construction, after finding a type $t$ for $e$, we choose a regular expression for types $R$ such that $t \simeq [R]$ and we apply the typing of the pattern macthing of $B$ to each type that appears in $R$ in order to produce a new regular expression $R'$.*

*This typing depends on the choice of $R$, and there is no choice that provides the most precise type in general (and no schema that captures all the possible types). For example, suppose $e$ has the type $[0\text{—}\infty]$, and that $B = x \to x + x$. We can take $R = 0\text{—}\infty$, but also some finer regular expressions, such as $R = (0|1|\ldots|n|((n+1)\text{—}\infty))$ (the alternation $|$ is the regular expression constructor). With such a decomposition, the branch $x \to x + x$ will be typed $n + 2$ times, and the exact typing of the addition gives a result type $0|2|\ldots|2n|((2n+2)\text{—}\infty)$. We see that we can get arbitrarily accurate types, but the exact set of values that can be obtained is not representable. It is necessary to be able to* specify *a decomposition in regular expression that guarantees the expected properties for typing (elimination of subsumption and admissibility of the intersection rule). One way to do this is by producing the automaton associated with a subtype of $\mathbb{1}_{\mathbf{seq}}$, using a function $\pi$ that associates to a type $t \leq \mathbb{1}_{\mathbf{prod}}$ the set of types $(t_1, t_2)$ such that the Cartesian product $t_1\mathbf{\times}t_2$ is maximal among the sub-types of $t$ (i.e., if $t_1\mathbf{\times}t_2 \leq t'_1\mathbf{\times}t'_2 \leq t$, then $t_1 \simeq t'_1$ and $t_2 \simeq t'_2$), by choosing one representative for each maximal product. This construction was mentioned after Theorem 4.36.*

The language has other iterators that we will not present here.

### 10.3.6 Strings

ℂDuce does not have a specific base type to represent strings. Those are simply character sequences. The `String` type is defined as [`Char`∗]. We also define `PCDATA` as the regular expression `Char`∗.

The string [′`a` ′`b` ′`c`′] can be written [′`abc`′], or also "`abc`". A string literal can also be seen as a singleton type.

The choice to see strings as sequences allows us to seamlessly reuse sequence operators (concatenation, iteration), as well as regular expression types and patterns to finely specify subtypes `String`, and to extract information from strings by pattern matching.

## 10.4 XML

ℂDuce is able to represent XML documents as values of the language. An XML element has the form:

`<tag attr`$_1$`="..." ... attr`$_n$`="..."> ... </tag>`

The content, between the tag $<$ `tag`... $>$ and the tag $<$ /`tag` $>$ is a sequence that mixes characters and XML elements. Such an element is encoded in ℂDuce by a value of the form:

`<(‘tag) {attr`$_1$`="..."; ...; attr`$_n$`="..."}>[ ... ]`

or more simply, with the conventions we already introduced:

`<tag attr`$_1$`="..." ... attr`$_n$`="..."}>[ ... ]`

For example, the XML document

```
<animal kind="insect">
  <name>Bumble bee</name>
  <comment>There are about <strong>19</strong> different species
           of bumblebee.</comment>
</animal>
```

is encoded, after deleting some space characters, with:

```
<animal kind="insect">[
  <name>['Bumble bee']
  <comment>['There are about ' <strong>['19'] ' different species of bumblebee.']
]
```

The type of all XML documents can be defined as:

```
type XML = <(Atom) { _ = String }>[ (Char | XML)* ]
```

In ℂDuce, we can write subtypes that capture an important fragment of the constraints imposed on documents by specifications such as DTD or XML-Schema, for example:

```
type Animals = <animals {||}>[ Animal* ]
type Animal = <animal {|king=String|}>[ <name>String <comment>Text ]
type Text = [ TextItem* ]
type TextItem = Char | <strong>Text
```

(The braces $\{|\ldots|\}$ prohibit the presence of attributes other than those mentioned.)

A utility program `dtd2cduce` automatically translates a DTD into such a set of type declarations. XML-Schema specifications can also be imported; the type of the values validated against such a specification is not necessarily a subtype of `XML`. For example, if the specification indicates that a certain attribute is an integer (type `xs:int`), then value of the corresponding field, in the document obtained after validation, will be an integer CDuce (type `Int`) and not a string (type `String`), and this is reflected in the static type associated with XML-Schema.

There are predefined functions (in fact, operators), of type `String→XML` and `XML→String` that read an XML document contained in a string, and reciprocally. In the case of a *parsing* error, if a string does not represent a well-formed XML document, an exception (see below) is raised to report the error. In fact, to improve error messages, XML documents can be loaded directly from an external file (or URL).

## 10.5 Derivative constructions

**Local binding** The construction `let` $p = e_1$ `in` $e_2$ is syntactic sugar for:

$$\texttt{match}\ e_1\ \texttt{with}\ p \to e_2$$

**Type constraint** The type constraint operator $(\_ : t)$, for a type $t$, can be defined as syntactic sugar for the application of function

$$\texttt{fun}\ (t\texttt{→}t)\ x \to x$$

If we prefer to see it as an operator $o_t$, the typing relation $o_t : \_\texttt{→}\_$ is given by $o_t : t_1\texttt{→}t_2 \iff t_1 \leq t \leq t_2$. The semantics is the identity. The typing is of course not exact, and that is the goal: this operator makes it possible to forget some type information, which can be useful to document some code, to develop programs (by improving the localization of error messages), or to simplify the task of the typechecker when the types manipulated become too complex and uselessly accurate.

**Curryfied abstractions** The syntax below makes it easy to write currified functions:

$$\texttt{fun}\ f(x_1 : t_1)\ldots(x_n : t_n) : s = e$$

The identifier $f$ is optional. For $n = 1$, this expression is translated into:

$$\texttt{fun}\ f(t_1\texttt{→}s)\ x_1 \to s$$

For $n \geq 2$, the translation is given by:

$$\texttt{fun}\ f(t_1\texttt{→}(\ldots(t_n\texttt{→}s)\ldots))\ x_1 \to (\texttt{fun}\ (x_2 : t_2)\ldots(x_n : t_n) : s = e)$$

For example, the complete translation of `fun` $f(x_1 : t_1)(x_2 : t_2) : s = e$ is:

$$\texttt{fun}\ f(t_1 \texttt{→} (t_2 \texttt{→} s))\ x_1 \to (\texttt{fun}\ (t_2 \texttt{→} s)\ x_2 \to e)$$

**Booleans** The two atom constants ‘`true` and ‘`false` are used to represent boolean. The type `Bool` est is predefined as ‘`true` | ‘`false`. The construction `if` $e$ `then` $e_1$ `else` $e_2$ is translated into:

$$\texttt{match}\ e\ \texttt{with}\ \texttt{`true} \to e_1 \mid \texttt{`false} \to e_2$$

The boolean operators are defined from this construction, which gives them a lazy semantics regarding their second argument (for binary operators). For example, the logical conjunction is defined by:

$$e_1 \&\& e_2 \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{`false}$$

## 10.6 Imperative traits

The ℂDuce language extends the purely functional core of Chapter 5 with some imperative traits coming from ML. The way to extend operational semantics to account for these constructions is classic, and we are not going to describe it.

The value [] (in other words ‘`nil`) and the associated singleton type are used as the `unit` type in ML, to represent the argument or the result of functions that operate with side effects.

The sequencing operator $e_1; e_2$ first evaluates $e_1$ (whose type must be []) then $e_2$ and outputs the result of $e_2$. The small-step semantics, defined in Section 5.5 does not completely specify the evaluation order of the language; for example, the implementation can be evaluate in arbitrary order the two sub-expressions $e_1$ and $e_2$ in an expression $(e_1, e_2)$. For a record expression $\{l_1 = e_1; \ldots; l_n = e_n; \_ = e_0\}$, the expression $e_0$ can be evaluated an arbitrary (finite) number of times.

We are not going to describe the usual I/O operations, accessing the operating system, ...

**References** References are typed memory cells, whose content can be extracted and modified. To avoid introducing a new category of constructors, we will define the reference type, denoted by `ref` $t$, as:

$$\{|\ \texttt{get} = [] \to t;\ \texttt{set} = t \to []\ |\}$$

In other words, a reference is seen as a two-field record (an object with tho methods, in the terminology of oriented-object programming languages), that correspond to the two operations can be done on references (reading and assigning content).

We introduce an operator $\texttt{ref}_t$ that takes a value $v$ of type $t$, creates a memory cell initialized with the value $v$, and outputs a record of type `ref` $t$ whose methods are "magic" functions (non-definable in the language) which actually access the memory cell.

The assignment $e_1 := e_2$ is syntactic sugar for $e_1$.`set` $e_2$. The access $!e$ is syntactic sugar for $e$.`get`[].

It should be noted that the two functions `get` and `set` of a value of type `ref` $t$ do not necessarily come (directly) from a call to the reference constructor $\texttt{ref}_t$. It is possible, for example, to create values of type `ref` $t$ that modify an actual reference by printing a trace of the accesses made. We can also define uninitialized references in the following way (for example, for the type `Int`):

```
type Int_option = Int | `none
type Int_ref = ref Int_option
let new_ref(_ : []) : Int_ref =
  let r = ref Int_option `none in
  { get =
      (fun (_ : []) : Int = match r.get [] with
            | `none -> raise "Empty reference !"
            | x -> x);
    set = r.set
  }
```

If we try to access the content of an uninitialized reference, an exception gets raised (see the next paragraph). Note that in the definition of the `set` method, we use the subsumption authorized by the subtyping:

$$\texttt{Int\_option}\texttt{→}[] \leq \texttt{Int}\texttt{→}[]$$

**Exceptions** The semantics of the exceptions of ℂDuce is similar to that of ML. The content of an exception can be any value. We raise an exception $v$ using the construction `raise` $v$ (whose type is $\mathbb{0}$) and we catch an exception by pattern matching:

$$\texttt{try}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

The branches of the pattern match are typed with the input type $\mathbb{1}$ (because $e$ can lift any exception *a priori*). An implicit branch $x \to \texttt{raise}\ x$ added at the end of the pattern match raises the exception again if no branch can catch it.

## 10.7 An example

Figure 10.1 gives an example of a ℂDuce program, that defines a unique XML transformation function, called `split`.

The program starts with type declarations. The type `Person` represents the genealogical tree of a person's progeny. A person's gender is given by the value of an attribute. The `split` function transforms a value of that type into a tree in which, at every level, the sons and daughters are grouped. The gender of people is indicated by an element label. In addition, the names of the people, which were contained in an XML element, are found in an attribute.

The function is a pattern match with a single branch. The pattern allows the person's name to be extracted in one fell swoop (in the variable `n`), its gender (variable `g`), and the list of his children is split in two: the variable `mc` captures all the sons, and the variable `fc` captures all the daughters. In the body of the branch, we begin by calculating the label that corresponds to the sex of the person. We then recursively apply `split` to all of the sons, and all of the daughters, using the iterator `map`. Finally, we build the XML element to be outputted.

```
type Person = FPerson | MPerson
type FPerson = <person gender = "F">[ Name Children ]
type MPerson = <person gender = "M">[ Name Children ]
type Children = <children>[ Person* ]
type Name = <name>[ PCDATA ]

type Man = <man name=String>[ Sons Daughters ]
type Woman = <woman name=String>[ Sons Daughters ]
type Sons = <sons>[ Man* ]
type Daughters = <daughters>[ Woman* ]

let fun split (MPerson -> Man ; FPerson -> Woman)
  <person gender=g>[
    <name>n
    <children>[(mc::MPerson | fc::FPerson)*]
  ] ->
      let tag = match g with "F" -> `woman | "M" -> `man in
      let s = map (split,mc) in
      let d = map (split,fc) in
      <(tag) name=n>[ <sons>s <daughters>d ]
```

Figure 10.1: A program ℂDuce

**Expressive power of pattern matching** This example shows the expressive power of pattern matching, which is able to do a complex extraction of information in a single pattern. We could go further, and directly compute the `tag` label in the pattern, by using pattern constants:

```
<person gender=("F" &(tag := `woman)) | ("M" &(tag := `man))>[
  <name>n
  <children>[(mc::MPerson | fc::FPerson)*]
]
```

or, setting aside this calculation:

```
<person>[
  <name>n
  <children>[(mc::MPerson | fc::FPerson)*]
]
&((FPerson &(tag := `woman)) | (MPerson &(tag := `man)))
```

**Typing** The interface of the function shows two arrow types, which is a more precise type than `Person -> Man | Woman`. This greater precision is necessary to be able to properly type the body of the function. Indeed, when the `map` operator recursively applies the function to all elements of `mc`, that are of type `MPerson`, it outputs a sequence of elements of type `Man`. Similarly, starting from `fc`, we obtain elements of type `Woman`. This shows that the contents of the `<sons>` and `<daughter>` elements are indeed, respectively, of type `[ Man*`

`]` and `[ Woman* ]`, which is necessary to obtain a value of acceptable type in output. If we only had the type `Person -> Man | Woman` for function `split`, the typechecker would give the type `[ (Man | Woman)* ]` to the variables `s` and `d`, and the typing would fail.

The body of the function is typed twice: once under the assumption that the argument is of type `MPerson`, once under the assumption that it is of type `FPerson`. The typechecker has to check that the label of the outputted element is indeed either `‘man` or `‘woman` depending on the case. This is done in the following way. First, the exact typing of pattern matching gives a singleton type to the variable `g` . Then, in the pattern matching used to compute `tag`, only one of the two branches contributes to the type of the result (the pattern matching typing rule makes it possible to ignore the branch that is not being used). Hence we find a singleton type for `tag`, and thus we can check that the label of the returned item is the one expected.

**Compilation of pattern matching** The pattern uses a declarative style: it indicates how to separate the list of children in two, by using the types `MPerson` and `FPerson`. The compiler realizes that the two types can in fact be distinguished simply by looking at the value of the `gender` attribute. If the attribute has value `"F"`, the element is of type `FPerson`, otherwise it is of type `MPerson` (because it is necessarily of one of the two types, given the type of the arguments of the function). It is really the way the produced code distinguishes between types. A naive compilation scheme would need to go through the entire XML tree to see if the element is of type `FPerson` or if it is of type `MPerson`.

In addition, the tests on the labels of the XML elements `person`, `name`, `children` are only there to simplify the rereading of the code. The compiler knows that those tests are verified, and it does not generate any code for them.

We see on this example that the efficient compilation of the typing, that uses static types, allows the programmer to use a programming style that is both more declarative (with an "abstract" use of the types `MPerson`, `FPerson`, instead of a concrete method to distinguish them), and more informative (redundant label tests), without reducing the efficiency of the code produced.

# Chapter 11

# Implementation techniques

In this chapter, we informally describe some techniques used in the implementation of ℂDuce.

## 11.1 Typechecker

**Restriction of rule** (*abstr*) We have introduced (Section 5.7) a notion of schema to symbolically represent the set of types of an expression. An alternative solution to obtain a typing algorithm is to restrict the (*abstr*) typing rule by prohibiting negative arrow types (i.e. by imposing $m = 0$). In the type system obtained, the best type for an expression $e$ for a given typing environment $\Gamma$ (under certain assumptions about the type of operators) can be easily computed. Of course, this system allows fewer programs to be typed than the schema-based algorithm. The characteristic situation is the one where we pattern match an abstraction with a negative arrow type:

$$\texttt{match (fun (Int→Int)}\ldots\texttt{) with ¬(Char→Char)} \to 0$$

To be able to type this expression, we have to allow negative arrow types (which gives the type ¬(Char→Char) to the abstraction). We have never encountered such a situation in practice, so we have chosen to implement the system without negative arrow types. Of course, this is only the static type system: the dynamic semantics, which can perform dynamic type tests on *values*, are not affected. In particular, the following expression, which is well-typed in the implemented system, evaluates to 0 as expected:

$$\texttt{match (fun (Int→Int)}\ldots\texttt{) with ¬(Char→Char)} \to 0 \mid \_ \to 1$$

In other words, the typing algorithm without negative arrow types is correct with regard to the type system (and it guarantees the absence of type errors during execution), but it is not complete.

**Error localization** The naive implementation of the typechecker in the form of a top-down algorithm (which computes the type of a leaf expression up to the root) does not accurately localize type errors. Consider for example the following expression:

$$\texttt{fun}\ (t\texttt{→}s)\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

A bottom-up typing algorithm will start by computing a type $s_i$ for each $e_i$, then the type for the pattern match (the union of the types obtained for each branche), before checking that each type is indeed a subtype of $s$. If that is not the case, rule (*abstr*) cannot be applied, and an error is raised at the level of the abstraction. But actually the error is inside (at least) one of the branches, that is, the one(s) whose result type is not a subtype of $s$. If the expression $e_i$ is incriminated, we can possibly identify an even more precise sub-expression that generates the type error. For example if $s = \texttt{Int}\times\texttt{Char}$ and $e_i = (1, 2)$, we see that the error comes form the constant 2, and not from the expression $e_i$ overall.

In order to accurately locate errors, we use an algorithm that synthesizes types upwardly, but also propagates a constraint downwards. We can formalize this algorithm in the form of a judgement $\Gamma \vdash e : t \leq t_0$, where $\Gamma, e, t_0$ are inputs, and $t$ is the output of this algorithm (the most precise type for $e$, modulo the restriction on rule (*abstr*)). If this judgment cannot be derived, the algorithm points out the faulty sub-expression, and if not, it returns a type $t$ that is necessarily more accurate than $t_0$. Let us describe the main cases of the algorithm, depending on the form of $e$.

If $e = (e_1, e_2)$, then we start by checking that $t'_0 = t_0 \wedge \mathbb{1}_{\mathbf{prod}}$ is non-empty. If it is empty, which is not necessarily a typing error, then we have to check that one of the two expressions $e_1$ or $e_2$ has type $\mathbb{0}$ and that the other is well-typed. If both expressions are well-typed but none has type $\mathbb{0}$, we raise an error for $e$, which is a pair expression, while the expected type does not contain any pair. If $t'_0$ is non-empty, we compute $t_1$, the most precise type for $e_1$, with the constraint $\pi_1[t'_0]$. For $e_2$, we use as a constraint the largest type $t_2$ such that $t_1\times t_2 \leq t'_0$ (an easy calculation shows that we can take $t_2 = \neg\pi_2[(t_1\times\mathbb{1})\setminus t_0]$). The case of records is treated in a similar way.

If $e$ is an abstraction $\mu f(t_1\to s_1; \dots t_n\to s_n).\lambda x.e$, we start by checking that the intersection of the arrow types of the interface is indeed a subtype of $t_0$. We then compute, for each of those types $t_{i_0}\to s_{i_0}$, the type of the body $e$ of the abstraction, with the constraint $s_{i_0}$ and with the environment extended by the assumptions $(x : t_{i_0})$ and $(f : \bigwedge t_i\to s_i)$. The type of the result can be ignored.

For pattern matching, we compute the type of the matched expression, under the constraint $t'_0$ that is the union $\bigvee \lbag p_i \rbag$ of the types accepted by the patterns of the different branches. An alternative approach is to compute this type without any constraint and to check after the fact that the type obtained is a subtype of this union, which may possibly give a more explicit error message ("non-exhaustive pattern matching"). In both cases, a type is then computed for each branch (except those that cannot succeed), with the same constraint $t_0$ than for the pattern match.

For each operator, we adopt a specific strategy to compute the constraint to be used to type the argument. We can choose not to use the most precise constraints, so as not to introduce too complex constraints (which would give unclear error messages). For example, to type the concatenation $e_1@e_2$ with a constraint $t_0$, we can start by computing the smallest type $t'_0$ such that $t_0 \leq [t'_0*]$ and type both arguments with the constraint $[t'_0*]$. To type an application $e_1e_2$, we start by computing a type $t_1$ for $e_1$ under the constraint $\mathbb{0}\to\mathbb{1}$; we then use the type $\mathrm{Dom}(t_1)$ to type $e_2$ (in fact, we could do better and compute the largest type $t_2$ such that $\mathbf{app} : t_1\times t_2 \leq t_0$).

**Error messages** When a type error is detected, it is usually a constraint $t \leq t_0$ that is not verified, where $t$ is the type computed for a sub-expression $e$ and $t_0$ is the constraint that was used to type $e$. In the error message, we point out that the best type we can compute for the expression is $t$, while the expected type is $t_0$, and to explain why $t$ is not a subtype of $t_0$, we can exhibit a value of type $t \backslash t_0$ (as that type is non-empty, there exists such a value, and we can in fact compute one explicitly).

The typing algorithm can detect ill-typed expressions. It can also detect other types of potential errors, which can give rise to warnings. Here are some examples.

— Empty-type declaration. Consider the following program:

```
type T = <a>[ T+ ]
let fun f(x : T) : Int = ...
```

This program is well-typed, independently from the body of the function. Indeed, the type `T` is empty (because the values are finite objects, and `T` denotes necessarily infinite trees), and the typing rule for pattern matching indicates that it is not necessary to type all the branches that cannot be used (which is the case, for a function, when the argument type is empty). We can assume that the programmer made a mistake by defining the type `T`, and that he instead wanted to write:

```
type T = <a>[ T* ]
```

To detect such errors, the typechecker points out, in the form of a warning, the type declarations that define an empty type.

— Unused branch or pattern. In a pattern match, a branch may not be used (when the input type of the pattern match is disjoint from the type accepted by the pattern), and it is not typed in that case. This is useful for typing an overloaded function, where some branches must be ignored to check some of the constraints given in the interface (remember that the body of an overloaded function is typed once for each arrow type of the interface). Nevertheless, a branch that will never be typed probably corresponds to a programming error. This situation is detected and reported to the programmer. This analysis can be refined to detect unused sub-patterns; to do so, the pattern typing algorithm can be used to detect patterns that systematically receive an empty type as input.

— Unused capture variable. Variables and type names are not syntactically distinguishable. An ambiguity therefore arises when interpreting an identifier $x$ in a pattern: it may be a capture variable or a type constraint pattern, if $x$ is declared earlier in the program (or predefined). The following heuristics are used to remove ambiguity: if the type $x$ is defined, the identifier is interpreted as a type constraint pattern; otherwise, it is interpreted as a capture variable. This heuristic can hide typing errors; consider, for example, the program below:

```
fun ( (Int | Char, Int | Char) -> Int)
| (Inte,Int) -> 0
| _ -> 1
```

In the first branch, the `identifier` is interpreted as a type, and the `Inte` identifier is interpreted as a capture variable. The function is well typed

(no warning is given for the second branch, because it is not useless). However, it can be assumed that the identifier `Inte` is a typo, and that the programmer wanted to type `Int`. To detect this type of error, we simply give a warning when a capture variable does not appear in the body of the branch (this is the case for the `Inte` variable above).

## 11.2 Representation of values

For performance reasons, in the implementation, the algebra of values is extended with a number of derived forms.

**Strings** In ℂDuce, string are conceptually sequences. Hence, the string `"abc"` is in fact the value $[\texttt{'a}\ \texttt{'b}\ \texttt{'c}]$, that is, $(\texttt{'a}, (\texttt{'b}, (\texttt{'c}, \texttt{`nil}())))$. This encoding is very costly in memory. During execution, the implementation manipulates values of the form $\texttt{string}(s, i, j, v)$ where $s$ is an array of characters, $i$ and $j$ are two index in $s$, and $v$ is a sequence value. This value actually represents the sequence made of items of $s$ between the $i$ and $j$ indexes, followed by the $v$ sequence. We then have the equivalence:

$$\texttt{string}(s, i, j, v) \equiv \begin{cases} (s[i], \texttt{string}(s, i+1, j, v)) & \text{if } i < j \\ v & \text{if } i = j \end{cases}$$

This derived form provides a compact memory representation for strings, or more generally for consecutive character sub-sequences. Note that the array $s$ is not copied in the equivalence above: unfolding the compact representation character by character is done in constant time (we simply have to increment an integer). This representation also makes it possible to "jump" over a sequence of characters in a sequence. Hence, if we apply the pattern $[((x :: t)|\_)*]$ (that captures all the elements of type $t$ of a sequence) with $t \mathbin{\texttt{\&}} \texttt{Char} \simeq \mathbb{0}$, and we end up on $\texttt{string}(s, i, j, v)$, we can jump directly to $v$, without considering all the characters of $s$ between $i$ and $j$. Similarly, the result for a sequence capture variable, when the pattern matched value is of the form $\texttt{string}(s, i, j, v)$, can be represented by using compact representations of the form $\texttt{string}(s, i', j', v')$. We can use the following equivalence to regroup consecutive fragments of the same array $s$:

$$\texttt{string}(s, i, j, \texttt{string}(s, j, k, v)) \equiv \texttt{string}(s, i, k, v)$$

**Lazy concatenation** The concatenation operator `@`, naively implemented, has an execution time that is linear in the length of its first argument. Indeed, the concatenation of the sequence $[v_1 \dots v_n]$ and of the sequence $v'$ is $(v_1, (v_2, \dots, (v_n, v') \dots))$, hence we have to build $n$ nested pairs. A program that constructs a sequence by concatenating the elements one by one at the end thus has a quadratic complexity (only for concatenations) in the length of the result.

We introduce a new form of value at execution, denoted by $v_1 \texttt{@} v_2$, that symbolically represents the concatenation of $v_1$ and $v_2$ (two sequences). This value can be built in constant time from $v_1$ and $v_2$. To iterate on a value $v_1 \texttt{@} v_2$, we simply have to iterate over $v_1$, then over $v_2$. On the other hand, when

evaluating pattern matching, it is sometimes necessary to normalize $v_1 @ v_2$; for this, we use the following equivalences:

$$\begin{array}{lcl} \text{`}\texttt{nil}@v & \equiv & v \\ (v_1, v_2)@v & \equiv & (v_1, v_2@v) \end{array}$$

These equivalences allow the first element of a sequence to be exposed, that is, to put it in the form $(v_1, v_2)$. This is an incremental normalization. To normalize $v'@v$, when $v'$ is itself a symbolic concatenation, we start by normalizing $v'$ before applying the above rules.

In fact, we can see any value as a binary tree whose nodes are @, and leafs are values which are not of the form $v_1@v_2$. It is possible to normalize such a tree in time linear in the size of the result. It is then a global normalization, which avoids building unnecessary intermediate structures.

To avoid having to evaluate the same concatenations several times, we can modify in place a value $v_1@v_2$ with its normalized form when computing it (either incrementally, or globally).

**Atom representation** Atoms are pairs of symbols (made of an XML *namespace* and a local name). To reduce the memory occupied by documents, and to speed up operations on atoms (equality and comparison used by pattern matching decision trees), they are represented internally with optimal sharing (obtained by *hash-consing*) and a unique digital identifier.

**Comparison with other works** The Xtatic project independently proposed the same kind of techniques [GLPS04] to lazily evaluate the concatenation of sequences, and to represent strings in a compact way.

A completely different approach was adopted to represent values in the XHaskell project [LS04a, LS04b]. XDuce, ℂDuce and Xtatic use a uniform representation for the value algebra; it means that a value is represented independently from its static type. In that case, the subsumption rule is transparent during execution: we can indeed see a value of type $t$ as a value of type $s$ when $t \leq s$, without applying any transformation to it. XHaskell proposes to associate to a regular expression type a concrete representation of the values of this type. Thus, each regular expression type is associated with a Haskell type, in a compositional way. For example, the Kleene star of regular expressions is translated into Haskell's list constructor, and the choice operator | is translated into a sum type with two constructors. The concrete representation of the result of a language expression then depends on its static type. The subsumption rule therefore requires changing the representation of values, and to do this, we have to make use of the subtyping algorithm to extract from the proof of an instance $t \leq s$ a coercion function between the representation of $t$ and that of $s$ (in XHaskell, this is done by encoding the subtyping algorithm in an extensive system of Haskell type classes).

The Xen/C$\omega$ language [MS03] is an extension of C$^\#$ with structural types that represent XML documents. Again, each XML type of the language is associated with a type in the implementation architecture (Microsoft .NET's virtual machine CLR), and coercions have to be introduced to account for subtyping. Unlike XHaskell, however, Xen is not trying to implement a set-theoretic-inspired subtyping between regular expression types.

The choice between a uniform representation as in XDuce, ℂDuce, Xtatic and a specialized-by-types representation as in XHaskell is delicate. In addition to its simplicity, the uniform representation allows for subsumption to be implemented "for free" between subtypes, while a specialized representation may require to completely copy the values. The specialized representation, on the other hand, allows faster access to data within complex structures (direct addressing); it is also more compact, because part of the structure of values is stored implicitly in their static type, and allows better integration with general-purpose languages.

## 11.3 Representation of Boolean combinations

Many algorithms presented in that thesis work in a given socle $\supseteq$. Even though we have formally studied the complexity of those algorithms, it depends crucially on the size of the socle, that is, in the number of different types that can be obtained from the types given in the problem, by saturation with boolean combinations and by decomposition of the atoms inside of types. To ensure that this number is always finite, we have introduced in Section 3.1 a certain representation of Boolean combinations (normal disjunctive form).

In this section, we propose alternative techniques to optimize this representation, in order to decrease the number of different types considered in the algorithms, and therefore their complexity.

### 11.3.1 Simplification by boolean tautologies

The representation of boolean combinations can be optimized by taking into account a number of boolean tautologies. For example, representation in normal disjunctive form already takes into account the commutativity and associativity of union and intersection. We can go further and impose stronger normalization constraints. Formally, we consider a rewriting relation $\rightsquigarrow$ on the $\mathcal{B}X$ such that for every set-theoretic interpretation of $\mathcal{B}X$ into a set $D$, if $t \rightsquigarrow t'$, then $[\![t]\!] = [\![t']\!]$ (the interpretation does not change when simplifying the boolean combination). The simplifications considered reduce the size of boolean combinations, which ensures their termination. All manipulated boolean combinations can be normalized, reducing their size, and also their number, as more combinations can be identified with each other. We give some examples of possible simplifications.

In a normal disjunctive form, any "line" that contains the same atom in positive form and in negative form at the same time can be removed. This translates into the simplification rule:

$$\frac{P \cap N \neq \emptyset}{\{(P,N)\} \cup t \rightsquigarrow t}$$

Another simplification consists in noting that if in a disjunctive normal form we can go from a line to another by adding literals, then the obtained line is useless (because its set-theoretic interpretation is included in the original line):

$$\frac{P' \subseteq P \quad N' \subseteq N}{\{(P,N),(P',N')\} \cup t \rightsquigarrow \{(P',N')\} \cup t}$$

We can merge two lines that are equal except for an atom that appears in positive form in one and negative in the other. This corresponds to the

tautology: $(x \wedge A) \vee (\neg x \wedge A) \simeq A$. We write:

$$\frac{}{\{(P \cup \{x\}, N), (P, N \cup \{x\})\} \cup t \rightsquigarrow \{(P, N)\} \cup t}$$

A last example of a simplification consists in deleting the literal $\neg x$ in $(x \wedge B) \vee (\neg x \wedge A \wedge B)$:

$$\frac{P' \subseteq P \quad N' \subseteq N}{\{(P, N \cup \{x\}), (P' \cup \{x\}, N')\} \cup t \rightsquigarrow \{(P, N), (P' \cup \{x\}, N')\} \cup t}$$

and symmetrically:

$$\frac{P \subseteq P' \quad N \subseteq N'}{\{(P, N \cup \{x\}), (P' \cup \{x\}, N')\} \cup t \rightsquigarrow \{(P, N \cup \{x\}), (P', N')\} \cup t}$$

The more simplifications we can detect, the more we reduce the size and number of the boolean combinations manipulated. Algorithms that manipulate types are then more efficient. But detecting these simplifications can be costly in itself. So there is a trade-off to be made in the implementation.

### 11.3.2 Disjoint atoms

Assuming as given an orthogonality relation between atom types, denoted $\perp$, such that two orthogonal atoms are necessarily disjoint ($x \perp y \Rightarrow x \mathbin{\&} y \simeq \mathbb{0}$). Additional simplifications can then be considered. For example, a line in a normal disjunctive form can be removed if it contains two orthogonal atoms in positive position:

$$\frac{x \perp y}{\{(\{x, y\} \cup P, N)\} \cup t \rightsquigarrow t}$$

Similarly, we can remove from a line any negative literal whose atom is orthogonal to a positive atom of the same line:

$$\frac{x \perp y}{\{(\{x\} \cup P, \{y\} \cup N)\} \cup t \rightsquigarrow \{(\{x\} \cup P, N)\} \cup t}$$

What remains to be done is to define the binary relation $\perp$. Of course, we can already establish that two atoms of different kinds (e.g. an arrow atom and a product atom) are orthogonal, and the same for two disjoint base types (for the representation of the combinations of base types, see Section 11.3.4). We can also unfold the product types. For example, we know that $t_1 \times t_2$ and $t'_1 \times t'_2$ will be disjoint as soon as $t_1$ and $t'_1$ (or $t_2$ and $t'_2$) are disjoint. The subtyping algorithm can also be used to detect disjoint types: obviously, this has a cost and a circularity appears (because the subtyping algorithm must be able to compute boolean combinations, hence we have to be careful not to get into a loop).

### 11.3.3 Alternative representation: decision trees

**Binary decision trees** Rather than trying to simplify boolean combinations represented in normal disjunctive form, other representations that may sometimes be more compact can be used directly.

A classic technique is to represent boolean combinations with binary decision trees. Formally, these are the objects generated by the following grammar:

$$t \quad := \quad 0 \mid 1 \mid (a?t_1 : t_2)$$

We modify the definition of a set-theoretic interpretation; the interesting condition is:

$$[\![(a?t_1 : t_2)]\!] = ([\![a]\!] \cap [\![t_1]\!]) \cup (D \backslash [\![a]\!] \cap [\![t_2]\!])$$

In general, we make sure that each atom appears at most once on each branch, which can be achieved in practice by choosing a total order $\preceq$ on the atoms and requiring that the atoms on each branch appear in strict increasing order. This ensures that if the number of atoms is finite, then so is the number of boolean combinations. We can define the union, intersection and complement operators directly on this representation. For example, if $t = (a?t_1 : t_2)$ and $t' = (a'?t'_1 : t'_2)$, we define $t \vee t'$ by:

$$\begin{cases} (a?t_1 \vee t'_1 : t_2 \vee t'_2) & \text{if } a = a' \\ (a?t_1 \vee t' : t_2 \vee t') & \text{if } a \preceq a' \\ (a'?t \vee t'_1 : t \vee t'_2) & \text{if } a' \preceq a \end{cases}$$

If the atom at the root of $t'$ is larger than all of the atoms of $t$, for example, we see that $t'$ is "copied" in each leaf of $t$; hence, the size of the representation of the result of operator $\vee$ is not linear in the size of its arguments anymore.

Here are the rules to compute $t \wedge t'$:

$$\begin{cases} (a?t_1 \wedge t'_1 : t_2 \wedge t'_2) & \text{if } a = a' \\ (a?t_1 \wedge t' : t_2 \wedge t') & \text{if } a \preceq a' \\ (a'?t \wedge t'_1 : t \wedge t'_2) & \text{if } a' \preceq a \end{cases}$$

Various simplifications can be introduced for the binary decision tree representation, for example: $(a?t : t) \rightsquigarrow t$.

It is easy to return to a normal disjunctive form representation from a binary decision tree (each branch that ends on a 1 leaf gives a "line"). This allows this alternative representation to be used internally for storage and computations on boolean combinations, while maintaining a common interface to observe these objects.

**Ternary decision trees** A problem with binary decision trees is that the size of trees can grow exponentially while computing the boolean union operator. One solution is to use ternary decision trees:

$$t \quad := \quad 0 \mid 1 \mid (a?t_1 : t_2 : t_3)$$

The interesting condition in the definition of a set-theoretic interpretation is then:

$$[\![(a?t_1 : t_2 : t_3)]\!] = ([\![a]\!] \cap [\![t_1]\!]) \cup [\![t_2]\!] \cup (D \backslash [\![a]\!] \cap [\![t_3]\!])$$

It is then possible to define the boolean operators on this representation, and the size of the tree that represents the union of two trees becomes linear in

the sum of the sizes of these trees. If $t = (a?t_1 : t_2 : t_3)$ and $t' = (a'?t'_1 : t'_2 : t'_3)$, we define indeed $t \vee t'$ by:

$$\begin{cases} (a?t_1 \vee t'_1 : t_2 \vee t'_2 : t_3 \vee t'_3) & \text{if } a = a' \\ (a?t_1 : t_2 \vee t' : t_3) & \text{if } a \preceq a' \\ (a'?t'_1 : t \vee t'_2 : t'_3) & \text{if } a' \preceq a \end{cases}$$

Here are the rules to compute $t \wedge t'$:

$$\begin{cases} (a?(t_1 \vee t_2) \wedge (t'_1 \vee t_2) : 0 : (t_3 \vee t_2) \wedge (t'_3 \vee t_2)) & \text{if } a = a' \\ (a?t_1 \wedge t' : t_2 \wedge t' : t_3 \wedge t') & \text{if } a \preceq a' \\ (a'?t \wedge t'_1 : t \wedge t'_2 : t \wedge t'_3) & \text{if } a' \preceq a \end{cases}$$

(Termination is guaranteed because the representation of $t_1 \vee t_2$ is smaller than that of $t$.) We see that it is necessary, in a case, to go back to a binary representation to compute the intersection (that is, to have 0 as a neutral component); this is done by using the equivalence $(a?t_1 : t_2 : t_3) \simeq (a?t_1 \vee t_2 : 0 : t_3 \vee t_2)$. Similarly, to compute the complement $\neg t$ where $t = (a?t_1 : t_2 : t_3)$, we take

$$(a?\neg(t_3 \vee t_2) : 0 : \neg(t_1 \vee t_2))$$

We can consider several simplifications, such as $(a?t_1 : t_2 : t_3) \rightsquigarrow t_1|t_2$ if $t_1 = t_3$, or $(a?t_1 : 1 : t_3) \rightsquigarrow 1$.

### 11.3.4 Representation of base types

We use specific representations for boolean combinations of basic types. Thus, any boolean combination of integer or character types (intervals) can be represented in a unique way in the form of a union of maximal intervals (thus disjoint). Boolean combinations of atom types are represented in a unique way using one of the forms $a_1 \vee \ldots \vee a_n$ or $\neg(a_1 \vee \ldots \vee a_n)$, each $a_i$ being of the form $ns : ln$ or $ns : *$ and two-to-two disjoint (meaning we cannot have $ns : ln$ and $ns : *$ simultaneously for the same namespace $ns$).

These finite unions are represented as sets (not sequences), which ensures the uniqueness of the representation (hence the number of types considered is minimized). Boolean operations can be computed directly on these representations, and the emptiness test is trivial.

## 11.4 Interface with Objective Caml

The implementation of ℂDuce has an interface system with the Objective Caml language. This interface allows:

— to reuse from ℂDuce programs most of the existing OCaml libraries;
— to developed hybrid projects, in which some OCaml and ℂDuce modules cohabit.

The choice of Objective Caml is natural since it is the implementation language for ℂDuce (in particular, ℂDuce programs use Ocaml's memory management system). The two languages also have a certain number of common traits: they are both higher-order functional languages, with the same kind of semantics - strict, with mutable-in-place values, and exceptions. Many OCaml libraries encapsulate lower-level libraries (usually written in C), and offer them

an interface that is abstract enough for ℂDuce (for example, they handle memory management, exception error reporting, and present enough information in the types, which is not the case with C headers). The interface with OCaml indirectly allows these libraries to be used (for example, to access databases, digital computing libraries, system libraries, ...).

Due to the implementation of ℂDuce in OCaml, their memory management is common, hence there is no problem on that side; for example, there is no need to have two different *garbage collectors* interacting.

One of the major challenges in interfacing two strongly-typed languages is linking the two type systems. Consider the abstract situation of having two languages $L_1$ and $L_2$, each using an implicit and transparent subtyping relation in their semantics. Consider also a translation $t \mapsto \tilde{t}$ from the types of $L_1$ to the types of $L_2$. The idea is that a value of the $L_1$ language, of type $t$, can be automatically translated into a value of type $\tilde{t}$ to be used in $L_2$. This relation has to be compatible with the subtyping relations: if $t_1 \leq t_2$, then any value of type $t_1$ can be seen as a value of type $t_2$, and therefore automatically translated into a value of type $\tilde{t_2}$. This value can also be directly translated into a value of type $\tilde{t_1}$. Then, to preserve the semantic transparency of subsomption, we need to have $\tilde{t_1} \leq \tilde{t_2}$ as soon as $t_1 \leq t_2$. As a result, since OCaml does not have implicit subtyping (apart from the identity), a function translating ℂDuce types to OCaml types is necessarily constant. This function is a uniform translation that forgets any type information, meaning that seen from OCaml, all ℂDuce values have the same type, say `CDuce.value`. In fact, the implementation of ℂDuce naturally uses such an OCaml type to represent values during execution, and this representation is visible from OCaml programs. However, it is hardly a typed interface, since any type information is lost in this representation. Instead, we are going to define a translation the other way, from OCaml types to ℂDuce types. Since there is no implicit subtyping in OCaml, the constraint mentioned above is empty, and we have a lot of freedom to define this translation.

The translation of base types does not pose any problem: for example, OCaml's `int` type is translated into an interval type $i$—$j$ where $i$ (resp. $j$) is the smallest (resp. the largest) representable integer in OCaml. Similarly, OCaml's `char` type is translated into the subtype of `Char` that corresponds to the character set iso-8859-1 (the initial segment of Unicode consisting of 256 characters). OCaml's structural types (arrows, products, records) are translated using the similar constructions of ℂDuce. An OCaml sum type declared by:

`type t =` $\mathtt{A}_1$ `of` $t_1$ `|` ... `|` $\mathtt{A}_n$ `of` $t_n$

is translated into the ℂDuce type

(‘$\mathtt{A}_1$,$\tilde{t_1}$) | ... | (‘$\mathtt{A}_n$,$\tilde{t_1}$)

Recursive OCaml types are of course translated into recursive ℂDuce types. A special treatment is reserved for the abstract types of OCaml: these are added to the type algebra of ℂDuce. If $t$ is an abstract type of OCaml, a value of type $\tilde{t}$ is, from ℂDuce's point of view, a black box with the label $t$, and its actual content (an OCaml value of type $t$) remains inaccessible. Keeping the label $t$ on the value is necessary to be able to do pattern matching on types in ℂDuce. We do not handle parametric abstract types.

If the OCaml type $t$ is associated with the ℂDuce type $\tilde{t}$, with the rules above, we have natural translation functions from OCaml values of type $t$ into ℂDuce values of type $\tilde{t}$, and reciprocally. These translation functions allow us to set up the interface system between the two languages.

**OCaml to ℂDuce** The ℂDuce programmer can use an OCaml value `M.v` (`M` is the name of the OCaml module in which the value `v` is defined) in a ℂDuce program. The ℂDuce compiler searches in the compiled interface of `M` the OCaml type of `M.v`, which is $t$, and produces the code that translates it into a ℂDuce value of type $\tilde{t}$, which is the type given to the expression `M.v` in the ℂDuce program.

If the value has a polymorphic OCaml type scheme (a polymorphic function, usually), the programmer has to explicitly instantiate the variables used with ℂDuce types. For example, the type scheme of `List.map` in OCaml standard library is:

$$\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$$

To use this value in ℂDuce, by instantiating $\alpha$ with the type `Int` and $\beta$ with the type [`Int`$*$], the programmer has to write

$$\texttt{List.map}\{\texttt{Int}; [\texttt{Int}*]\}$$

(Schemes are defined modulo alpha-renaming, we give instantiations in the order in which the variables appear in the type scheme, from left to right.) The translation functions at the value level for OCaml type variables instantiated with ℂDuce types are identities.

**ℂDuce to OCaml** A ℂDuce value $v$ can be seen from OCaml as a value of OCaml type $t$, as soon as $v$ is indeed of type $\tilde{t}$. There is no canonical choice for $t$. For example, if $v$ has type (‘`A`, `0`—`10`)|(‘`B`, `0`—`20`), we can see it as a value of OCaml type `t` defined by:

```
type t = A of int | B of int
```

But we do not want to *declare* such a type (recall that two concrete OCaml types, even if they textually have the same definition, are incompatible). For example, if the OCaml already uses a type `s` defined by:

```
type s = A of int | B of int | C of string
```

we may want to see $v$ as a value of that type.

The interface system asks the programmer to explicitly give an OCaml type to the ℂDuce values that he wants to use from ℂDuce modules. This is done by associating to a ℂDuce compilation unit an OCaml interface module. The ℂDuce compiler checks that the ℂDuce values exported by this compilation unit are indeed compatible with the OCaml types that the interface file gives them, and it produces an OCaml module that does the translation from ℂDuce values to OCaml values.

## 11.5 Performance

We have not studied the complexity of the algorithms presented in this thesis. In particular, the subtyping algorithm solves the problem of the inclusion of regular tree languages (given as automatons), which is EXPTIME-complete. Since types actually represent alternating tree automatons, with complement, we should expect a larger theoretical lower bound. The precise study of the complexity of the algorithm obviously depends on the choice of the algorithm that computes inductive predicates (with or without backtracking, see Chapter 7), and the way to represent and compute boolean connectors (see Chapter 3 and Section 11.3).

In practice, on realistic (although relatively small) ℂDuce programs, all the algorithms involved in compiling ℂDuce programs seem to be performing well enough. For example, the program used to generate ℂDuce's web site from an XML description takes less than two tenth of a second to compile, on a Pentium 4 at 2.7 Ghz. About half of that time is spent in the subtyping algorithm, which is called 19956 times (for a total of 70999 internal iterations). The program consists of about 450 lines of ℂDuce code, plus about 300 lines that correspond to the DTD XHTML 1.0 Strict, represented in ℂDuce types. About 3500 internal type nodes are introduced. Other examples confirm the fact, noted by Hosoya [Hos01], that despite the large theoretical complexity of the subtyping algorithm, the various implementation techniques allow us to obtain an efficient algorithm, at least for the XML types encountered in practice.

The performance of ℂDuce programs during execution seems quite satisfactory, in particular thanks to the efficient pattern matching compilation algorithm, and to the techniques used for representing values during execution presented in this chapter.

We have deliberately chosen not to present any performance metrics (or, a *fortiori*, comparisons with other "competing" languages). Such metrics assess both the quality of the implementation and of the implementation language, as well as the efficiency of algorithms. For example, we have observed gains or losses of a factor 2 in efficiency by simply playing with the parameters of Objective Caml's *garbage collector*. Similarly, the removal of dead code resulted in a loss of efficiency of around 20% (probably due to code alignment issues). We believe that performance metrics or comparisons are of little scientific interest. However, we refer the interested reader to a previous publication [BCF03] or to the ℂDuce website to see these kind of metrics.

# Conclusion

# Chapter 12

# Conclusion and outlook

The work presented in this thesis establishes the theoretical basis for the ℂDuce language. In parallel to these theoretical works, I have developed an implementation of the ℂDuce language. A first prototype was made at the beginning of the thesis, and was not released. It tested the feasibility of the algorithms involved. It was completely rewritten, and ℂDuce 0.1 was made public in June 2003, allowing us to introduce the language to a small community of users.

It seems to me that the goal of this thesis, which was to propose a functional programming language adapted to XML, with general-purpose traits and a usable implementation, has been reached. I have proposed solutions to several of the open questions outlined in Hosoya's thesis [Hos01] (*Records, Higher-Order Functions, Improvement of the Type Inference Algorithm, Non-linear patterns, Pattern Optimization*), as well as other extensions to XDuce.

There have been very strong interactions between the implementation work and the directions taken by the theoretical work. Sometimes the implementation even preceded formalization, and a formalization effort after the fact made it possible to better understand and improve the implementation in return. Implementation has played an essential role in the formalism of type algebras (Chapter 3) and it is relatively low-level considerations (sharing of internal structures) that led to the establishment of the theory of Chapter 2. Of course, all the results related to implementation techniques and algorithmic work were motivated by the implementation, and developed in parallel. This thesis is therefore the result of a continuous back and forth between theory and practical implementation.

The implementation was done in the Objective Caml language. This language proved to be a great help in the development of ℂDuce and has kept its promises perfectly as the code evolved consistently and therefore required general changes. Of course, the philosophy of OCaml could only permeate the design of ℂDuce; there are important syntactic and semantic similarities, as well as the presence of an interface (Section 11.4) that allows the languages to interoperate. The implementation of the version 0.2 of ℂDuce, which was released in July 2004, consists in about 20 000 lines of code. It rests on a certain amount of libraries developed by the OCaml user community.

This thesis presents the ℂDuce language itself only briefly and informally (Chapter 10). It is not intended to replace a reference manual or a tutorial for the language. In addition, the language keeps evolving. Interested readers will

refer themselves to other sources of information about ℂDuce [BCF03, Ftct04b, Ftct04a] and its implementation.

## 12.1 Perspectives

**Polymorphism** The question of polymorphism, mentioned in Hosoya's thesis, is still relevant today. The aim is to integrate the kind of parametric polymorphism found in ML into the formalism of the types of XDuce or ℂDuce. One of the technical difficulties is to extend the subtyping relation to polymorphic types, while preserving both the set-theoretic approach and an efficient practical algorithm. Another difficulty is the inference of type variables instantiated during the call of a polymorphic function. Finally, there is a semantic choice to be made, namely whether type variables can be used in a pattern (in which case the semantics of a polymorphic function may depend on the instantiation of type variables, which poses problems for the inference of these instantiations, and which can also have an impact on the representation of values at execution). Preliminary work [HFC05] studying the addition of parametric polymorphism to XDuce was conducted, with Hosoya and Castagna.

**Reconciliation with ML** The approach presented in this thesis and pursued by project ℂDuce was to define a language adapted to the manipulation of XML documents, with specific constructions, but which nevertheless possesses general-purpose characteristics coming from existing languages, such as first-class functions. Another possible direction would have been to integrate the specific characteristics required by XML into a general-purpose language (types and regular expression patterns, in the first place). A natural "target" would be a language of the ML family. The problem would then be to figure out how to bring closer the type system of XDuce/ℂDuce (propagation of types that represent automatons), and the one of ML (inference by unification). A possible starting point would be the HM(X) [OSW99] type system, that extends the one of ML with constraints. Another approach would be to introduce a unique XML type in a ML language, and to "refine" that type (with a regular expression type), using a formalism similar to Dependent ML [Xi98, Xi99].

**Inference** In ℂDuce, functions have to be explicitly annotated with their type. Since these annotations can in fact change the semantics of programs (because patterns are able to test the declared type of a function), it is not possible to remove them completely. However, we may try to infer them in some cases, if we can verify that these functions will not be pattern matched with a type test, or if we accept that the inference algorithm influences program semantics (by making a certain choice). In particular, it would be nice for the programmer not to have to specify the type of "anonymous" abstractions, which are used as arguments to higher-order functionals. Even without considering the fact that annotations change the semantics of functions, it seems inconceivable to achieve "total" inference. One reason is that a function can have an infinite number of different and incomparable arrow types (in addition, in the presence of singleton types, the set of types of a function often completely characterizes the semantics of the function, for example with $\lambda x.x + 1$, or a function of the kind of `map` on sequences). Local inference techniques [PT98, PT00, OZZ01] are a reasonable

starting point. In fact, the structure of the typing algorithm implemented for ℂDuce - and described in the "Error localization" paragraph of Section 11.1 - is close to the bidirectional type propagation used in local inference: it propagates a constraint downwards in the syntax tree. In its current version, it simply uses this constraint (an upper bound on the type of the expression) to better locate type errors (and in the absence of error, it returns a finer type). It could easily be modified to use this constraint to "fill" the annotations to be inferred for abstractions. Thus, the programmer could often avoid having to specify the type of anonymous abstractions used as arguments to higher-order functionals.

**A representation for values depending on type and context** The implementation of ℂDuce uses a uniform representation for all values manipulated during execution. This means that the machine representation of the value determines it entirely without having any information about its static type. XHaskell [LS04b], on the other hand, translates the static XML types of expressions into host language types (Haskell, in this case). Each XML type has its own concrete representation, and in order to be able to use a value of a certain type where a value of a supertype is expected, it is necessary to apply a coercion on the value. This may have a significant cost, but in return, the representation is more compact, and it allows direct access to the middle of sequences, where with the uniform representation, it would be necessary to go through the elements one by one. One approach that would be interesting to consider is to make representation dependent not only on the static type, but also on the context where the value is used. Thus, if we be foresee that a coercion will have to be applied to the value, we can avoid it directly using the expected representation. If there is nothing to predict about the value, a uniform representation can be used as a defect. Obviously, this approach requires a fairly accurate information flow analysis (hopefully we can reuse the work of Benzaken, Burelle and Castagna [BBC03] on the analysis of information flow for safety in ℂDuce).

**Optimal sharing of cyclical structures** The formalism of Chapter 2 allows us to develop the rest of the theory without worrying about the question of sharing between structurally "isomorphic" types. Of course, the type algorithms (especially the subtyping algorithm) are all the more effective as the number of different types manipulated is reduced. It is therefore in our interest to share as many types as possible, as long as this sharing is less expensive than the gain obtained. In the implementation, types are shared during the translation between external syntax types and the internal type algebra, modulo $\alpha$-renaming. Types $X$ `where` $X = (X, X)$`|Int` and $X'$ `where` $X' = (X', X')$`|Int` are therefore translated into (physically) equal types in the implementation, whereas the three recursive types $X$ `where` $X = (X, X)$`|Int`, $Y$ `where` $Y =$ `Int|`$(Y, Y)$ and $Z$ `where` $Z = ((Z, Z)$`|Int`$, (Z, Z)$`|Int)|Int` produce three different types internally, although it would be reasonable to share them. Techniques of optimal sharing for recursive terms [Mau99, Mau00, Con00] would make it possible to identify $X$ and $Z$ (that differ only by unfolding), but not $X$ and $Y$, that can be made equal by using a commutativity property in the internal algebra. It would be interesting to adapt these works to take into account these kind of properties (associativity, commutativity, idempotence).

We can easily describe the predicate that detects structural equivalence between two types in the internal algebra in a coinductive form; we use the fact that two sets $A$ and $B$ are equal if and only if:

$$(\forall x \in A.\ \exists y \in B.\ x = y) \wedge (\forall y \in B.\ \exists x \in A.\ x = y)$$

The negation of this structural equivalence predicate is, as for subtyping, an inductive property, and we can therefore use the algorithms of Chapter 7 to implement if efficiently. This sharing technique should be formalized and its concrete impact on the typechecker's performance assessed.

**Combinator algebra: iterators, XML paths, pattern matches** ℂDuce has predefined iterators, to work on sequences (Section 10.3.5) or XML trees. The typing of these operators is much more precise than what can be obtained with ML-like parametric polymorphism. It is disappointing not to be able to *define* these kind of operators in the language. Hosoya [Hos04] introduced an operation that at the same time generalizes pattern matching and iteration on sequences. It would be interesting to extend this approach to allow the programmer to also integrate XML paths (as in XPath[XPA]), and to specify iterators that do not necessarily preserve the order of the elements of the sequences. One way would be to introduce an algebra of combinators that makes it possible to express in a uniform framework all these constructions, then to use it to study the issues of typing and evaluation, and finally to propose agreeable syntaxes to define these combinators. The typing of combinators would occur at the time of their application, when the type of the input expression is completely known (which makes it possible to iterate on it). Efficiency issues for the implementation could lead to developments similar to those of Chapter 8, to take into account static types.

**Heuristics and cost models for pattern matching** The formalism of Chapter 8 makes it possible to consider several strategies for compiling pattern matching. We described a sequential left-right strategy, and an iterative strategy. The current implementation of ℂDuce uses a sequential left-right strategy.

A compilation scheme that aims at producing globally sequential code for pattern matching must choose between a left-right or right-left strategy for each query. Levin and Pierce [LP04] suggest a "maximum connection factor" heuristic. It can be adapted to our formalism to suggest, for each query, either a left-right strategy or a right-left strategy. Here is how to do it. We use the Section 8.3.8 construction to compute the left-hand sub-request (in a left-right strategy), and similarly, the right-hand sub-request if a right-left strategy is used. The connection factor for a query is the maximum size of a disjoint partition of that query, that is, a partition where two atomic queries that accept a common value are necessarily in the same class (the corresponding partition is given by the transitive closure of the "accept a common value" relation). This heuristic advises to start with the side that maximizes this connection factor. It would be interesting to measure the impact of such a heuristic.

We could also seek to extend these kind of heuristics to non-iterative strategies. One way to do this, inspired by the optimization of queries in databases, is to define a cost model, that is, to associate a numerical cost with any strategy.

We could then seek to optimize this cost, either exactly (possibly by listing all possible strategies, as there are only a finite number that do not perform the same atomic sub-request several times), or using an approximation. Defining the cost of strategies is not easy: we would have to take into account the cost of a sub-request, which itself may depend on the cost of the current query that is being calculated, in case of a recursion.

## 12.2 Around ℂDuce

in parallel to my thesis, other work has been initiated around the ℂDuce language.

**Safety analysis for ℂDuce** The thesis of Marwan Burelle, directed by Véronique Benzaken and Giuseppe Castagna, focuses on static and dynamic safety analysis for the ℂDuce language. It is based on an analysis of the information flow, by modifying the semantics of the language with labels (which represent levels of security). Preliminary results [BBC03] have been published.

**A query language for ℂDuce** The thesis of Cédric Miachon, directed by Giuseppe Castagna and Véronique Benzaken, studies a query language, called CQL, built on top of the ℂDuce language. CQL uses the great expressive power ℂDuce's pattern matching. This work has led to the integration in ℂDuce of the construction `select .. from ... where ....` Another line of research, led by Véronique Benzaken and Ioana Manolescu, seeks to interface ℂDuce and CQL with XML document physical storage systems; the aim is to avoid fully loading potentially gigantic documents into central memory, and to use algorithms from the database world to efficiently represent data in mass memory, with random access, and to enable rapid evaluation of queries.

**Pattern matching combinators ℂDuce** Kim Nguyen studies in his thesis, directed by Véronique Benzaken, a combinator algebra that makes it possible to define strongly-polymorphic combinators (see above): iterators, paths, structural transformations. Preliminary results were obtained during his DEA [Ngu04] internship, which I co-supervised with Giuseppe Castagna and Véronique Benzaken.

**Channel and semantic subtyping** Castagna, De Nicola and Varacca [CNV04] define, by extending the set-theoretic approach presented in this thesis, a subtyping relation for channels, whose goal is to type $\pi$-calculus.

# Bibliography


[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993. (cité page 31)

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structures.* M.I.T. Press, 1991. (cité page 37)

[BBC03] Véronique Benzaken, Marwarn Burelle, and Giuseppe Castagna. Security analysis for xml transformations. In *Eighth Asian Computing Science Conference*, 2003. (cité pages 213, 215)

[BCF02] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: a white paper. In *Programming Languages Technologies for XML (PLAN-X)*, 2002. (cité page 5)

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: An XML-centric general-purpose language. In *ACM International Conference on Functional Programming (ICFP)*, 2003. (cité pages 5, 208, 212)

[BH97] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210, pages 63–81. Springer-Verlag, 1997. (cité page 32)

[CF04] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Second workshop on Programmable Structured Documents*, 2004. (cité page 5)

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995. (cité page 33)

[CNV04] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the $\pi$-calculs, 2004. Unpublished. (cité page 215)

[Con00] Jeffrey Considine. Efficient hash-consing of recursive types. Technical Report 2000-006, Boston University, January 2000. (cité pages 51, 213)

[Dam94] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789, pages 687–706. Springer-Verlag, 1994. (cité page 29)

[DCFGM02] Mariangiola Dezani-Ciancaglini, Alain Frisch, Elio Giovannetti, and Yoko Motohama. The relevance of semantic subtyping. In *Intersection Types and Related Systems (ITRS)*. Electronic Notes in Theoretical Computer Science, 2002. (cité page 5)

[DG84] William Dowling and Jean Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulas. *Journal of Logic Programming*, 3:267–284, 1984. (cité page 32)

[DOM] Document Object Model (DOM); http://www.w3.org/DOM/. (cité page 21)

[FC04] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop. (cité pages 6, 186)

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 137–146. IEEE Computer Society Press, 2002. (cité page 6)

[Fri01] Alain Frisch. Types récursifs, combinaisons booléennes et fonctions surchargées: application au typage de XML, 2001. Rapport de DEA (Université Paris 7). (cité pages 5, 66)

[Fri04] Alain Frisch. Regular tree language recognition with static information. In *3rd IFIP International Conference on Theoretical Computer Science (TCS)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop. (cité pages 5, 31)

[Ftct04a] Alain Frisch and the ℂDuce team. ℂDuce: Tutorial, 2004. http://www.cduce.org/tutorial.html. (cité page 212)

[Ftct04b] Alain Frisch and the ℂDuce team. ℂDuce: User's manual, 2004. http://www.cduce.org/manual.html. (cité page 212)

[GLP00] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. In *ACM SIGPLAN Notices*, volume 35(9), pages 221–231, 2000. (cité page 32)

[GLPS04] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. Xml goes native: Run-time representations for xtatic, 2004. (cité page 201)

[GP03] Vladimir Gapeyev and Benjamin Pierce. Regular object types. In *European Conference on Object-Oriented Programming*, 2003. (cité page 29)

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *Proceedings of the 32st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005. (cité pages 6, 212)

[HM02] Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*, 2002. (cité page 32)

[HM03] Haruo Hosoya and Makoto Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, 2003. (cité page 32)

[Hos01] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001. (cité pages 19, 23, 30, 67, 130, 208, 211)

[Hos03] Haruo Hosoya. Regular expression pattern matching - a simpler design, 2003. (cité page 31)

[Hos04] Haruo Hosoya. Regular expression filters for XML. 2004. (cité page 214)

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databas es (WebDB2000)*, 2000. (cité page 23)

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001. (cité page 23)

[HP02] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *Journal of Functional Programming*, volume 13(4). 2002. (cité page 23)

[HP03] Haruo Hosoya and Benjamin C. Pierce. A typed XML processing language. In *ACM Transactions on Internet Technology*, volume 3(2), pages 117–148. 2003. (cité page 23)

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000. (cité pages 23, 23)

[KPMI95] Dexter Kozen, Jens Palsberg, and Schwartzbach Michael I. Efficient recursive subtyping. In *Mathematical Structures in Computer Science*, volume 5(1), pages 113–125, 1995. (cité page 32)

[Lev03] Michael Y. Levin. Matching automata for regular patterns. In *International Conference on Functional Programming (ICFP)*, 2003. (cité page 31)

[LP04] Michael Y. Levin and Benjamin C. Pierce. Type-based optimization for regular patterns. In *First International Workshop on High Performance XML Processing*, 2004. (cité pages 31, 31, 163, 214)

[LS04a] Kenny Zhuo Ming Lu and Martin Sulzmann. An implementation of subtyping among regular expression types. In *ASIAN Symposium on Programming Languages and Systems*, 2004. (cité page 201)

[LS04b] Kenny Zhuo Ming Lu and Martin Sulzmann. XHaskell: Regular expression types for haskell, 2004. (cité pages 29, 201, 213)

[Mau99] Laurent Mauborgne. *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, École Polytechnique, 1999. (cité pages 51, 213)

[Mau00] Laurent Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4):290–311, 2000. (cité pages 51, 213)

[MS03] Erik Meijer and Wolfram Schulte. Unifying tables, objects, and documents. In *Declarative Programming in the Context of OO-Languages (DP-COOL)*, 2003. (cité page 201)

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Symposium on Principles of Database Systems*, pages 11–22, 2000. (cité page 24)

[Nam] Namespaces in XML; http://www.w3.org/TR/REC-xml-names. (cité page 182)

[Ngu04] Kim Nguyễn. Une algèbre de filtrage pour le langage CDuce, 2004. Rapport de DEA (Université Paris 7). (cité page 215)

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999. (cité page 212)

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, 2001. (cité page 212)

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998. (cité page 212)

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), 2000. (cité page 212)

[REL] RELAX NG Specification; http://www.oasis-open.org/committees/relax-ng/spec-20011203.html. (cité page 21)

[SCH] XML Schema Part 1: Structures; http://www.w3.org/TR/xmlschema-1. (cité page 21)

[SH04] Tadahiro Suda and Haruo Hosoya. A non-backtracking top-down algorithm for checking tree automata containment, 2004. Unpublished. (cité page 32)

[Van04] Stijn Vansummeren. Type inference for unique pattern matching, 2004. (cité page 184)

[VM04] Jerome Vouillon and Paul-André Melliès. Semantic types: a fresh look at the ideal model for types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 52–63. ACM Press, 2004. (cité page 30)

[Wat94] Bruce W. Watson. A taxonomy of finite automata construction algorithms. Technical Report Computing Science Note 93/43, Eindhoven University of Technology, May 1994. (cité page 51)

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998. (cité page 212)

[Xi99] Hongwei Xi. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999. (cité page 212)

[XML] Extensible Markup Language Recommendation (XML) 1.0; http://www.w3.org/TR/REC-xml. (cité pages 19, 20)

[XPA] XML Path Language (XPath); http://www.w3.org/TR/xpath. (cité page 214)