Dipartimento di Informatica e
Scienze dell'Informazione

# Shape, Intensional Semantics and Polymorphism

by

Gianna Bellè

Theses Series DISI-TH-1999-01

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

# Shape, Intensional Semantics and Polymorphism

by

Gianna Bellè

April, 1999

**Dottorato di Ricerca in Informatica**
**Dipartimento di Informatica e Scienze dell'Informazione**
**Università degli Studi di Genova**

DISI, Università degli Studi di Genova
via Dodecaneso 35
I-16146 Genova, Italy
`http://www.disi.unige.it/`

**Ph.D. Thesis in Computer Science**

Submitted by Gianna Bellè
DISI, Università degli Studi di Genova
`gbelle@disi.unige.it`

Date of submission: December 1998

Title: Shape, Intensional Semantics and Polymorphism

Advisor:
Eugenio Moggi
DISI, Università degli Studi di Genova
`moggi@disi.unige.it`

Ext. Reviewers:
C. Barry Jay
SOCS, University of Technology, Sydney
`cbj@socs.uts.edu.au`

Andrew M. Pitts
Computer Laboratory, University of Cambridge
`ap@cl.cam.ac.uk`

# Abstract

This dissertation shows a semantic approach to the design of intermediate languages and to the analysis of programs. In particular we focus on the benefits deriving from information about the shape of data, as advocated by Barry Jay.

In the first part we propose a type theory with the corresponding categorical model. It captures what can be computed at compile-time and what has to be computed at run-time. We then show that shape errors can be detected at compile-time.

In the second part we explore a shape approach to polymorphism, by proposing an extension of ML with an additional layer of functors underneath the types, called Functorial ML. We study its relationship to polytypism with particular attention to PolyP, an extension of Haskell by Patrick Jansson and Johan Jeuring, emphasizing that shape polymorphic algorithms are parametrically polymorphic.

To Cicciona,
always standing on my desk, while I was working on this thesis.

*In the shape of things to come*
*Too much poison come undone* (Placebo in "Every You Every Me")

*From the shape of your shaved head*
*I recognize your silhouette* (Ani DiFranco in "Little Plastic Castle")

# Acknowledgements

First and foremost, I wish to thank Eugenio Moggi, my advisor, for his constant and thoughtful guidance.

During these years I had the privilege of working with Barry Jay, I thank him for his teaching and for inviting me to visit the School of Computing Sciences at University of Technology, Sydney.

Pino Rosolini taught me a lot about category theory and read the preliminary draft of the thesis. I thank him and the other members of my committee, Egidio Astesiano and Giovanni Chiola, for their comments.

The external reviewers, Andrew Pitts and Barry Jay, carefully read the thesis and gave me very helpful suggestions. Thank you.

I had stimulating and interesting discussions with many people I met during my phd studies, in particular with Daniela Archieri, Lars Birkedal, Marco Danelutto, Patrik Jansson, Johan Jeuring, Susanna Pelagatti, Milan Sekanina, Val Tannen.

People at DISI provided a good environment for working, in particular phd students, some of them are good friends and I have had the pleasure to share some of my spare time with them.

I am profoundly grateful to my family, my parents Carmela and Raffaele, my sisters Dorella and Carla, and my twin brother Siro, because they have been reminding me what is important in life, especially during hard periods. And of course, to my flatmates because they made me feel at home in Genova.

I am indebted to Elena, Johan, and all my friends, because they have made my days brighter thanks to their support, wisdom, beauty, witticism, etc.

Lastly, I would like to remember Kaj for the lovely conversations we used to have, before he suddenly passed away.

# Declaration

I hereby declare that this thesis has been written by myself, that the work reported has not been submitted for any university degree before, and that any help that I have received in preparing this thesis, and all sources used, have been acknowledged in this thesis.

Gianna Bellè

# Table of Contents

**Chapter 1 Introduction** **5**

1.0.1 Shape and intensional semantics . . . . . . 7

1.0.2 Shape and polymorphism . . . . . . 8

1.1 Overview . . . . . . 9

## I Intensional Semantics 11

**Chapter 2 Toy languages and intensional semantics** **12**

2.1 The $\mathcal{NSC}$ language . . . . . . 13

2.1.1 Syntax . . . . . . 13

2.1.2 Naïve semantics of $\mathcal{NSC}$ . . . . . . 15

2.2 The Vec language . . . . . . 16

2.2.1 Syntax . . . . . . 17

2.2.2 Semantics for vectors . . . . . . 19

2.3 Semantics for collection types . . . . . . 20

2.3.1 Shapely functors and natural transformations . . . . . . 20

2.3.2 Semantics in a lextensive category . . . . . . 21

2.3.3 General semantics for collection types. . . . . . . 29

2.4 Summary . . . . . . 31

2.4.1 $\mathcal{NSC}$'s model . . . . . . 31

2.4.2 VEC's model . . . . . . . . . . 33

**Chapter 3 Intermediate language and phase-distinction** **35**

3.1 Overview of the language **S2** . . . . . . . . . . 35

3.1.1 Intermediate language . . . . . . . . . . 35

3.1.2 Categorical semantics . . . . . . . . . . 36

3.1.3 Extensional vs. intensional type theories. . . . . . . . . . 40

3.2 Types, terms and rules . . . . . . . . . . 40

3.2.1 Rules . . . . . . . . . . 41

3.3 Categorical characterization of types . . . . . . . . . . 49

3.4 Models of **S2** . . . . . . . . . . 54

3.4.1 Model based on **Fam** construction . . . . . . . . . . 60

3.4.2 Adding collection types and computational monad . . . . . . . . . . 62

**Chapter 4 Making phase-distinction explicit** **64**

4.1 From models of **S2** to models of $\mathcal{NSC}$ . . . . . . . . . . 65

4.2 Shape-error detection for $\mathcal{NSC}$ . . . . . . . . . . 66

4.2.1 Simple case . . . . . . . . . . 66

4.2.2 General case . . . . . . . . . . 70

4.3 Shape-error detection for VEC . . . . . . . . . . 74

4.4 Translations of $\mathcal{NSC}$ (and VEC) into **S2** . . . . . . . . . . 80

4.4.1 The simple translation . . . . . . . . . . 80

4.4.2 The mixed translation . . . . . . . . . . 82

4.4.3 Auxiliary notation and notational conventions . . . . . . . . . . 87

4.5 Summary . . . . . . . . . . 92

# II Polymorphism 94

**Chapter 5 Functor-based polymorphism** **95**

5.1 Functors, types and type schema . . . 99
5.1.1 Functors . . . 99
5.1.2 Syntax . . . 100
5.1.3 Examples . . . 103
5.1.4 Substitutions and unification . . . 104
5.1.5 Semantics . . . 105
5.2 Basic terms . . . 110
5.2.1 Syntax . . . 111
5.2.2 Type inference algorithm . . . 114
5.3 Constants and reductions . . . 116
5.3.1 Basic constants and $\mathrm{FML}_{basic}$ . . . 116
5.3.2 Syntactic sugar . . . 118
5.3.3 Shape-based constants and $\mathrm{FML}_{shape}$ . . . 121
5.3.4 Properties of reduction . . . 123
5.4 Extended examples . . . 124
5.4.1 One-side matching . . . 125
5.4.2 Unification . . . 127
5.5 Variations and extensions to FML . . . 135
5.5.1 Type-theoretic variations and extensions . . . 138
5.5.2 Polytypic definitions . . . 140
5.5.3 Polytypic definitions in $\mathrm{FML}_{poly}$ . . . 142
5.6 Comparison of FML with **PolyP** . . . 146
5.7 Conclusions . . . 148

**III Conclusion** **150**

**Chapter 6 Future work** **151**

**Appendix A Glossary: mathematical preliminaries** **153**

A.1 Category theory . . . . . 153

A.1.1 Strength, monads and Kleisli categories . . . . . 153

A.1.2 List object and natural number object . . . . . 155

A.1.3 Extensivity . . . . . 157

A.1.4 Some categorical constructions . . . . . 157

A.1.5 Miscellany . . . . . 159

A.2 Fibred category theory . . . . . 160

**Appendix B Strong normalization for FML** **163**

B.0.1 System $\mathrm{FF}_{shape}$ . . . . . 164

B.0.2 Semantic domains . . . . . 167

B.0.3 Semantic functions . . . . . 168

B.0.4 Interpretation . . . . . 177

B.0.5 Strong normalization proof . . . . . 178

**Bibliography** **183**

**Index** **192**

# Chapter 1

# Introduction

This dissertation proposes a semantic approach to the design of intermediate languages and to the analysis of programs. In particular it focuses on the benefits deriving from information about the shape of data.

Intermediate languages occur in the compilation process to fill the gap between high-level languages and machine code. In fact, to overcome the complexity of the translation of a high-level language to machine code, a compiler is usually decomposed into a series of simpler compilers that successively map their source languages to closely related but slightly simpler target languages. By breaking up this task, correctness of the entire compiler can be more easily established by proving the correctness of each of the simpler compilers, and optimizations during the different phases can be performed.

We are interested in the first phase of the compilation process, when the compiler extracts information from the high-level language to be used in the successive steps. Once the information in the source program is made explicit in the intermediate language, it can be used in several ways, for examples for analysis, optimization, or code generation. A common feature of these intermediate languages is an *aggressive* use of types to incorporate additional information, e.g. binding times ([NN92]), boxed/unboxed values ([PJ91]), effects ([TJ92]). In particular, among the ML community the use of types in intermediate languages has been advocated for the TIL compiler ([HM95]) and for region inference ([TT94]). In areas like parallel programming, where efficiency is a paramount issue, good intermediate languages are critical to bridge the gap between high-level languages and efficient implementations on a variety of architectures. However in this area of computing, intermediate languages have not made significant use of types, yet.

In this dissertation we turn our attention to information about shape of data, whose utilization has been advocated by Jay, also in [Jay96]. The notion of shape of data refers to the data structures into which data are organized. For example, the shape of a labelled graph is its underlying unlabelled graph and the shape of a matrix is given by the number of rows and the number of columns. It is important not to confuse types with shapes. Consider for example a vector of vectors of integers, say $[[5,7,8],[6,8,4]]$, where square brackets contain the entries, separated by a comma, of a vector. Its shape is given by the number of the entries of the outermost vector and the number of the entries of the innermost vectors, that is the pair $\langle 2,3\rangle$. The type of the shape is given by the product of natural numbers.

Notions of structure for the data are relevant and partially supported in computing paradigms like object-oriented, database and parallel programming (e.g. [AHV95, BNTW95], [CCL91]). However they have been exploited only in minimal part for analysis during the compilation process. Knowledge of shapes can be used to optimize large-scale computations, and this is particularly important for *parallel* processing. Shapes could carry size information so that memory allocation can be pre-determined. Moreover, it is commonly the case that the structure of the shape, e.g. the edges of a graph, determines the communication patterns within the data. This information is vital for optimizing the data distribution. In some fields such as data parallel computation and systolic array computation, it is common that the shape of the result is determined only by the shape of the arguments. In these cases the shape errors can be detected very early, before any numerical computation begins. Although this dissertation does not specifically address parallel issues, we expect this research will have application to skeleton-based languages (e.g. [DFH+93]) or data parallel languages such as NESL ([Ble95]).

Jay has proposed a theory for shape ([Jay93, JC94, Jay95b]) based on elementary ideas from category theory. Intuitively, a shape can be thought of as a structure with a finite number of holes into which data elements, represented by a list, can be inserted. Shapely types are those types whose values can be decomposed into shape and data. Inductive types are shapely. For example, a tree with leaves of type $A$ has shape given by the corresponding unlabelled tree, and data given by its list of leaves (in, say, left-to-right order). Array types are shapely. Arrays of dimension $k$ have shape of type $N^k$ while arrays of arbitrary dimension have shapes given by lists of numbers. Sparse arrays are typically represented by a list of position-data pairs; the list of positions is the shape. Another example of shapely types are record types. There must be a given order on field names (e.g. lexicographic order). The underlying shape of a record is then its set(list) of field names. Examples of types that cannot have this decomposition include function types and types of sets. Function types are not shapely since their shape is trivial and their holes are not finite. Consider for example the data functor $FX = X^A$, the number of holes is the cardinality of $A$ which typically is uncountable. Sets are not shapely, since the cardinality of a set depends on knowing which elements are equal, i.e. depends on the data. Formally, the characterization of a shapely type constructor $F$ uses pullback diagrams such as

$$
\begin{tikzcd}
FA \arrow[r, "data"] \arrow[d, "shape"'] \arrow[dr, phantom, "\lrcorner", very near start] & LA \arrow[d, "length"] \\
S \arrow[r, "arity"] & N
\end{tikzcd}
$$

Values of type $FA$ are uniquely represented by a shape (of type $S$) and the collection of data elements (using a list, hence the type $LA$). The pullback specifies the constraint that the length of the data list is equal to the arity (the holes) of the shape.

### 1.0.1 Shape and intensional semantics

Jay's approach characterizes shapes in term of properties of type constructors, defining the notion of shapely functor. We push that idea to the level of type itself, in the sense that we want to model a type by a structured object that distinguishes the shape from the data. In other words, we think of the information about (the type of) the shape as implicit in the type of the data, and we identify the shape inside the data type by giving a more refined semantics, compatible with Jay's approach (so that a type will no longer be identified simply with the set of its elements, as in **Set** or **Cpo**).

Since in many situations shapes can be known before any computation begins, our aim is to provide a categorical semantics for a language where shape analysis is possible. Following a semantic approach, in particular a categorical one, we can explore various directions and characterize the structure of data without the need to develop a particular language. Moreover, a refined semantics allows the validation of program transformations and suggests optimizing transformations, in the style of rewriting rules. Shape analysis at compile-time requires that semantics models not only the shape but also phase-distinction. For simplicity, in this thesis we suppose that compile-time information is constituted only by shapes. The semantics takes inspiration from techniques for modeling the distinction between compile-time and run-time, in particular using indexed categories and fibrations (see e.g. [Mog91a]). An example is given by **Fam** construction (see Appendix A) which formalizes the notion of a family of objects indexed by the elements of a set. In giving the semantics, particular attention is devoted to the characterization of data types that represent collections, such as lists and vectors. The notion of collection type, that is its semantics, is a good notion also for modelling distribution of data in distributed and parallel computer systems.

The syntactic counterpart of the categorical semantics yields a metalanguage. Besides the correspondence between category theory and type theory, the language we propose in this dissertation, borrows ideas from work on partial evaluation, in particular 2-level lambda calculi for binding time analysis (see [NN92, Dav96]) and dependent types.

The metalanguage is the starting point for the design of a two-level intermediate language which can be used as a basis for static analysis and program transformation of high-level programming languages implemented on parallel architectures. The language is envisaged to be extended to take into account concurrency and data distribution as well as complexity aspects (some preliminary results appear in the unpublished work [Bel98]). It builds on the notion of action model and its relationship to linear logic (as in [BGHP98]). Static analysis will require adapting traditional techniques used in partial evaluation, such as constant propagation and program specialization.

To corroborate our approach, we provide a semantics and the corresponding intermediate language for taking into account phase-distinction. Moreover, as a sample of shape analysis, we show that the detection of shape errors can be done at compile-time.

### 1.0.2 Shape and polymorphism

The idea of polymorphism is to write functions that have the same functionality on different types. The major benefit is the possibility to reuse code when datatypes change during the development of programs or when functions with similar functionality are needed on different datatypes (like equality functions, print functions, parsing functions, etc.). Various sorts of polymorphism are obtained by quantifying on different kinds. Kinds classify classes of types or classes of higher-order functionals on types. For example, if we quantify on the kind of all types we have the usual notion of data polymorphism. But we can consider subclasses also: the equality function in Standard ML is polymorphic only on equality types. The ability to abstract over shapes suggests the possibility of shape polymorphism, i.e. of writing programs that are independent of the shape of their arguments. This means that we can define a function, for example, that computes the sum of the integers inside a data structure, whatever, whether it is a list or a tree.

In the second part of the dissertation we propose a shape approach to polymorphism emphasizing that shape polymorphic algorithms are parametrically polymorphic. An extension of ML called Functorial ML or FML, has been developed in collaboration with Barry Jay and Eugenio Moggi. Functorial ML is so named because its type system is similar to that of Standard ML with the addition of a new class, that of functors. The class of functors represents a wide class of type constructors that can be extended to mappings from functions to functions satisfying a few simple laws. These mappings correspond to the categorical notion of the action of functors on morphisms and are known in programming as *map*. Initially for lists, but extended to other datatypes such as trees and matrices, map is generally a useful function and it is at the basis of many algorithms, as has been shown by the work of the "Dutch-Oxford" school of functional programming and algorithm design (see e.g.

[Bac88, Bir88, Mee89]).

The main contribution of the second part of the thesis is the description of a mapping algorithm that is independent of the choice of functor, i.e. supports parametric functorial polymorphism. FML provides a fine analysis of access to data via canonical isomorphisms and thus allows a uniform algorithm to retrieve the data inside the data structure and apply the function that is passed to map. The isomorphisms are based on the separation of shape from data and suggested by the semantics of the class of functors as shapely functors. So, among the various denotational models of the language considered, the most interesting is that based on Jay's idea of *shape*. Recently, this approach has been applied to the study of traversals also, (see [MBJ98]), which generalize mapping by performing side-effects when each datum is located.

FML should be considered as an intermediate language and should be compared with other intermediate languages. In particular the access to data via the canonical isomorphisms can be compared, for example, with the approach (see [Ler92, PJ91]) that distinguishes between boxed and unboxed values and the explicit coercions between them.

## 1.1 Overview

The dissertation is organized into two parts. The first part, Chapters 2–4, focuses on phase-distinction and shows that shape errors can be detected at compile-time. It is based on the work presented in [BM97]. The second part, Chapter 5, deals with polymorphism over type constructors. The chapter is a revised version of the paper *Functorial ML* ([JBM98]).

Chapter 2 presents some toy languages. $\mathcal{NSC}$ ([ST94]) has been developed in the area of distributed databases and can be seen as a core of an entire class of collection oriented parallel languages. VEC ([JS97, Sek98]) supports a vector type constructor and has been introduced for the study of shape checking, i.e. the detection of shape errors, without handling the data stored within. We see some models of these languages and a characterization of the notion of collection type in terms of shapely monad and in terms of the corresponding weaker notion of relative shapely monad. Using the weaker notion, more examples can be modelled.

Chapter 3 concentrates on phase-distinction for taking into account binding-time information. We develop a two-level intermediate language, called **S2**, and its categorical semantics.

Chapter 4 shows how suitable translations of $\mathcal{NSC}$ and VEC into **S2** make explicit (at syntactic level) phase-distinction. From a semantic view these translations correspond to interpreting $\mathcal{NSC}$ in a suitable category obtained from a model of **S2**. Moreover, relations between the various interpretations allow one to conclude that shape errors can be detected at compile-time.

Chapter 5 describes Functorial ML. We give the syntax and the semantics of functors. Then we introduce basic term structure and elementary properties relative to a set of term constants. We consider a variety of functor-polymorphic constants motivated by the desired choice of functor semantics and we analyze examples and possible extensions of FML.

Finally, Chapter 6 addresses future work.

Appendix A recalls less common categorical notions appearing in the dissertation.

Appendix B contains a proof of strong normalization for FML.

Paul Taylor's macros for diagrams have been used throughout.

## Note for the reader

Chapter 5 can be read independently from Chapters 2–4.

Familiarity with basic category theory is assumed.

We have prefered to define most of the standard notions used in the glossary (Appendix A), rather than when they are mentioned throughout the dissertation.

# Part I

# Intensional Semantics

# Chapter 2

# Toy languages and intensional semantics

This chapter describes some idealized high-level languages and outlines their intensional semantics that makes explicit some information implicit in the source program.

We have chosen a couple of typed languages that are enough simple to analyze (if necessary we have dropped features we are not interested in) and that support collection types. Collections are types that organize collections of data, such as lists, sets, trees, and vectors (see e.g. [BNTW95]). We turn our attention on collection types which are shapely, i.e. they can split into the shape and data, in the way discussed in Chapter 1. So, for example, we will not deal with sets. The languages considered in this chapter are $\mathcal{NSC}$ ([ST94]), which supports sequences or lists and VEC ([JS97, Sek98]), which supports vectors.

A refined semantics allows one to distinguish collections that could not otherwise be distinguished. For example, in a coarse framework like **Set**, vectors can be *confused* with lists. A more refined semantics allows one to take into account shapes, and so it is possible to distinguish between list and vector. We give some examples: suitable categories highlight intensional aspects and suggest a general semantics for a language where shape analysis can be done at compile-time.

We characterize from a semantic point of view the notion of collection type in terms of a monad with suitable properties. We present a categorical semantics for the toy languages parameterized over the interpretation of the collection type. Moreover, according to Moggi's monadic approach to denotational semantics [Mog91b], we consider a notion of computation $T$ (that is a type constructor $T$, or better a strong monad $T$) which takes into account errors.

The chapter is organized as follows. Section 2.1 and section 2.2 describe respectively $\mathcal{NSC}$ and VEC. Section 2.3 gives semantics for collection types and shows some examples. Section 2.4 summarizes the categorical models of $\mathcal{NSC}$ and VEC.

## 2.1 The $\mathcal{NSC}$ language

$\mathcal{NSC}$ (Nested Sequence Calculus, [ST94]), is a high-level parallel calculus for nested sequences, and it can be seen as a possible theoretical core of an entire class of collection-oriented parallel languages. This language has been developed in the database area, where a database programming language must support one or more collection types in order to represent and manipulate large amount of data. Unlike other languages, such as NESL [Ble95], $\mathcal{NSC}$ has a small set of primitive operations and no polymorphism, therefore is simple to analyze.

### 2.1.1 Syntax

$\mathcal{NSC}$ is a typed language. The syntax is parameterized w.r.t. a signature $\Sigma$ of atomic types $D$ and operations *op*. The atomic types $D$ may include types like string, real, int, etc.

Types are given by the following grammar:

- $\tau ::= 1 \mid N \mid D \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid [\tau]$

where 1 is the unit type, $N$ is the type of natural numbers, $\tau_1 \times \tau_2$ is the product type, $\tau_1 + \tau_2$ is the sum type and $[\tau]$ is the *finite sequence* type. Sequences are just lists. Since there are no function types in $\mathcal{NSC}$, arities are associated with operations, abstractions and term constructors. Arities for operations and abstractions are of the form $\sigma ::= \overline{\tau} \to \tau$, and those for term-constructors are $\overline{\sigma}, \overline{\tau} \to \tau$ (we denote with $\overline{\tau}$ a sequence of type $\tau_1, \ldots, \tau_n$, for some $n$).

The raw syntax of terms is given next, where *op* ranges over the operations given with the signature, $c$ ranges over term-constructors and $f$ over abstractions $f ::= \lambda \overline{x : \tau}.e$,

- $e ::= x \mid op(\overline{e}) \mid f(\overline{e}) \mid c(\overline{f}, \overline{e})$.

The term-constructors in $\mathcal{NSC}$ are given in Figure 2.1.

Term-constructors associated with sequences are the following: `emp` denotes the empty sequence, `sng` is the singleton sequence and `at` is the concatenation operator. Then `flat` flattens a sequence of sequences, `length` returns the length of some sequences and `get` gets the unique element of a sequence. `map` applies to each element of a sequence a common function. Given two sequences with the same length, `zip` returns the sequence of the pairing of the corresponding elements of the two sequences. Finally, given a sequence and a sequence

| $c$ | arity | informal meaning |
|---|---|---|
| err† | $\tau$ | error |
| zero | $N$ | zero |
| succ | $N \to N$ | successor |
| eq | $N, N \to 1 + 1$ | equality of natural numbers |
| for† | $(N, \tau \to \tau), \tau, N \to \tau$ | iteration |
| $*$ | $1$ | empty tuple |
| pair | $\tau_1, \tau_2 \to \tau_1 \times \tau_2$ | pairing |
| proj$_i$ | $\tau_1 \times \tau_2 \to \tau_i$ | projection |
| in$_i$ | $\tau_i \to \tau_1 + \tau_2$ | injection |
| case† | $(\tau_1 \to \tau), (\tau_2 \to \tau), \tau_1 + \tau_2 \to \tau$ | case analysis |
| emp | $[\tau]$ | empty sequence |
| sng | $\tau \to [\tau]$ | singleton sequence |
| at | $[\tau] \times [\tau] \to [\tau]$ | concatenation |
| flat | $[[\tau]] \to [\tau]$ | flattening |
| map† | $(\tau_1 \to \tau_2), [\tau_1] \to [\tau_2]$ | mapping |
| length | $[\tau] \to N$ | length |
| get† | $[\tau] \to \tau$ | get unique element |
| zip† | $[\tau_1] \times [\tau_2] \to [\tau_1 \times \tau_2]$ | zipping |
| split† | $[\tau] \times [N] \to [[\tau]]$ | splitting |

Term-constructors marked with † can raise an error

Figure 2.1: Term-constructors of $\mathcal{NSC}$

of lengths that specify the splitting pattern, split returns the sequence of sequences according to the pattern. The meaning of the operation will be clear in the following section, when we show the semantics.

Some of these operations associated to sequence types can raise errors related to the shape of data. We refer to such errors as shape errors: zip is defined on pairs of sequences having the same length, get is defined only on sequences of length 1, split requires that the length of the sequence of data corresponds to the sum of the entries of the list of sizes.

**Remark 2.1** We have taken a term-constructor for instead of the original term-constructor while : $(\tau \to \tau), (\tau \to 1{+}1), \tau \to \tau$ . We have decided to ignore the issue of non-termination, to avoid additional complications when we translate the language in the intermediate language **S2** we develop (see Chapter 4). However in partial evaluation it is acceptable to have divergence during specialization-time.

We refer the interested reader to [ST94] for the operational semantics of $\mathcal{NSC}$.

### 2.1.2 Naïve semantics of $\mathcal{NSC}$

We model types and the terms of $\mathcal{NSC}$ in **Set**.

Interpretation of types 1, $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ are respectively, terminal object, product and sum. Interpretation of type $N$ is given by the set of natural numbers. The finite sequence type $[\tau]$ is interpreted by the set of finite sequences on the interpretation of $\tau$. In general, the finite sequence type constructor is given by the list monad (see Appendix A), so semantically the operations on sequences correspond to the morphisms associated to the list monad. For example, the empty sequence is given by the natural transformation nil and the concatenation of sequences by the natural transformation @.

The denotation $[\![\_]\!] : \mathcal{NSC} \to \mathbf{Set}$ is defined on types as follows.

$$
\begin{array}{lcl}
[\![1]\!] & = & \{*\} \\
[\![\tau_0 \times \tau_1]\!] & = & [\![\tau_0]\!] \times [\![\tau_1]\!] \\
[\![\tau_0 + \tau_1]\!] & = & [\![\tau_0]\!] + [\![\tau_1]\!] \\
[\![[\tau]]\!] & = & [\![\tau]\!]^* \quad \text{where } * \text{ is the star Kleene operator} \\
[\![N]\!] & = & \mathbf{N}
\end{array}
$$

A sequence in **Set** will be denoted by $[x_0, \ldots, x_m]$, in particular, a sequence of 0 elements by $[\_]$. We use the same notation for expressions of type $\mathbf{N}$ and for their interpretation as natural numbers.

The interpretation of abstractions and operations associated to the sequence types, are partial functions and are defined as follows:

$$
\begin{array}{lcl}
[\![\mathsf{for}(f,x,n)]\!] & = & \begin{cases} x & \text{if } n=0 \\ [\![f(n,\mathsf{for}(f,x,n))]\!] & \text{if } [\![\mathsf{for}(f,x,n-1)]\!] \text{ is defined} \end{cases} \\[2ex]
[\![\mathsf{emp}]\!] & = & [\_] \\[2ex]
[\![\mathsf{sng}(x)]\!] & = & [[\![x]\!]] \\[2ex]
[\![\mathsf{at}(s_0,s_1)]\!] & = & [\![s_0]\!]@[\![s_1]\!] \\[2ex]
[\![\mathsf{flat}(s)]\!] & = & l_0@\ldots@l_{m-1} \\
 & & \quad \text{if } [\![s]\!] = [l_{i\in m}] \\[2ex]
[\![\mathsf{map}(f,s)]\!] & = & [([\![f]\!](a_i))_{i\in m}] \\
 & & \quad \text{if } [\![s]\!] = [a_{i\in m}] \text{ and for } i\in m,\ [\![f]\!](a_i) \text{ are defined} \\[2ex]
[\![\mathsf{length}(s)]\!] & = & m \\
 & & \quad \text{if } [\![s]\!] = [a_{i\in m}] \\[2ex]
[\![\mathsf{get}(s)]\!] & = & a \\
 & & \quad \text{if } [\![s]\!] = [a] \\[2ex]
[\![\mathsf{zip}(s_0,s_1)]\!] & = & [\langle a_i,b_i\rangle_{i\in n}] \\
 & & \quad \text{if } [\![s_0]\!] = [a_{i\in n}] \text{ and } [\![s_1]\!] = [b_{i\in n}] \\[2ex]
[\![\mathsf{split}(l_0,l_1)]\!] & = & [[a_0,\ldots,a_{n_0-1}],\ldots,[a_{\Sigma_{i\in m-1}n_i},\ldots,a_{(\Sigma_{i\in m}n_i)-1}]] \\
 & & \quad \text{if } [\![l_0]\!] = [a_{i\in k}], [\![l_1]\!] = [n_{i\in m}] \text{ and } k = \Sigma_{i\in m}n_i
\end{array}
$$

## 2.2 The Vec language

The second language we refer to is Vec [JS97, Sek98]. It is a simply typed lambda calculus that supports a vector type constructor $vec$ and a type $sz$ of sizes, used to represent sizes, such as lengths of vectors.

This language has been introduced for the study of shape analysis primarily in the context of arrays and array-based computations. The shape analysis is given a translation from Vec to a sub-language of sizes, called Size, which maps a term in Vec to its shape in Size. The shape of a term will typically be much simpler than the term itself as the shape translation ignored all data. Finite dimensional arrays can be represented in Vec by nesting the vector type constructor — shape analysis ensures that all entries in a vector have the same length and that nested vectors are thus, indeed, arrays.

Differently from $\mathcal{NSC}$, VEC is a fully higher-order typed language.

### 2.2.1 Syntax

In presenting the language VEC we adapt the syntax given in [JS97] to the one of $\mathcal{NSC}$ in order to stress analogies and differences.

The type system is parameterized w.r.t. a signature $\Sigma$ of atomic types $D$ and operations $op : \overline{\tau} \to \tau$, as for $\mathcal{NSC}$.

Types are given by the following grammar:

- $\tau ::= 1 \mid N \mid D \mid \tau_1 \times \tau_2 \mid [\tau]$
  $\theta ::= \tau \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2$.

In [JS97] the type of sizes is called $sz$, but we use $N$ instead, since this type plays the same role as $N$ in $\mathcal{NSC}$. It is used to represent *sizes*, such as lengths of vectors, and it is treated differently from the datum type, like the integers. The atomic types $D$, such as integers, booleans, etc., are called *datum* types — they represent the basic data of the system. $\tau$ ranges over *data* types — these include the datum types, the type $N$ of sizes, and are closed under finite products and vector constructions, $[\tau]$ (which in [JS97] is called $vec$). Function types are not allowed inside vectors, since there is the need of equality on the (shapes of) entries of vectors. $\theta$ ranges over the data types and all the functions between them. There are no sum types, so conditional is given only on sizes.

In the calculus there is a recursion operator and operations such as $\mathsf{map}$ and $\mathsf{zip}$, are implemented using it. If we do not want to deal with recursion (as done for $\mathcal{NSC}$, see remark 2.1) we can consider those operations as primitive. In this case the treatment is similar to the one of $\mathcal{NSC}$, and so we omit it. We focus instead on the differences between lists and vectors.

The raw syntax of terms are the following.

- $e ::= x \mid op \mid c \mid \lambda x.e \mid e_1 e_2$.

Note that since we have function types, $op$ and $c$ are treated as combinators, rather than term-constructors as happens in $\mathcal{NSC}$.

Combinators associated with $N$ and vector types are given in Figure 2.2: The equality $\mathsf{eq}$ combinators is defined on a subclass of data types, the so called *ground data-free types*, which range over the unit type and the type $N$ of sizes, and are closed under finite products. The grammar is the following: $\sigma ::= 1 \mid N \mid \sigma_1 \times \sigma_2$ .

| $c$ | : | types | informal meaning |
|---|---|---|---|
| ifs† | : | $N \times \theta \times \theta \to \theta$ | conditional on sizes |
| zero | : | $N$ | zero |
| succ | : | $N \to N$ | successor |
| pred† | : | $N \to N$ | predecessor |
| eq | : | $\sigma \to \sigma \to N$ | equality on ground types |
| sng | : | $\tau \to [\tau]$ | singleton |
| cons† | : | $\tau \to [\tau] \to [\tau]$ | cons |
| hd | : | $[\tau] \to \tau$ | head of the vector |
| tail† | : | $[\tau] \to [\tau]$ | tail of the vector |
| length | : | $[\tau] \to N$ | length of the vector |

Term-constructors marked with † can raise an error

Figure 2.2: Term-constructors of VEC

The operations on the vectors are the usual operations on non-empty lists co-opted for vectors. Vectors are distinguished from lists because their length is given as part of their structure. Differently from lists, elements of vector have the same shape. This is reflected in the different meaning of the operation cons, with respect to cons for lists. The concatenation of an element to a vector is possible only if the shape of the element is the same of the shape of the elements of the vector.

The restriction on vectors allows one to define multi-dimensional array as nested vectors (so a vectors is a one-dimensional array). The challenge then is to ensure that the nested vectors do correspond to arrays, in other words, that all entries in a vector have the same length.

For type inference and reduction rules we refer to [JS97].

**Remark 2.2** We briefly outline the translation from VEC to the sub-language SIZE, which produces the shapes. The SIZE type system is simpler than that of VEC, as it involves neither atomic types $D$ nor vector types.

The shape translation # of VEC types is a SIZE type defined inductively by

$$\begin{array}{rcl}
\#D &=& 1 \\
\#1 &=& 1 \\
\#N &=& N \\
\#[\tau] &=& N \times \#\tau \\
\#(\theta_1 \times \theta_2) &=& \#\theta_1 \times \#\theta_2
\end{array}$$

$$\#(\theta_1 \to \theta_2) \quad = \quad \#\theta_1 \to \#\theta_2$$

For additional details we refer to [JS97, Sek98].

### 2.2.2 Semantics for vectors

In order to capture the fact that vectors carry their shape as part of their structure, [Jay93] proposes to model them in the arrow category $\mathbf{C}^{\to}$, for a locos $\mathbf{C}$. $\mathbf{C}^{\to}$ inherits some properties of $\mathbf{C}$, in particular that of being a locos (see [Jay93], Theorem 3.1). For instance we can consider $\mathbf{Set}^{\to}$.

The construction of vectors from $\alpha : A \to I$, object in $\mathbf{Set}^{\to}$, is given by $V\alpha : VA \to I \times N$ arising from the pullback

$$\begin{tikzcd} VA \arrow[r] \arrow[d, "V\alpha"'] \arrow[dr, phantom, "\lrcorner", very near start] & LA \arrow[d, "L\alpha"] \\ I \times N \arrow[r, "copy"] & LI \end{tikzcd}$$

where $L$ is the list functor and *copy* is the copy morphism, to be thought of as taking $\langle x, n\rangle \in I \times N$ to the list $[x, x, \ldots, x]$ of length $n$. An element of $VA$ is given by a list of $A$'s that all have the same shape in $I$. The morphism $V\alpha$ maps such a list to the common shape of the elements, paired with the length of the vector. $V$ turns out to be a monad (see [Jay93]), with unit and multiplication inherited from $L$. Its multiplication $\mu$ flattens a vector of vectors to a vector, just as is done for a list of lists.

We leave out the interpretation of the others combinators, since it is similar to the one in $\mathcal{NSC}$. We focus instead on the interpretation of the operation $\mathsf{at}$ (which is interchangeable with $\mathsf{cons}$, so we consider the first one for uniformity with $\mathcal{NSC}$). We need to restrict the operation to vectors whose entries have the same shape. This corresponds to working in the slice category $\mathbf{Set}/I$, for some shape $I$. In fact, even if $V$ is not the list functor on $\mathbf{Set}^{\to}$, it generates the list functors on the slice categories of $\mathbf{Set}$. The list of $A \stackrel{\alpha}{\to} I$ is given by $VA \stackrel{V\alpha}{\to} I \times N \stackrel{\pi}{\to} I$. $VA$ satisfies the domain equation for lists and inherits the same operations, (see [Jay93]). Thus, the concatenation operation for vectors is given by the concatenation operation for the list functor in the slice categories.

## 2.3 Semantics for collection types

In previous sections, we outlined a semantics for $\mathcal{NSC}$ and VEC. Note that the sub-languages without collection types can be interpreted in a standard way using a category with finite products, possibly cartesian closed if there are function types, or extensive if there are sums.

In this section we characterize collections. We have seen the semantics for two examples of collection types, lists and vectors, but one can consider a generic collection calculus with operations common to a class of collections, as done in [Suc95]. If necessary, one can then restrict the language to a specific collection type, for example vectors, and consider the specific operations associated to the particular collection type also.

Semantics of collection types have been given in literature mainly in terms of monads. In [WT91] the notion of *collection type* is characterized in terms of *ringad*, that is a monad with additional operations and laws for formalizing the presence of constructor operations such as binary union of sets or nullary empty set. In [LT97], the authors generalize Wadler's notion of ringad, and give an equivalent formulation.

### 2.3.1 Shapely functors and natural transformations

Shape theory [Jay93, Jay95b] provides a distinctive approach to data types, giving a precise categorical account of how data is stored within data structures, or shapes, as we have seen in the case of vectors.

The notion of *shapely functor* uses two properties of the list functor like the strength and the stability.

Let $\mathbf{E}$ be a lextensive category (Def. A.10, p. 157).

**Definition 2.3 (Shapely functor)** *Let $(F, \tau) : \mathbf{E} \to \mathbf{E}$ be a strong functor (see Def. A.1, p. 153). Then it is a* shapely functor *if $F$ is a stable functor (i.e. preserves all pullbacks). Then $F1$ is the* object of $F$-shapes *and $\# = F! : FA \to F1$ is the* shape *of $FA$.*

**Definition 2.4 (Strong transformation)** *A* strong *natural transformation, $\alpha : (F, \sigma) \Rightarrow (G, \tau)$ between strong functors is a natural transformation $\alpha : F \to G$ that commutes with the strengths. That is:*

$$
\begin{tikzcd}
FA \times B \arrow[r, "\alpha_A \times id"] \arrow[d, "\sigma"'] \arrow[dr, phantom, very near start, "\lrcorner"] & GA \times B \arrow[d, "\tau"] \\
F(A \times B) \arrow[r, "\alpha_{A\times B}"] & G(A \times B)
\end{tikzcd}
$$

**Definition 2.5 (Cartesian transformation)** *A natural transformation $\alpha : F \to G$ is* cartesian *if, for every morphism $f : A \to B$, the following square is a pullback:*

$$
\begin{tikzcd}
FA \arrow[r, "\sigma_A"] \arrow[d, "Ff"'] \arrow[dr, phantom, very near start, "\lrcorner"] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "\sigma_B"] & GB
\end{tikzcd}
$$

**Definition 2.6 (Shapely transformation)** *A cartesian, strong natural transformation $\alpha : (F, \sigma) \Rightarrow (G, \tau)$ between shapely functors is a* shapely transformation*, in which case $F$ is* shapely over $G$ by $\alpha$

If $\mathbf{E}$ has list objects also, that is $\mathbf{E}$ is a locos, then we can define the list functor on $\mathbf{E}$. Since the list functor is a shapely functor ([Coc90]), we have the notion of shapely functor over lists. All inductive functors are shapely over lists ([Jay95b]).

In [Jay95b], functors shapely over lists have been taken as a model for type constructors. This notion is too strong to characterize the vector type constructors $V$ in section 2.2.2, which is shapely but not shapely over lists. So, we consider monads that are shapely.

**Definition 2.7 (Shapely monad)** *A monad $T : \mathbf{E} \to \mathbf{E}$ is a* shapely monad*, if it is a shapely functor and $\eta$ and $\mu$ are cartesian natural transformations.*

### 2.3.2 Semantics in a lextensive category

The notion of shapely functor is developed in a lextensive category. So we model types and terms of the source language by objects and arrows of a lextensive category $\mathbf{E}$, with a natural number object. Denotations of types 1, $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ are respectively, terminal object, product and sum. The denotation of type $N$ is given by the natural number object. The denotations of term constructors associated with these types are the obvious ones. We model the collection type by a *shapely monad*.

The interpretation of operations like emp and at requires an additional algebraic structure, which can be different accordingly to the specific collection. As we have seen in the case of vectors, we do not deal with empty vectors, and the concatenation of vectors is partial. So in general, we could require an additional structure given by a natural transformation $@ : L \times L \to L$ and possibly another natural transformation $nil : 1 \to L$. We require also that $L1$ has decidable equality $eq : L1 \times L1 \to 2$.

The constructors sng, at and flat are interpreted by using respectively $\eta$, @ and $\mu$. In the case of lists, emp is interpreted by using $nil$. Note that in the case of vectors, @ is partial and defined only on vectors whose entries have the same shape. Thus we need to require that the equality of shape is decidable.

If $L1 \cong N$, then length is given by the action of the functor $L$ on the terminal object morphism.

The interpretation of operations that can raise an error uses a computational monad: terms are interpreted as morphisms in the Kleisli category $\mathbf{E}_T$, where $T$ is the error monad $TA = A + 1$. The interpretation of get, zip and split is given next and exploits the fact the monad is shapely.

Let $L$ be a shapely monad, and let $B$ and $C$ such that

$$\begin{tikzcd}
B \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & LX \arrow[d, "L!"] \\
C \arrow[r] & L1
\end{tikzcd}$$

Then we can define a monomorphism from $B$ to $LX \times C$ that makes $B$ a decidable subobject of $LX \times C$, as described in the diagram given next.

$$\begin{tikzcd}
B \arrow[r] \arrow[d, tail] \arrow[dr, phantom, very near start, "\lrcorner"] & L1 \arrow[r] \arrow[d, tail] \arrow[dr, phantom, very near start, "\lrcorner"] & 1 \arrow[d, tail, "true"] \\
LX \times C \arrow[r] & L1 \times L1 \arrow[r, "eq"] & 2
\end{tikzcd}$$

The right-hand square is a pullback because $L1$ is supposed to have a decidable equality. The left-hand square is a pullback because of the pullback given earlier. The upward arrows are monomorphisms since $true : 1 \to 2$ is a monomorphism and pullback along monomorphism is a monomorphism. Moreover the big square is a pullback too by the usual pasting property of pullback squares.

So we can define $op : LX \times C \to B + 1$ such that

$$\begin{tikzcd}
B \arrow[r] \arrow[d, tail] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "in_0"] \\
LX \times C \arrow[r, "op"] & B + 1
\end{tikzcd}$$

Following this pattern we can define the interpretation of the term-constructors get, zip and split.

- $get : LX \to X + 1$

$$\begin{tikzcd}
X \arrow[r, "\eta_X"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & LX \arrow[d, "L!"] \\
1 \arrow[r, "\eta_1"'] & L1
\end{tikzcd}$$

  where the square is a pullback because $\eta$ is cartesian.

- $zip : LX_0 \times LX_1 \to L(X_0 \times X_1) + 1$

$$\begin{tikzcd}
L(X_0 \times X_1) \arrow[r, "L\pi_1"] \arrow[d, "L\pi_0"'] \arrow[dr, phantom, "\lrcorner", very near start] & LX_1 \arrow[d, "L!"] \\
LX_0 \arrow[r, "L!"'] & L1
\end{tikzcd}$$

  where the square is a pullback because $L$ preserves pullbacks.

- $split : LX \times L(L1) \to L(LX) + 1$

$$
\begin{tikzcd}
LLX \arrow[r, "\mu_X"] \arrow[d, "L^2!"'] \arrow[dr, phantom, "\lrcorner", very near start] & LX \arrow[d, "L!"] \\
LL1 \arrow[r, "\mu_1"'] & L1
\end{tikzcd}
$$

where the square is a pullback since $\mu$ is cartesian and we suppose that $L1$ is isomorphic to $N$ (like in the case $L$ is the list functor) in order to get the correct domain for *split*.

map should be the action of the functor $L$ on morphisms. But since the morphisms are defined in the Kleisli category, the application of the functor $L$ on $f : A \to TB$ yields $Lf : LA \to L(TB)$ which has not the right type. We need something that converts a data structure of computations into a computation of a data structure, by traversing the structure and performing computations as they are found. What we need is the so called *monadic map*, called active traversal also, (see e.g. [Fok94, MJ95, MBJ98]) whose type is $(A \to TB) \to LA \to T(LB)$. More simply we need a passive traversal, that is a natural transformation $traverse : L(TB) \to T(LB)$ (see [MBJ98] for further details about traversals).

Thus a further requirement for the interpretation of collection types is the existence of a traversal of the shapely monads $L$ by the monad $T$.

In the case of list functor $L_m$, defined as $L_m X = X^m$, this requirement is satisfied. A traversal is given by a family $\zeta_m : (TX)^m \to T(X^m)$ of natural transformations. In particular, the family inducing the traversal from left to right is

$$\zeta_m : u_{i:m} \quad \mapsto \quad \text{let}\, x_0 \Leftarrow u_0 \,\text{in}\, \ldots \text{let}\, x_{m-1} \Leftarrow u_{m-1} \,\text{in}\, [x_{i:m}]$$

where $[\_]$ and let $_ \Leftarrow _$in $\_$ correspond respectively to $\eta$, the inclusion of values into computation and to the extension of a function from values to computations to a function from computations to computations (see Appendix A.1.1).

In general, as shown in [MBJ98], traversals over a monad exist for functor shapely over lists. Unfortunately vectors are not functors shapely over lists, but just shapely functors.

However traversals exist for vectors too. In Example 2.9 we show how define traversal for vectors.

Examples of collection types and their interpretation are given next.

**Example 2.8** In the lextensive category **Set** of sets, we can consider the list functor $L$, which can be represented as $LX = \coprod_{n:N} X^n$, the binary tree functor $BX = \mu X'.X + (X' \times X')$ and ordered tree functor $TX = \mu X'.X + LX'$.

As we have seen in section 2.2, we move to the arrow category to characterize vectors and distinguish them from lists. Besides the arrow category of a locos, we can consider other constructions, in particular the set-indexed family. The definition and some basic properties of set-indexed families are given in Appendix A.1.4.

**Example 2.9**

Let $\mathbf{Fam}(\mathbf{C})$ be a set-indexed family. Suppose that $\mathbf{C}$ is lextensive. This assures that $\mathbf{Fam}(\mathbf{C})$ is lextensive too (see Prop. A.17,Appendix A).

Given an object $(X_i)_{i\in I}$ in $\mathbf{Fam}(\mathbf{C})$, the set $I$ can represent the shape (which is supposed to be known at compile-time) whereas the family of sets $X_i$ can represent run-time data. The data type of lists is defined by the functor

$$L((X_i)_{i\in I}) = (\prod_{i\in m} X_{f(i)})_{\langle m,f\rangle \in \coprod_{n\in N} I^n}.$$

The data type of vectors is defined by the functor

$$V((X_i)_{i\in I}) = (X_i^m)_{\langle m,i\rangle \in N\times I}$$

or equivalently

$$\begin{tikzcd}
V((X_i)_{i\in I}) \arrow[r] \arrow[d, "V\pi"'] \arrow[dr, phantom, "\lrcorner", very near start] & L((X_i)_{i\in I}) \arrow[d, "L\pi"] \\
1_{\langle m,i\rangle \in N\times I} \arrow[r, "copy"'] & L(1_{i\in I})
\end{tikzcd}$$

So the shape of an vector is given by its length and the common shape of the entries.

We see now how to define a traversal for vectors. Let $T((X_i)_{i\in I}) = [(SX_i)_{i\in I}|1_{*\in 1}]$ be a computational monad, parameterized over the computational monad $S$. (This monad will be used in Chapter 4 to show the error-detection result.)

We define the natural transformation

$$traverse_{(X_i)_{i\in I}} : V(T((X_i)_{i\in I})) \to T(V((X_i)_{i\in I})),$$

that is

$$traverse_{(X_i)_{i\in I}} : (Y_j)_{\langle m,j\rangle \in N\times(I+1)} \to Z_{k\in(N\times I)+1},$$

where $Y_{in_0(i)} = (SX_i)^m$ and $Y_{in_1(*)} = 1$, and $Z_{in_0(\langle m,i\rangle)} = S(X_i)^m$ and $Z_{in_1(*)} = 1$.

It is given by the function

$$u : N \times (I+1) \to (N \times I) + 1$$
$$\begin{cases} \langle m, in_0(i)\rangle & \mapsto & in_0(\langle m, i\rangle) \\ \langle m, in_1(*)\rangle & \mapsto & in_1(*) \end{cases}$$

and the family

$$f_{\langle m,j\rangle} : Y_j \to Z_{u(j)}$$

where $f_{\langle m,in_1(*)\rangle} : 1 \to 1$ are just the identity morphisms and $f_{\langle m,in_0(i)\rangle} : (SX_i)^m \to S(X_i)^m$ are the traversals $\zeta_m$ for the list functor $L_m$.

Relations between more refined semantics in **Fam**(**C**) and coarser semantics in **C** will be studied in Chapter 4.

The next example proposes a category for capturing distribution of data. Unfortunately the category is not lextensive. In the next section we see how we can relax the hypothesis in order to capture this example.

**Example 2.10** In parallel programming area, the coordination language SCL ([DGTY95]) uses distributed multidimensional arrays as parallel data structures. Distributed arrays are called *parallel arrays*, in contrast with the ordinary non distributed arrays, called *sequential arrays*. Both arrays can be modelled using the category **Fam**(**Dp**), where **Dp** is the following category:

**objects** families $(X_i)_{i\in I}$ where for each element $i$ of the finite index set $I$, $X_i$ is an object of **Set**.

**morphisms** $f : (X_i)_{i\in I} \to (Y_j)_{j\in J}$ is given by a relation $R \subseteq I \times J$ such that for every $j \in J$ one has $f_j : \prod_{i:iRj} X_i \to Y_j$ in **Set**.

**composition** composition is defined as follows, given $f = \langle R, (f_j)_{j\in J}\rangle : (X_i)_{i\in I} \to (Y_j)_{j\in J}$ and $g = \langle S, (g_k)_{k\in K}\rangle : (Y_j)_{j\in J} \to (Z_k)_{k\in K}$, their composition is given by $(f \circ g) =$

$\langle U, (h_k)_{k\in K}\rangle : (X_i)_{i\in I} \to (Z_k)_{k\in K}$, where $iUk$ iff it exists $j \in J$ such that $iRj$ and $jSk$, and $h_k$ is given by the following composition:

$$\Pi_{i:iUk}X_i \xrightarrow{expansion} \Pi_{j:jSk}(\Pi_{i:iRj}X_i) \xrightarrow{\Pi_{j:jSk}f_j} \Pi_{j:jSk}Y_j \xrightarrow{g_k} Z_k$$

Note that we need the *expansion* function because if $iUk$ then it could exist more $j \in J$ such that $iRj$ and $jSk$.

**cartesian** terminal object is given by $\{\_\}_{\_\in\emptyset}$; product of $\{X_i\}_{i\in I}$ and $\{Y_j\}_{j\in J}$ is the family $\{Z_k\}_{k\in I+J}$ such that $Z_{\langle 0,i\rangle} = X_i$ and $Z_{\langle 1,j\rangle} = Y_j$.

Since **Dp** has finite products, **Fam**(**Dp**) is extensive. However, it is not necessarily lextensive.

Given an object $((D_{i,j})_{j\in n_i})_{i\in I}$ in **Fam**(**Dp**), the set $I$ can represent the data known at compile-time, $n_i$ the finite set of positions/processors (the distribution of the data) and $D_{i,j}$ run-time data at position $j$. Parallel Arrays can be defined as

$$A_p(((D_{i,j})_{j\in n_i})_{i\in I}) = ((D_{i,j})_{\langle\_,j\rangle\in m\times n_i})_{\langle m,i\rangle\in N\times I}$$

whereas sequential arrays can be defined as

$$A_s(((D_{i,j})_{j\in n_i})_{i\in I}) = ((D^m_{i,j})_{j\in n_i})_{\langle m,i\rangle\in N\times I}$$

In both cases the shape of the array is $N \times I$. But the positions for parallel arrays become $m \times n_i$, each one filled with elements of type $D_{i,j}$, whereas the positions for sequential arrays remain $n_i$ and each one is filled with elements of type $D^m_{i,j}$, that is $m$-tuple of types $D_{i,j}$.

Parallel and sequential arrays turn out to be monads. For parallel arrays, $\eta$ and $\mu$ are defined as

$\eta : ((D_{i,j})_{j\in n_i})_{i\in I} \to ((D_{i,j})_{\langle\_,j\rangle\in m\times n_i})_{\langle m,i\rangle\in N\times I}$ is defined as:

- the function between shapes: $\lambda i : I.\langle 1, i\rangle$ from $I$ to $N \times I$
- the relation between positions: $R \subseteq n_i \times (1 \times n_1)$, $jR\langle 0, j\rangle$
- the identities

$\mu : ((D_{i,j})_{\langle\_,\_,j\rangle\in m_1\times m_2\times n_i})_{\langle m_1,m_2,i\rangle\in N\times N\times I} \to ((D_{i,j})_{\langle\_,j\rangle\in m\times n_i})_{\langle m,i\rangle\in N\times I}$ is defined as:

- the function between shapes: $\lambda m_1, m_2, i : N \times N \times I.\langle m_1 * m_2, i\rangle$ from $N \times N \times I$ to $N \times I$

- the relation between positions: $R \subseteq m_1, m_2, n_i \times (m_1 * m_2 \times n_1)$, $\langle k_1, k_2, j\rangle R \langle k, j\rangle$, for $k = k_1 * m_2 + k_2$
- the identities

For sequential arrays, $\eta$ and $\mu$ are defined as

$\eta : ((D_{i,j})_{j\in n_i})_{i\in I} \to ((D^m_{i,j})_{j\in n_i})_{\langle m,i\rangle \in N\times I}$ is defined as:

- the function between shapes: $\lambda i : I.\langle 1, i\rangle$ from $I$ to $N \times I$
- the relation between positions: $R \subseteq n_i \times n_1$, is the diagonal
- the isomorphisms from $D_{i,j}$ to $D^1_{i,j}$

$\mu : (((D^{m_2}_{i,j})^{m_1})_{j\in n_i})_{\langle m_1,m_2,i\rangle \in N\times N\times I} \to ((D^m_{i,j})_{j\in n_i})_{\langle m,i\rangle \in N\times I}$ is defined as:

- the function between shapes: $\lambda m_1, m_2, i : N \times N \times I.\langle m_1 * m_2, i\rangle$ from $N \times N \times I$ to $N \times I$
- the relation between positions: $R \subseteq n_i \times n_1$, the diagonal
- the isomorphisms from $(D^{m_2}_{i,j})^{m_1}$ to $D^{m_1 * m_2}_{i,j}$

It is interesting to see that we can define a *forgetful* functor that maps models in $\mathbf{Fam}(\mathbf{Dp})$ to models in $\mathbf{Fam}(\mathbf{Set})$ identifying parallel and sequential arrays. Let $U : \mathbf{Dp} \to \mathbf{Set}$ be the global element functor defined as $U(-) = \mathbf{Dp}(1, -)$ and let $\mathbf{Fam}(U) : \mathbf{Fam}(\mathbf{Set}) \to \mathbf{Fam}(\mathbf{Dp})$, defined as in the following diagram.

$$\begin{tikzcd}
(X_i)_{i\in I} & \in \mathbf{Fam}(\mathbf{Set}) & (\prod_{j:n_i} D_{i,j})_{i\in I} \\
((D_{i,j})_{j\in n_i})_{i\in I} \in & \mathbf{Fam}(\mathbf{Dp}) \arrow[u, "\mathbf{Fam}(U)"] & ((D_{i,j})_{j\in n_i})_{i\in I} \arrow[u, mapsto]
\end{tikzcd}$$

It is straightforward to see that $U((D_{i,j})_{j\in n_i}) \cong \prod_{j:n_i} D_{i,j}$.

$\mathbf{Fam}(U)$ preserves finite products and sums because of the following propositions. This means that the denotation of the languages in the various categories is preserved.

**Proposition 2.11 ([Mac71])** *For any category $\mathbf{C}$ with small hom-sets, each hom-functor $\mathbf{C}(A, -) : \mathbf{C} \to \mathbf{Set}$ preserves all limits.*

**Proposition 2.12** *Suppose that* $\mathbf{C}_1$ *and* $\mathbf{C}_2$ *have finite products and* $F : \mathbf{C}_1 \to \mathbf{C}_2$ *preserves them. Then* $\mathbf{Fam}(F) : \mathbf{Fam}(\mathbf{C}_1) \to \mathbf{Fam}(\mathbf{C}_2)$ *preserves finite products and coproducts.*

### 2.3.3 General semantics for collection types.

Example 2.10 shows that the requirement the category for modelling the toy languages is lextensive could be too much restrictive. In this section we propose a weaker framework and we do not require the category has pullbacks along arbitrary morphisms.

The framework is given by an extensive category $\mathbf{E}$, with finite products and with a class of display maps $\mathcal{D}$, that is

**Definition 2.13 (Class of display maps)** *Let* $\mathcal{D}$ *be a class of maps of any category* $\mathbf{B}$*. Then we call* $\mathcal{D}$ *a* class of displays*, if*

- *it is closed under pullback, that is, for each* $\varphi : X \to I \in \mathcal{D}$ *and morphism* $u : J \to I \in \mathbf{B}$*, a pullback square*

$$\begin{tikzcd} X' \arrow[r, "u'"] \arrow[d, "u^*(\varphi)"'] \arrow[dr, phantom, "\lrcorner", very near start] & X \arrow[d, "\varphi"] \\ J \arrow[r, "u"'] & I \end{tikzcd}$$

  *exists in* $\mathbf{B}$*, and for every such pullback square one has that* $u^*(\varphi) \in \mathcal{D}$

- *it is closed under composition with isomorphisms.*

Then we introduce the following definitions.

**Definition 2.14 (Relative shapely functor)** *Let* $\mathbf{E}$ *be an extensive category with finite products and let* $\mathcal{D}$ *be a class of display maps that includes product projections. Let* $F : \mathbf{E} \to \mathbf{E}$ *be a functor with a given strength* $\tau$*. Then* $F$ *is a* relative shapely functor *if it preserves pullbacks along display maps in* $\mathcal{D}$*, that is, it preserves pullbacks of the following form:*

$$
\begin{tikzcd}
B' \arrow[r, "f'"] \arrow[d, "d'"'] \arrow[dr, phantom, "\lrcorner", very near start] & A' \arrow[d, "d"] \\
B \arrow[r, "f"] & A
\end{tikzcd}
$$

*with* $d : A' \to A$ *in* $\mathcal{D}$.

Note that we require that the class of display maps $\mathcal{D}$ includes product projections (equivalent to include terminal morphisms $! : X \to 1$) in order to assure the existence of pullbacks for the interpretation of the operations that can raise errors (see section 2.3.2).

**Definition 2.15 (Relative cartesian transformation)** *Let* $\mathbf{E}$ *be an extensive category with finite products, let* $\mathcal{D}$ *be a class of display maps on* $\mathbf{E}$ *and let* $F, G : \mathbf{E} \to \mathbf{E}$ *be two relative shapely functors. A natural transformation* $\sigma : F \to G$ *is* $\mathcal{D}$-cartesian *if*

$$
\begin{tikzcd}
FA \arrow[r, "\sigma_A"] \arrow[d, "Fd"'] \arrow[dr, phantom, "\lrcorner", very near start] & GA \arrow[d, "Gd"] \\
FB \arrow[r, "\sigma_B"] & GB
\end{tikzcd}
$$

*is a pullback whenever* $d : A \to B$ *is a display map in* $\mathcal{D}$.

**Definition 2.16 (Relative shapely monad)** *Let* $\mathcal{D}$ *be a class of display maps on* $\mathbf{E}$. *A monad* $T : \mathbf{E} \to \mathbf{E}$ *is a* relative shapely monad, *if it is a relative shapely functor w.r.t.* $\mathcal{D}$, *and* $\eta$ *and* $\mu$ *are relative cartesian natural transformations w.r.t.* $\mathcal{D}$.

The interpretation follows the same pattern of subsection 2.3.2. Types and terms are modelled by objects and arrows of an extensive category $\mathbf{E}$ with finite products and with a class of display maps $\mathcal{D}$ that includes products projections. The collection type constructor is given by a $\mathcal{D}$-shapely monad, possibly with additional algebraic structure. Note that the square diagrams seen in subsection 2.3.2 for the definition of operations which can raise errors, are pullbacks since $! : X \to 1$ is in the class of display maps $\mathcal{D}$.

**Example 2.17** Consider the category $\mathbf{Fam}(\mathbf{C})$, where $\mathbf{C}$ has finite products. The **Fam**-construction freely add extensive sums. A class of display maps is given by morphisms $f : (X_i)_{i\in I} \to (Y_j)_{j\in J}$, where the family $f_i : X_i \to Y_{f(i)}$ comprises product projections. Example 2.10 is an instance of this case, with $\mathbf{C} = \mathbf{Dp}$. Note that $A_p$ and $A_s$ preserve the class of display maps, therefore they are relative shapely monads.

## 2.4 Summary

In this section we summarize what constitutes a categorical model of $\mathcal{NSC}$ and VEC. For $\mathcal{NSC}$ we consider a model parameterized over a generic collection type, whereas for VEC we restrict to consider only the vector type.

### 2.4.1 $\mathcal{NSC}$'s model

Let $\mathbf{C}$ be a lextensive category with natural number object $NNO$. Let $L$ be a shapely monads such that $L1$ has a decidable equality $eq : L1 \times L1 \to 2$, with additional structure given by a natural transformation $@ : L \times L \to L$ and a natural transformation $nil : 1 \to L$.

Let $T$ be a computational monad (i.e. a strong monad) with a notion of error $err : TA$, for every object $A$ in $\mathbf{C}$.

Moreover let $traverse_A : L(TA) \to T(LA)$ be a natural transformation that converts a data structure of computations into a computation of the data structure.

Recall that $It$, $zero$, and $succ$ are the morphisms defined in Appendix A.1.2, and that $get$, $zip$, and $split$ are the morphisms defined in Section 2.3.2.

To define the interpretation we can use a metalanguage similar to the one in [Mog91b] since the category has finite products and a strong monad. In this metalanguage we have two terms $[\_]_T$ and let $_ \Leftarrow _$in $\_$ corresponding respectively to $\eta$, the inclusion of values into computation and to the extension of a function from values to computations to a function from computations to computations, characteristic of the Kleisli triples, alternative description for monads.

Types of $\mathcal{NSC}$ are interpreted as objects of $\mathbf{C}$ in the following way:

| Types | Objects in $\mathbf{C}$ |
|---|---|
| $1$ | $1$ |
| $N$ | $NNO$ or $\mathbf{N}$ |
| $\tau_1 \times \tau_2$ | $A_1 \times A_2$ |
| $\tau_1 + \tau_2$ | $A_1 + A_2$ |
| $[\tau]$ | $LA$ |

Terms (and abstractions) are interpreted as morphisms in the Kleisli category $\mathbf{C}_T$,

| $e : \tau$ of $\mathcal{NSC}$ | Morphisms $e^\star$ in $\mathbf{C}$ |
|---|---|
| $x$ | $[x]_T$ |
| $op(\overline{e})$ | $\mathrm{let}\ \overline{x}{\Leftarrow}\overline{e}^\star\ \mathrm{in}\ op^\star(\overline{x})$ |
| $f(\overline{e})$ | $\mathrm{let}\ \overline{x}{\Leftarrow}\overline{e}^\star\ \mathrm{in}\ f^\star(\overline{x})$ |
| $c(\overline{f}, \overline{e})$ | $\mathrm{let}\ \overline{x}{\Leftarrow}\overline{e}^\star\ \mathrm{in}\ c^\star(\overline{f}^\star, \overline{x})$ |
| $\lambda\overline{x} : \overline{\tau}.e$ | $\lambda\overline{x} : \overline{\tau}^\star.e^\star$ |

in particular, combinators are interpreted as follows:

| $c(\overline{f})$ | arity and definition of $c^\star(\overline{f}, \overline{x})$ |
|---|---|
| err | $TA$ |
| | $\mathsf{err}^\star = err$ |
| zero | $T\mathbf{N}$ |
| | $\mathsf{zero}^\star = [zero]_T$ |
| succ | $T\mathbf{N} \to T\mathbf{N}$ |
| | $\mathsf{succ}^\star(x) = [succ(x)]_T$ |
| eq | $\mathbf{N} \times \mathbf{N} \to T(1+1)$ |
| | $\mathsf{eq}^\star(n_1, n_2) = [eq(n_1, n_2)]_T$ |
| for | $(\mathbf{N}, A \to TA), A\mathbf{N} \to TA$ |
| | $\mathsf{for}^\star(f, x, n) = It([x]_T, \mathrm{let}\, y \Leftarrow \mathsf{for}^\star(f, x, n) \,\mathrm{in}\, f(n, y), n)$ |
| case | $(A_1 \to TA), (A_2 \to TA), A_1 + A_2 \to TA$ |
| | $\mathsf{case}^\star(f_0, f_1, in_i(x)) = f_i(x) \qquad (i = 0, 1)$ |
| emp | $TLA$ |
| | $\mathsf{emp}^\star = [nil_A]_T$ |
| sng | $A \to TLA$ |
| | $\mathsf{sng}^\star(x) = [x]_T$ |
| at | $L(A) \times L(A) \to TL(A)$ |
| | $\mathsf{at}^\star(l_1, l_2) = [@(l_1, l_2)]_T$ |
| flat | $LLA \to TL(A)$ |
| | $\mathsf{flat}^\star(l) = [\mathrm{let}_L\, x \Leftarrow l \,\mathrm{in}\, x]_T$ |
| map | $(A_1 \to TA_2), LA_1 \to T(LA_2)$ |
| | $\mathsf{map}^\star(f, l) = traverse_{A_2^\star}(\mathrm{let}_L\, x \Leftarrow l \,\mathrm{in}\, [f(x)]_L)$ |
| length | $LA \to T\mathbf{N}$ |
| | $\mathsf{length}^\star(l) = [\mathrm{let}_L\, x \Leftarrow l \,\mathrm{in}\, [*]_L]_T$ |
| get | $LA \to TA$ |
| | $\mathsf{get}^\star(l) = get(l)$ |
| zip | $LA_1, LA_2 \to T(L(A_1 \times A_2))$ |
| | $\mathsf{zip}^\star(l_1, l_2) = zip(l_1, l_2)$ |
| split | $LA, L\mathbf{N} \to T(L(LA))$ |
| | $\mathsf{split}^\star(l_1, l_2) = split(l_1, l_2)$ |

### 2.4.2 VEC's model

Let $\mathbf{C}$ be a cartesian closed category. Consider $\mathbf{Fam}(\mathbf{C})$, which is cartesian closed, extensive and with a natural number object because of Proposition A.17 in Appendix A.

Let $V : \mathbf{Fam}(\mathbf{C}) \to \mathbf{Fam}(\mathbf{C})$ be the shapely monad $V((X_i)_{i \in I}) = (X_i^m)_{\langle m, i \rangle \in N \times I}$.

Let $T((X_i)_{i\in I}) = [(SX_i)_{i\in I}|1_{*\in 1}]$ be a computational monad parameterized over a computational monad $S$, which has a notion of error given by the morphism $err = \langle in_1(*), id\rangle : 1 \to T((X_i)_{i\in I})$ in $\mathbf{Fam}(\mathbf{C})$

Moreover, let $traverse_A : V(TA) \to T(VA)$, with $A \in \mathbf{Fam}(\mathbf{C})$ be a traversal of the functor $V$ by the monad $T$, as seen in Example 2.9.

Since the category is lextensive, we can suppose the metalanguage has a constant $case : (A_0 \to A) \to (A_1 \to A) \to (A_0 + A_1) \to A$ to mimic the natural isomorphism.

Let $\# : \mathbf{Fam}(\mathbf{C}) \to \mathbf{Set}$ defined as $\#((X_i)_{i\in I}) = I$, be the functor that returns the indices of the families, that is, the shapes.

Types of VEC are interpreted as objects of $\mathbf{Fam}(\mathbf{C})$ in the following way:

| Types | Objects in $\mathbf{Fam}(\mathbf{C})$ |
|---|---|
| $1$ | $1$ |
| $N$ | $NNO$ or $\mathbf{N}$ |
| $\tau_1 \times \tau_2$ | $A_1 \times A_2$ |
| $\tau_1 \to \tau_2$ | $A_1 \to A_2$ |
| $[\tau]$ | $VA$ |

Terms are interpreted as morphisms in the Kleisli category $\mathbf{Fam}(\mathbf{C})_T$.

| $e : \tau$ of $\mathcal{NSC}$ | Morphisms $e^\star$ in $\mathbf{Fam}(\mathbf{C})$ |
|---|---|
| $x$ | $[x]_T$ |
| $op$ | $op^\star$ |
| $c$ | $c^\star$ |
| $\lambda\overline{x} : \overline{\tau}.e$ | $\lambda\overline{x} : \overline{\tau}^\star.e^\star$ |
| $e_1 e_2$ | $\mathrm{let}\ \overline{x} \Leftarrow \overline{e}_2^\star\ \mathrm{in}\ e_1^\star(x)$ |

In particular we see the interpretation of the concatenation combinator at. The interpretation of the other combinators is similar to the one for $\mathcal{NSC}$. In the interpretation of at, the subexpression $\pi_1 \circ \#(v_1)$ locates the shape of the entries of the vector $v_1$; @ is just the concatenation for lists.

| $c(\overline{f})$ | arity and definition of $c^\star(\overline{f}, \overline{x})$ |
|---|---|
| at | $VA \times VA \to T(VA)$ |
| | $\mathsf{at}^\star(v_1, v_2) = case(\mathrm{eq}(\pi_1 \circ \#(v_1), \pi_1 \circ \#(v_2)))([@(v_1, v_2)]_T)(err)$ |

# Chapter 3

# Intermediate language and phase-distinction

This chapter is devoted to describing the typed intermediate language we have developed in order to capture phase-distinction and shape at compile-time. The language is based on intensional Martin-Löf type theory (see e.g. [ML84]). Since there is a well-established categorical semantics of type theory, the development of the language has been influenced by the categorical semantics we had in mind, in particular it takes inspiration from categorical models of phase-distinction discussed in [Mog91a].

Section 3.1 gives an overview of the intermediate language **S2**. Section 3.2 gives the rules of the calculus. and Section 3.3 shows the categorical semantics of type constructors. Finally, Section 3.4 defines models for **S2**.

## 3.1 Overview of the language S2

We start the description of the language by giving the kind of judgments involved and the pattern of interpretation in a suitable model.

### 3.1.1 Intermediate language

We propose a two level language that we call **S2** ($S$ stands for *shape* and 2 for the two levels).

Two-level languages have been used for taking into account binding-time information in applications such as partial evaluation, abstract interpretation and code generation (see e.g. [Dav96, Lau91, Mog97, NN92, NN96]). Different applications pose different demands on the exact details of the two-level structure and the corresponding inference rules. The two levels in **S2** classify what can be computed at compile-time and what has to be computed at run-time. We take as fundamental that all compile-time computations take place before any run-time computation. Formally this is captured by the impossibility of compile-time terms having run-time free variables.

**S2** uses *sorts* (using Jacob's terminology in [Jac91]) to distinguish between the two levels: sort $c$ classifies compile-time and sort $r$ classifies run-time. Sorts are similar to universes as they correspond to sub-classes of all the types closed under certain constructions. The difference from universes is that a sort may not be a type.

The judgments of the language are the following:

- $\Gamma \vdash$, means that $\Gamma$ is a well-formed context
- $\Gamma \vdash A : s$, means that $A$ is a type of sort $s$ in context $\Gamma$
- $\Gamma \vdash a : A : s$, means that $a$ is a term of type $A$ of sort $s$ in context $\Gamma$
- $\Gamma \vdash A = B : s$, means that $A$ and $B$ are definitionally equal types of sort $s$ in context $\Gamma$
- $\Gamma \vdash a = b : A : s$, means that $a$ and $b$ are definitionally equal terms of type $A$, in context $\Gamma$

An expression $A$ in $\Gamma \vdash A : s$ will be called an *$s$-type*, or simply a *type*, when the sort $s$ is not of much relevance. Similarly, an expression $a$ in $\Gamma \vdash a : A : s$, will be called an *($s$-)term of type* $A$. When the sort $s$ is clear we will omit it in the derivation of a term, i.e. $\Gamma \vdash a : A$.

One of the novelties of **S2** is the use of dependent types and of a few inductive types at compile-time, borrowed from type theory (e.g. see [NPS90, CPM89]). Dependent types allow more expressiveness, in particular we can express the length of a list and thus specify properties such as the reverse function on lists should return a list of the same length as its argument. Inductive types allow one to deal with the types commonly found in programming, such as natural numbers, finite cardinals and sequences.

### 3.1.2 Categorical semantics

Denotational semantics of **S2** has to deal with dependent types and sorts. The categorical description of type dependence is fairly complicated. This is reflected by the several different, though related, categorical structures for interpreting the basic framework of dependent types proposed in the literature. We follow the approach based on *categories with attributes*, introduced by Cartmell [Car78] and later used for example by Pitts [Pit95] and Moggi [Mog91a].

**Definition 3.1** *A* category with attributes *is specified by a quadruple* $(\mathbf{C}, \mathbf{D}, G, p)$ *made of the following data:*

- *a category* $\mathbf{C}$ *with terminal object* $1$*,*
- *a discrete* $\mathbf{C}$*-indexed category* $\mathbf{D}$*, that is* $\mathbf{D} : \mathbf{C}^{op} \to \mathbf{Set}$
- *a functor* $G$*, called context extension, from the Grothendieck construction* $\mathcal{G}\mathbf{D}$ *of* $\mathbf{D}$ *to* $\mathbf{C}$
- *a cartesian natural transformation* $p$

$$\begin{tikzcd} \mathcal{G}\mathbf{D} \arrow[r, bend left, "G"] \arrow[r, bend right, "\pi_{\mathbf{D}}"'] \arrow[r, phantom, "\Downarrow p"] & \mathbf{C} \end{tikzcd}$$

Since $\mathbf{D}$ is discrete, the cartesian condition on the natural transformation $p$ corresponds to saying that for all $f : Y \to X$ in $\mathbf{C}$ and $A \in \mathbf{D}(X)$, the following square is a pullback in $\mathbf{C}$:

$$\begin{tikzcd} Y.f^*A \arrow[r, "f.A"] \arrow[d, "p_{f^*A}"'] \arrow[dr, phantom, "\lrcorner", very near start] & X.A \arrow[d, "p_A"] \\ Y \arrow[r, "f"'] & X \end{tikzcd}$$

where we write

- $f^*A$ for $\mathbf{D}(f)(A)$,
- $X.A$ for the *context extension* $G(\langle X, A\rangle)$,
- $f.A$ for $G(\langle f, id_{f^*A}\rangle)$,
- $p_A$ for the *context projection* $p$ at $\langle X, A\rangle$.

We give now some examples of categories with attributes.

**Example 3.2** The category **Set** can be turned into a category with attributes. For $X, Y$ objects in **Set** and $u : X \to Y$ morphism in **Set** we have:

- an element of $\mathbf{D}(X)$ is a family of sets $(A_x)_{x\in X}$;
- $\mathbf{D}(u)$ is the functor that maps $(B_y)_{y\in Y} \in \mathbf{D}(Y)$ to $(B_{u(x)})_{x\in X} \in \mathbf{D}(X)$;
- $G(\langle X, (A_x)_{x\in X}\rangle) = \{(x, a) | x \in X, a \in A_x\}$;
- given $\langle u, (f_x)_{x\in X}\rangle : \langle X, (A_x)_{x\in X}\rangle \to \langle Y, (B_y)_{y\in Y}\rangle$, the action of $G$ on the morphism is $G(\langle u, (f_x)_{x\in X}\rangle) = (u(x), f_x(a))$;
- $p_A = \pi_0 : \{(x, a)|x \in X, a \in A_x\} \to X$.

**Example 3.3** A category **C** with finite products can be turned into a category with attributes. For $X, Y, A$ objects in **C** and $f : Y \to X$ morphisms in **C** we have:

- $\mathbf{D}(X) = Obj(\mathbf{C})$;
- $\mathbf{D}(u)$ is the identity;
- $G(\langle X, A\rangle) = X \times A$;
- given $\langle u, f\rangle : \langle X, A\rangle \to \langle Y, B\rangle$, $G(\langle u, f\rangle) = u \times f$;
- $p_A = \pi_0 : X \times A \to X$.

The partition in **S2** of the class of all types into two sorts, is reflected by equipping the category **C** with two categories with attributes over the same category **C**, one for each sort, say $(\mathbf{C}, \mathbf{D}^c, G^c, p^c)$ and $(\mathbf{C}, \mathbf{D}^r, G^r, p^r)$.

The interpretation pattern is then given as follows.

| | Judgment | Interpretation |
|---|---|---|
| Context | $\Gamma \vdash$ | object $X$ in **C** |
| Type | $\Gamma \vdash A : s$ | element $A$ of $\mathbf{D}^s[X]$ |
| Term | $\Gamma \vdash a : A : s$ | section $f$ of $p^s_A$ |

Categories with attributes require canonical choices. We prefer to use display map categories (as done, for instance, in [Tay86, Tay99]), so that the presentation of the universal properties required for the interpretation of types becomes simpler.

**Definition 3.4 (Display map category)** *A* display map category *(or a* category with display maps*) is a pair* $(\mathbf{B}, \mathcal{D})$*, where* $\mathbf{B}$ *is a category with terminal object* $1$ *and* $\mathcal{D} \subseteq Arr(\mathbf{B})$ *is a class of displays.*

Some example of classes of display maps are given by:

- all maps in **Set**, or in general in a lex category;
- product projections in a category with finite products.

**Remark 3.5** In general, a class $\mathcal{D}$ of display maps on a category $\mathbf{C}$, need not be a subcategory of $\mathbf{C}$ (same objects, morphisms given by the displays), i.e. include all identities and be closed under composition. But the semantic classes of displays are often subcategories, and in practice it is convenient to make this assumption.

So from now on, we take as definition of class of display map the following.

**Definition 3.6 (Class of display maps)** *Let* $\mathcal{D}$ *be a class of maps of any category* $\mathbf{B}$*. Then we call* $\mathcal{D}$ *a* class of displays*, if it is closed under pullback and composition with isomorphisms and moreover if it is closed under:*

- *identities;*
- *composition.*

A category with attributes $(\mathbf{C}, \mathbf{D}, G, p)$ can be turned into a display maps category $(\mathbf{C}, \mathcal{D})$, taking as display maps the context projections, that is:

$$\mathcal{D} = \{p_A : X.A \to X | A \in \mathbf{D}(X), X \in \mathbf{C}\}$$

and closing it under isomorphisms, identities and composition.

In this notion of model, types are identified with their associated context projections. So there is no distinction between denotation of types and their corresponding context extensions.

We give a denotational semantics of **S2** in terms of a category $\mathbf{C}$ with terminal object, equipped with two classes of display maps over the category $\mathbf{C}$, one for each sort, say $\mathcal{D}^c$ and $\mathcal{D}^r$.

### 3.1.3 Extensional vs. intensional type theories.

In order to enable equational reasoning inside type theory one can introduce a type corresponding to equality, the *identity* type $Id_A(a, b)$, for terms $a, b$ of type $A$ (see e.g. [Hof95, Hof97]). If the type $\Gamma \vdash Id_A(a, b)$ is inhabited, we call the two terms $\Gamma \vdash a, b : A$ *propositionally equal*. Type theories deal with identity types in two different ways. In extensional type theory the propositional equality induced by the identity types is identified with definitional equality, i.e. conversion. In fact, there is a rule that allows to conclude $\Gamma \vdash a = b : A$ from $\Gamma \vdash c : Id_A(a, b)$. This rule makes type-checking undecidable. In intensional type theory, propositional equality does not imply definitional equality, the latter being confined to definitional expansion and normalization. Then type-checking can be decidable and this variant is therefore adopted by most implementations.

Extensional type theories are easier to model. They correspond to the intuitive understanding of a single prevailing equality which is extensional (for example it identifies pointwise equal functions (functional extensionality)). Thus, whereas **S2** is based on intensional type theory because of the properties of decidability of intensional type theories, the categorical models of **S2** we present are *extensional*, that is extensional type theory is correct w.r.t. this semantics. This means for instance that the denotation of finite cardinals and arrays is defined in terms of the NNO by exploiting extensionality.

## 3.2 Types, terms and rules

According to the high-level source language, **S2** is conceived as liable to variations and extensions.

We propose the following settings. The first one constitutes the core of the language, and we call it $\mathbf{S2}_{basic}$. The raw syntax of types is given informally by:

*c*-**types** $X, Y := 1 \mid \Sigma x : X_0.X_1 \mid X_0 + X_1 \mid R^+([x : X_0]Y_0, [x : X_1]Y_1, a) \mid N \mid F(n)$

*r*-**types** $A := 1 \mid \exists x : X.A \mid A_0 \times A_1 \mid R^+([x : X_0]A_0, [x : X_1]A_1, b) \mid \forall i : F(n).A$

Among *c*-types there are unit type 1, dependent sum $\Sigma x : X_0.X_1$, disjoint union $X_0 + X_1$, natural number type $N$ and finite cardinals types $F(n)$, where $n$ is a natural number, that is a term of type $N$. In the sequel when it will be clear, we will write $n$ instead of $F(n)$, confusing a natural number with the corresponding finite cardinal type. Among *r*-types there are unit type 1, dependent sum $\exists x : X.A$, cartesian product $A_0 \times A_1$, and a particular case of dependent product $\forall i : n.A$. Formally, the difference between $\exists$ and $\Sigma$ (and $\forall$ and $\Pi$ in $\mathbf{S2}_{ho}$) is that the body and result of one are *r*-types, of the other *c*-types. The range of the bound variable is in both cases a $c$-type. For every sort, we have the type $R^+([x : X_0]Z_0, [x : X_1]Z_1, a)$, where $a$ is a term of c-type $X_0 + X_1$, given by case analysis, whose type has sort $c$ or $r$ according to the sort of the types $Z_0$, $Z_1$ (see section 3.2). Note that defining the $c$-type $2 = 1 + 1$ we can derive the $r$-type disjoint union $A_0 + A_1$ as $\exists i : 2.A(i)$ where $A(i) = R^+(A_0, A_1, i)$. Moreover the cartesian product $X_0 \times X_1$ can be defined as $\Sigma x : X_0.X_1$.

If we want to model collection types we need to add other type constructors. For instance, consider as collection type the type of lists of length $n$. In extensional type theory we could define it as $\Pi i : n.A$, but in intensional type theory we need to add the type constructor $Seq(n, A)$. So the type system of $\mathbf{S2}_{Seq}$ is given by

$c$-**types** $X := 1 \mid \Sigma x : X_0.X_1 \mid X_0{+}X_1 \mid R^+([x : X_0]Y_0, [x : X_1]Y_1, b) \mid N \mid F(n) \mid Seq(n, X)$

$r$-**types** $A := 1 \mid \exists x : X.A \mid A_0 \times A_1 \mid R^+([x : X_0]A_0, [x : X_1]A_1, b) \mid \forall i : n.A$

Adding dependent product types $\Pi x : X_0.X_1$ and $\forall x : X.A$, and exponential type $A_0 \to A_1$, we obtain the higher-order setting, $\mathbf{S2}_{ho}$:

$c$-**types** $X := 1 \mid \Sigma x : X_0.X_1 \mid X_0{+}X_1 \mid R^+([x : X_0]Y_0, [x : X_1]Y_1, b) \mid \Pi x : X_0.X_1 \mid N \mid F(n)$

$r$-**types** $A := 1 \mid \exists x : X.A \mid A_0 \times A_1 \mid R^+([x : X_0]A_0, [x : X_1]A_1, b) \mid \forall x : X.A \mid A_0 \to A_1$

Similarly as in $A_0 + A_1$, we can define the $r$-type product $A_0 \times A_1$ as $\forall i : 2.A(i)$.

### 3.2.1 Rules

In this section we present the rules of **S2**. Since we are not aiming at a perfect correspondence of the semantics with the syntax, the rules we present should be seen as an approximation of the semantics. For instance, we will consider $\beta$-conversion but not $\eta$-conversion since $\eta$-rule is not computationally relevant.

In the sequel $s$ will range over sorts $c$ and $r$.

In presenting the rules we follow [NPS90, Hof95, Hof97].

**Remark 3.7** As we have already seen, **S2** uses *sorts* to distinguish between the two levels: sort $c$ classifies compile-time and sort $r$ classifies run-time. In [Jac91], Jacobs provides an abstract way to describe various type theories by specifying collection of sorts and dependences among them. We have not followed this approach since it is not possible to capture phase-distinction in that framework. Jacobs's setting can capture only dependences of type on terms, not of terms on terms, which is required by our notion of phase distinction.

**Rules for context.** Contexts are ordered lists of variable declarations. They are generated from the empty context by successive extensions.

$$\text{(empty context)}\ \frac{}{\diamond \vdash}$$

$$\text{(context extension)}\ \frac{\Gamma \vdash A : s}{\Gamma, x : A : s \vdash}\ x \text{ fresh}$$

**Rules for variables.** $$\text{(var)}\ \frac{\Gamma, x : A : s, \Delta \vdash}{\Gamma, x : A : s, \Delta \vdash x : A : s}$$

**Rules for conversions.** We omit the rules expressing that definitional equality is an equivalence relation.

Rule relating typing and definitional equality:

$$\text{(term conversion)}\ \frac{\begin{array}{c}\Gamma \vdash a : A : s \\ \Gamma \vdash A = B : s\end{array}}{\Gamma \vdash a : B : s}$$

Moreover, for every type constructor, we need rules that express the fact that the definitional equality respects type and term formers. We will refrain from writing down congruence rules. Such rules are silently understood for every type and term former we introduce.

For convenience we also introduce the following weakening and substitution rules, (which are admissible). Here $\Im$ ranges over one of the judgments $A : s$, $A = B : s$, $a : A : s$, $a = b : A : s$.

$$\text{(weakening)}\ \frac{\Gamma, \Delta \vdash \Im \quad \Gamma \vdash A : s}{\Gamma, x : A, \Delta \vdash \Im}\ x \text{ fresh}$$

$$\text{(substitution)}\ \frac{\Gamma \vdash a : A : s \quad \Gamma, x : A : s, \Delta \vdash \Im}{\Gamma, [a/x]\Delta \vdash [a/x]\Im}$$

Type and term formers are given next. They are introduced by formation, introduction, elimination and equality rules.

*c*-**types.** The unit type has a single canonical element $*$.

$$1\ \frac{\Gamma \vdash}{\Gamma \vdash 1 : c}$$

$$\text{1-intro}\ \frac{\Gamma \vdash}{\Gamma \vdash * : 1 : c}$$

$$\text{1-elim}\ \frac{\begin{array}{l}\Gamma, z : 1 : c \vdash C(z) : s\\ \Gamma \vdash a : [*/z]C : s\\ \Gamma \vdash b : 1 : c\end{array}}{\Gamma \vdash R^1(a, b) : [b/z]C : s}$$

$$\text{1-conv}\ \frac{\begin{array}{l}\Gamma, z : 1 : c \vdash C(z) : s\\ \Gamma \vdash a : [*/z]C : s\end{array}}{\Gamma \vdash R^1(a, *) = a : [*/z]C : s}$$

The $\Sigma$-type corresponds to disjoint union in set theory. If we are given a family of sets $(B_x)_{x\in A}$ we can form the set $\Sigma_{x\in A}B_x := \{(x,b)|x \in A, b \in B_x\}$ whose elements consist of an index $x$ and an element of the corresponding set $B_x$. In the ($\Sigma$-elim) rule, the variable $z$ in $C$ and the variables $x, y$ in $a$ become bound inside $R^\Sigma$. The idea behind $R^\Sigma$ is that in order to give a (possibly dependent) function out of $\Sigma x : A.B$, it is enough to specify it on canonical elements, i.e. the pairs. This rule is called *strong* elimination rule, because the type $C$ can depend on $\Sigma x : A.B$.

$$\Sigma\ \frac{\Gamma, x : A : c \vdash B : c}{\Gamma \vdash (\Sigma x : A.B) : c}$$

$$\Sigma\text{-intro}\ \frac{\begin{array}{l}\Gamma, x : A : c \vdash B(x) : c\\ \Gamma \vdash a : A : c \quad \Gamma \vdash b : [a/x]B : c\end{array}}{\Gamma \vdash \langle a, b\rangle : (\Sigma x : A.B) : c}$$

$$\Sigma\text{-elim}\ \frac{\begin{array}{l}\Gamma, z : (\Sigma x : A.B) : c \vdash C(z) : s\\ \Gamma, x : A : c, y : B : c \vdash a : [\langle x, y\rangle/z]C : s\\ \Gamma \vdash b : (\Sigma x : A.B) : c\end{array}}{\Gamma \vdash R^\Sigma([x : A, y : B]a, b) : [b/z]C : s}$$

$$\Sigma\text{-conv}\ \frac{\begin{array}{l}\Gamma, z : (\Sigma x : A.B) : c \vdash C(z) : s\\ \Gamma, x : A : c, y : B : c \vdash a : [\langle x, y\rangle/z]C : s\\ \Gamma \vdash c : A : c \quad \Gamma \vdash d : [c/x]B : c\end{array}}{\Gamma \vdash R^\Sigma([x : A, y : B]a, \langle c, d\rangle) = [c, d/x, y]a : [\langle c, d\rangle/z]C}$$

The following type corresponds to sum or coproducts, with the usual term formers $in_i$. We have two elimination rules: in addition to the usual one, there is an elimination rule over sorts (+-elim-s). The latter one is superfluous in a type theory in which $s$ is also a type, i.e. $s : s'$ for some sort $s'$.

$$+\ \frac{\Gamma \vdash A_i : c \quad (i = 0, 1)}{\Gamma \vdash A_0 + A_1 : c}$$

$$
+\text{-intro}\ \frac{\begin{array}{l}\Gamma \vdash A_i : c \quad (i = 0,1) \\ \Gamma \vdash a : A_i : c\end{array}}{\Gamma \vdash in_i(a) : A_0 + A_1 : c}
$$

$$
+\text{-elim}\ \frac{\begin{array}{l}\Gamma, z : A_0 + A_1 : c \vdash C(z) : s \\ \Gamma, x : A_i : c \vdash a_i : [in_i(x)/z]C : s \quad (i = 0,1) \\ \Gamma \vdash b : A_0 + A_1 : c\end{array}}{\Gamma \vdash R^+([x : A_0]a_0, [x : A_1]a_1, b) : [b/z]C : s}
$$

$$
+\text{-elim-}s\ \frac{\begin{array}{l}\Gamma, x : A_i : c \vdash C_i : s \quad (i = 0,1) \\ \Gamma \vdash b : A_0 + A_1 : c\end{array}}{\Gamma \vdash R^+([x : A_0]C_0, [x : A_1]C_1, b) : s}
$$

$$
+\text{-conv}\ \frac{\begin{array}{l}\Gamma, z : A_0 + A_1 : c \vdash C(z) : s \\ \Gamma, x : A_i : c \vdash a_i : [in_i(x)/z]C : s \quad (i = 0,1) \\ \Gamma \vdash b : A_i : c\end{array}}{\Gamma \vdash R^+([x : A_0]a_0, [x : A_1]a_1, in_i(b)) = [b/x]a_i : [in_i(b)/z]C : s}
$$

$$
+\text{-conv-}s\ \frac{\begin{array}{l}\Gamma, x : A_i : c \vdash C_i : s \quad (i = 0,1) \\ \Gamma \vdash b : A_i : c\end{array}}{\Gamma \vdash R^+([x : A_0]C_0, [x : A_1]C_1, in_i(b)) = [b/x]C_i : s}
$$

$\Pi$-types, or dependent product types, correspond to the set-theoretic notion of cartesian product of a family of sets $\Pi_{x \in A} B_x$ which has as elements families indexed over $x \in A$ of elements of the corresponding set $B_x$.

$$
\Pi\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A : s_1 \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}
$$

$$
\Pi\text{-intro}\ \frac{\Gamma, x : A \vdash a : B}{\Gamma \vdash (\lambda x : A.a) : (\Pi x : A.B)}
$$

$$
\Pi\text{-elim}\ \frac{\Gamma \vdash a : (\Pi x : A.B) : c \quad \Gamma \vdash b : A : c}{\Gamma \vdash ab : [b/x]B}
$$

$$
\Pi\text{-conv}\ \frac{\begin{array}{l}\Gamma, x : A : c \vdash a : B : c \\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash (\lambda x : A.a)b = [b/x]a : [b/x]B}
$$

An important type is the type of natural numbers. In order to define a function on $N$ it is enough to give it on the canonical elements 0 and $S(a)$.

$$
N\ \frac{\Gamma \vdash}{\Gamma \vdash N : c}
$$

$$N\text{-}0 \;\frac{\Gamma \vdash}{\Gamma \vdash 0 : N : c}$$

$$N\text{-}S \;\frac{\Gamma \vdash a : N : c}{\Gamma \vdash S(a) : N : c}$$

$$N\text{-elim} \;\frac{\begin{array}{l}\Gamma, n : N : c \vdash A(n) : s\\ \Gamma \vdash a_0 : [0/n]A : s\\ \Gamma, n : N : c, x : A(n) : s \vdash a_S : [Sn/n]A : s\\ \Gamma \vdash a : N : c\end{array}}{\Gamma \vdash R^N(a_0, [n : N, x : A(n)]a_S, a) : [a/n]A : s}$$

$$N\text{-conv-}0 \;\frac{\begin{array}{l}\Gamma, n : N : c \vdash A(n) : s\\ \Gamma \vdash a_0 : [0/n]A : s\\ \Gamma, n : N : c, x : A(n) : s \vdash a_S : [Sn/n]A : s\end{array}}{\Gamma \vdash R^N(a_0, [n : N, x : A(n)]a_S, 0) = a_0 : [0/n]A : s}$$

$N$-conv-S

$$\frac{\begin{array}{l}\Gamma, n : N : c \vdash A(n) : s\\ \Gamma \vdash a_0 : [0/n]A : s\\ \Gamma, n : N : c, x : A(n) : s \vdash a_S : [Sn/n]A : s\\ \Gamma \vdash a : N : c\end{array}}{\Gamma \vdash R^N(a_0, [n : N, x : A(n)]a_S, Sa) = [a, R^N(a_0, [n : N, x : A(n)]a_S, a)/n, x]a_S : [Sa/n]A : s}$$

**Finite cardinals.** We introduce also the types of finite cardinals. Every term of type $N$ is a finite cardinal. To be precise one should distinguish terms of type $N$, say $n$, from the corresponding cardinals $F(n)$, and in general terms 0, from the corresponding terms $0(n)$ of type $F(n)$, and $Sm$ of type $N$ from the corresponding $S(n,m)$ of type $F(Sn)$. In order to avoid cumbersome notation we confuse them.

The term 0 belongs to all the finite cardinals that are successors; if $a : n$, then $Sa : Sn$. The elimination rule is divided into two: the first one is for defining a function on the finite cardinal 0, the second one is for defining a function on the other finite cardinals. In this last case it is sufficient to give the behaviour on 0 and on the $Si$.

$$n \;\frac{\Gamma \vdash n : N : c}{\Gamma \vdash n : c}$$

$$S\text{-}0 \;\frac{\Gamma \vdash n : N : c}{\Gamma \vdash 0 : S\ n : c}$$

$$S\text{-}S \;\; \frac{\begin{array}{l}\Gamma \vdash n : N : c \\ \Gamma \vdash a : n : c\end{array}}{\Gamma \vdash Sa : Sn : c}$$

$$0\text{-elim} \;\; \frac{\begin{array}{l}\Gamma, i : 0 : c \vdash C(i) : s \\ \Gamma \vdash a : 0 : c\end{array}}{\Gamma \vdash R^0(a) : [a/i]C : s}$$

$$S\text{-elim} \;\; \frac{\begin{array}{l}\Gamma \vdash n : N : c \\ \Gamma, i : Sn : c \vdash C(i) : s \\ \Gamma \vdash a_0 : [0/i]C : s \\ \Gamma, i : n : c \vdash a_S : [Si/i]C : s \\ \Gamma \vdash b : Sn : c\end{array}}{\Gamma \vdash R^S(a_0, [i : n]a_S, b) : [b/i]C : s}$$

$$0\text{-conv} \;\; \frac{\begin{array}{l}\Gamma \vdash n : N : c \\ \Gamma, i : Sn : c \vdash C(i) : s \\ \Gamma \vdash a_0 : [0/i]C : s \\ \Gamma, i : n : c \vdash a_S : [Si/i]C : s\end{array}}{\Gamma \vdash R^S(a_0, [x : n]a_S, 0) = a_0 : [0/i]C : s}$$

$$S\text{-conv} \;\; \frac{\begin{array}{l}\Gamma \vdash n : N : c \\ \Gamma, i : Sn : c \vdash C(i) : s \\ \Gamma \vdash a_0 : [0/i]C : s \\ \Gamma, i : n : c \vdash a_S : [Si/i]C : s \\ \Gamma \vdash b : n : c\end{array}}{\Gamma \vdash R^S(a_0, [x : n]a_S, S\ b) = [b/x]a_S : [Sb/i]C : s}$$

In a stronger version of *intensional* **S2**, where sort $c$ and $r$ are universes (i.e. types of some bigger sort), one could have defined finite cardinals by induction on the natural numbers

$$F(0) = 0 \mid F(Sn) = 1 + F(n)$$

and derived the corresponding introduction and elimination rules. We have not used the stronger version of intensional **S2**, because its categorical semantics is more involved.

**Sequences of fixed length.** We introduce the collection of types of sequences of fixed length. This type is then used to define the $c$-type of lists. Given a natural number $n$ and a type $A$ we can construct the type $Seq(n, A)$ of sequences of length $n$ on $A$. There are two introduction rules, the first one introduces sequences of length 0, the second one introduces sequences of length $Sn$ by adding a new element of type $A$ to a sequence of length $n$. The corresponding elimination rules: the first one allows one to define functions on sequences of length 0 by giving their values on the canonical element []; the second one allows one to define functions on sequences of length $Sn$ by giving their values on the canonical elements, i.e. $[x, y]$.

$$Seq \; \frac{\begin{array}{l}\Gamma \vdash n : N : c \\ \Gamma \vdash A : c\end{array}}{\Gamma \vdash Seq(n, A) : c}$$

$$Seq_0\text{-intro} \; \frac{\Gamma \vdash A : c}{\Gamma \vdash [] : Seq(0, A) : c}$$

$$Seq_A\text{-intro} \; \frac{\begin{array}{l}\Gamma \vdash n : N : c \\ \Gamma \vdash a : A : c \\ \Gamma \vdash b : Seq(n, A) : c\end{array}}{\Gamma \vdash [a, b] : Seq(Sn, A) : c}$$

$$Seq_0\text{-elim} \; \frac{\begin{array}{l}\Gamma, z : Seq(0, A) : c \vdash C(z) : s \\ \Gamma \vdash a : [[]/z]C : s \\ \Gamma \vdash b : Seq(0, A) : c\end{array}}{\Gamma \vdash R^{Seq_0}(a, b) : [b/z]C : s}$$

$$Seq_S\text{-elim} \; \frac{\begin{array}{l}\Gamma \vdash n : N : c \\ \Gamma, z : Seq(Sn, A) : c \vdash C(z) : s \\ \Gamma, x : A : c, y : Seq(n, A) : c \vdash a : [[x, y]/z]C : s \\ \Gamma \vdash b : Seq(Sn, A) : c\end{array}}{\Gamma \vdash R^{Seq_S}([x : A, y : Seq(n, A)]a, b) : [b/z]C : s}$$

$$Seq_0\text{-conv} \; \frac{\Gamma \vdash R^{Seq_0}(a, []) : [[]/z]C}{\Gamma \vdash R^{Seq_0}(a, []) = a : [[]/z]C}$$

$$Seq_S\text{-conv} \; \frac{\Gamma \vdash R^{Seq_S}([x : A, y : Seq(n, A)]a, [c, d]) : [[c, d]/z]C}{\Gamma \vdash R^{Seq_S}([x : A, y : Seq(n, A)]a, [c, d]) = [c, d/x, y]a : [[c, d]/z]C}$$

As in the case of finite cardinals, in a stronger version of *intensional* **S2**, where sort $c$ and $r$ are universes, one could have defined arrays by induction on the natural numbers

$$Seq(0, A) = 1 \mid Seq(Sn, A) = A \times Seq(n, A)$$

and derived the corresponding introduction and elimination rules.

In the extensional type theory, we could have defined $Seq(n, A)$ as $\Pi i : F(n).A$. In this setting it is not possible, since for instance in ($Seq_0$-elim) we would need the $\eta$-rule to have $N = []$.

**$r$-types.** We do not want dependent types at run-time, so $r$-types can depend only on $c$-values. The elimination rules are given for $r$-types only, differently from what we have done in the previous paragraph for $c$-types. This is to avoid the possibility of having $c$-terms/types depending on $r$-terms and we obtain in this way the phase-distinction.

These are the rules for the unit type of sort $r$:

$$1 \quad \frac{\Gamma \vdash}{\Gamma \vdash 1 : r}$$

$$\text{1-intro} \quad \frac{\Gamma \vdash}{\Gamma \vdash * : 1 : r}$$

The following are the rule for cartesian products.

$$\times \quad \frac{\Gamma \vdash A_i : r \quad (i = 0, 1)}{\Gamma \vdash (A_0 \times A_1) : r}$$

$$\times\text{-intro} \quad \frac{\Gamma \vdash a_i : A_i : r \quad (i = 0, 1)}{\Gamma \vdash \langle a_0, a_1 \rangle : A_0 \times A_1 : r}$$

$$\times\text{-elim} \quad \frac{\Gamma \vdash a : A_0 \times A_1 : r}{\Gamma \vdash \pi_i(a) : A_i : r}$$

$$\times\text{-conv} \quad \frac{\Gamma \vdash a_i : A_i : r \quad (i = 0, 1)}{\Gamma \vdash \pi_i(\langle a_0, a_1 \rangle) = a_i : A_i : r}$$

The following are the rules for dependent sums of sort $r$. Unlike the rules for $c$-dependent sum, we have a *weak* form of elimination, that is the type $C$ in the ($\exists$-elim) does not depend on terms of type $\exists x : A.B$ and in general on $r$-terms.

$$\exists \quad \frac{\Gamma, x : A : c \vdash B : r}{\Gamma \vdash (\exists x : A.B) : r}$$

$$\exists\text{-intro} \quad \frac{\begin{array}{l} \Gamma, x : A : c \vdash B(x) : r \\ \Gamma \vdash a : A : c \\ \Gamma \vdash b : [a/x]B : r \end{array}}{\Gamma \vdash (\langle a, b \rangle) : (\exists x : A.B) : r}$$

$$\exists\text{-elim} \quad \frac{\begin{array}{l} \Gamma \vdash C : r \\ \Gamma, x : A : c, y : B : r \vdash a : C : r \\ \Gamma \vdash b : (\exists x : A.B) : r \end{array}}{\Gamma \vdash R^{\exists}([x : A, y : B]a, b) : C : r}$$

$$\exists\text{-conv}\ \frac{\begin{array}{l}\Gamma, x : A : c, y : B : r \vdash a : C : r\\ \Gamma, x : A : c \vdash B(x) : r\\ \Gamma \vdash c : A : c \quad \Gamma \vdash d : [c/y]B : r\end{array}}{\Gamma \vdash R^{\Sigma}([x : A, y : B]a, \langle c, d\rangle) = [c, d/x, y]a : C(\langle c, d\rangle)}$$

The following are the rules for $r$-dependent product.

$$\forall\ \frac{\Gamma \vdash A : c \quad \Gamma, x : A : c \vdash B : r}{\Gamma \vdash (\forall x : A.B) : r}$$

$$\forall\text{-intro}\ \frac{\Gamma, x : A \vdash a : B}{\Gamma \vdash (\lambda x : A.a) : (\forall x : A.B)}$$

$$\forall\text{-elim}\ \frac{\Gamma \vdash a : (\forall x : A.B) \quad \Gamma \vdash b : A}{\Gamma \vdash ab : [b/x]B}$$

$$\forall\text{-conv}\ \frac{\begin{array}{l}\Gamma, x : A \vdash a : B\\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash (\lambda x : A.a)b = [b/x]a : [b/x]B}$$

These are the rules for exponent

$$\to\ \frac{\Gamma \vdash A, B : r}{\Gamma \vdash (A \to B) : r}$$

$$\to\text{-intro}\ \frac{\Gamma, x : A : r \vdash a : B : r}{\Gamma \vdash (\lambda x : A.M) : (A \to B) : r}$$

$$\to\text{-elim}\ \frac{\Gamma \vdash a : (A \to B) : r \quad \Gamma \vdash b : A : r}{\Gamma \vdash ab : B : r}$$

$$\to\text{-conv}\ \frac{\Gamma \vdash (\lambda x : A.a) : (A \to B) : r \quad \Gamma \vdash b : A : r}{\Gamma \vdash (\lambda x : A.a)b = [b/x]a : B : r}$$

## 3.3 Categorical characterization of types

In this section we give the semantics of **S2**, following the correspondence between type theory and category theory (see e.g. [Pit95, Jac93, Tay99]).

**Phase-distinction.** Phase-distinction, that is the independence of $c$-types and $c$-terms from $r$-types, can be captured semantically, by the notion of independence of $\mathcal{D}^c$ from $\mathcal{D}^r$, adapted from the one for category with attributes introduced in [Mog91a].

First we need to introduce the notion of relative slice category.

**Definition 3.8 (Relative slice category)** *Let $X \in \mathbf{C}$ and $\mathcal{D}$ a class of display maps in $\mathbf{C}$, then the* relative slice category *$\mathbf{C}/_{\mathcal{D}}X$ has as objects the displays with codomain $X$ and as morphisms from $d_1 : Y_1 \to X$ to $d_2 : Y_2 \to X$ the $g : Y_1 \to Y_2$ s.t.*

$$\begin{tikzcd} Y_1 \arrow[r, "g"] \arrow[dr, "d_1"'] & Y_2 \arrow[d, "d_2"] \\ & X \end{tikzcd}$$

The following defines the notion of independence.

**Definition 3.9 (Independence)** *Given two display structures $\mathcal{D}_1$ and $\mathcal{D}_2$, over a category $\mathbf{C}$, we say that $\mathcal{D}_1$ is* independent *from $\mathcal{D}_2$ if for every $e : Y \to X \in \mathcal{D}_2$, the pullback functor*

$$e^* : \mathbf{C}/_{\mathcal{D}_1}X \to \mathbf{C}/_{\mathcal{D}_1}Y$$

*is an equivalence of categories.*

The notion of independence says that types and terms classified by $\mathcal{D}_1$ are invariant w.r.t. context extension by types classified by $\mathcal{D}_2$. At the syntactic level this corresponds to saying that $\Gamma \vdash J_c$ implies $\Gamma_c \vdash J_c$, where $\Gamma_c$ is the sequence of declarations $x : A : c$ of sort $c$ in $\Gamma$ and $J_c$ is an assertion of sort $c$ (i.e. nothing, $A : c$ or $a : A : c$).

A weaker notion in which we take into account only objects is the following.

**Definition 3.10 (Simple)** *Given two display structures $\mathcal{D}_1$ and $\mathcal{D}_2$, over a category $\mathbf{C}$, we say that $\mathcal{D}_1$ is* simple *w.r.t. $\mathcal{D}_2$ if for every $e : Y \to X \in \mathcal{D}_2$, the pullback functor*

$$e^* : \mathbf{C}/_{\mathcal{D}_1}X \to \mathbf{C}/_{\mathcal{D}_1}Y$$

*is essentially surjective on objects, i.e. for every $B \in \mathbf{C}/_{\mathcal{D}_1}Y$ there is $A \in \mathbf{C}/_{\mathcal{D}_1}X$ and an iso $e^*(A) \cong B$.*

The notion of *simple* applied to $\mathcal{D}^r$ w.r.t. itself, captures the fact that there are no run-time dependent types, that is $r$-types are invariant w.r.t. context extension by $r$-types. At the syntactic level this corresponds to saying that $\Gamma \vdash A : r$ implies $\Gamma_c \vdash A : r$, where $\Gamma_c$ is the sequence of declarations $x : A : c$ of sort $c$ in $\Gamma$.

**Type constructors.** We now give the categorical characterization of the type constructors we have in **S2**, in terms of properties over the collection of display maps.

When we closed the class of displays under composition and identities (see Definition 3.6) we freely added dependent sums and unit to the type theory (see for example [Tay99]). In fact, if we define a display map to be one which arises in the category of contexts from forgetting a single variable, so if the class is to be closed under composition, every (interpretation of) context $[\Gamma, x : X, y : Y]$ with two (extra) variables must be the same as some $[\Gamma, z : Z]$ with just one. In this case we have an isomorphism over $\Gamma$ making $Z$ the dependent sum $\Sigma x : X.Y$. The projection operations have arity $\Gamma, x : Z \vdash \pi_0(z) : X$ and $\Gamma, x : Z \vdash \pi_1(z) : [\pi_0(z)/x]Y$.

Similarly, if an identity (or more generally an isomorphism) is a display then it must be possible to omit the corresponding type, and conversely to introduce its value in a unique way. This type is therefore a singleton.

**Definition 3.11** *We say that a display map category* $(\mathbf{C}, \mathcal{D})$ *has*

**products** *if for each display map* $\phi : X \to I$ *the pullback functor* $\phi^* : \mathbf{C}/_{\mathcal{D}} I \to \mathbf{C}/_{\mathcal{D}} X$ *has a right adjoint* $\prod_\phi$ *and the Beck-Chevalley condition holds: for every pullback in* $\mathbf{C}$ *of the form*

$$\begin{tikzcd} X' \arrow[r, "v"] \arrow[d, "u^*(\phi)"'] \arrow[dr, phantom, "\lrcorner", very near start] & X \arrow[d, "\phi"] \\ J \arrow[r, "u"'] & I \end{tikzcd}$$

*the canonical natural transformation* $v^* \prod_\phi \to \prod_{u^*(\phi)} u^*$ *is an isomorphism.*

We can give an alternative and informal description of the definition above using the following bi-rule (which is not part of the calculus)

$$() \; \frac{\Gamma, x : X : s_1 \vdash \_ : Y : s_2}{\Gamma \vdash \_ : \Pi x : X.Y : s_2}$$

which stresses the presence of the adjunction. We use $\_$, here and in the sequel, to denote terms, etc., that are not relevant. The Beck-Chevalley condition guarantees that the induced structure is preserved by reindexing functors and thus that substitution rules hold in presence of $\Pi$-types also.

**Definition 3.12** *We say that a display map category* $(\mathbf{C}, \mathcal{D})$ *has:*

**strong natural number object** $N \in \mathbf{C}/_{\mathcal{D}}1$*, if, for all* $! : X \to 1$*,* $!^*N \in \mathbf{C}/_{\mathcal{D}}X$ *is a NNO in* $\mathbf{C}/X$*.*

This definition corresponds to the rule

$$\frac{\begin{array}{l}\Gamma \vdash a : X : s_2 \\ \Gamma, x : X : s_2 \vdash f(x) : X : s_2\end{array}}{\Gamma, n : N : s_1 \vdash f^n(a) : X : s_2}$$

By taking $\Sigma n : N.A(n)$ for $X$ we can obtain the rules for the type constructor $N$, described in section 3.2.

**Definition 3.13** *We say that a display map category* $(\mathbf{C}, \mathcal{D})$ *has*

**cartesian products** *if for each* $X \in \mathbf{C}$ *the relative slice category* $\mathbf{C}/_{\mathcal{D}}X$ *has cartesian products and for every* $e : Y \to X$ *in* $\mathbf{C}$*, the pullback functors* $e^*$ *preserves them.*

**exponentials** *if for each* $X \in \mathbf{C}$ *the relative slice category* $\mathbf{C}/_{\mathcal{D}}X$ *has exponentials and for every* $e : Y \to X$ *in* $\mathbf{C}$*, the pullback functors* $e^*$ *preserves them.*

*We say that an extensive, display map category* $(\mathbf{C}, \mathcal{D})$ *has*

**strong disjoint union** *if for each* $X \in \mathbf{C}$ *the slice category* $\mathbf{C}/X$ *has coproducts* $[d_0 + d_1] \in \mathcal{D}$*, of* $d_i : X_i \to X \in \mathcal{D}$*,* $i = 0, 1$*, and the reindexing functors preserve them.*

It is straightforward to see the correspondence of a display map category having cartesian products and the type theory having product types.

The first condition of the definition of strong disjoint union, corresponds to the rule:

$$\frac{\begin{array}{l}\Gamma \vdash B : s_2 \\ \Gamma, x : A_i : s_1 \vdash b_i : B : s_2\end{array}}{\Gamma, z : A_0 + A_1 : s_1 \vdash \_ : B : s_2}$$

from which we can derive rules for disjoint union in section 3.2, by taking $\Sigma z : X_0 + X_1.C(z)$ as $B$.

The second condition in the definition validates the rule for elimination over sorts.

**Definition 3.14** *Given a category* $\mathbf{C}$ *with two classes of display maps* $\mathcal{D}$ *and* $\mathcal{E}$*, where* $\mathcal{E}$ *is simple w.r.t. itself, we say that* $\mathbf{C}$ *has*

**$\exists$-quantifiers with range $\mathcal{D}$ and body $\mathcal{E}$** *If for each display map $\phi : X \to I \in \mathcal{D}$ the pullback functor $\phi^* : \mathbf{C}/_{\mathcal{E}} I \to \mathbf{C}/_{\mathcal{E}} X$ has a left adjoint $\exists_\phi$ and the Beck-Chevalley condition holds: for every pullback in $\mathbf{C}$ of the form*

$$\begin{tikzcd} X' \arrow[r, "v"] \arrow[d, "u^*(\phi)"'] \arrow[dr, phantom, "\lrcorner", very near start] & X \arrow[d, "\phi"] \\ J \arrow[r, "u"'] & I \end{tikzcd}$$

*the canonical natural transformation $\exists_{u^*(\phi)} u^* \to v^* \exists_\phi$ is an isomorphism.*

**$\forall$-quantifiers with range $\mathcal{D}$ and body $\mathcal{E}$** *If for each display map $\phi : X \to I \in \mathcal{D}$ the pullback functor $\phi^* : \mathbf{C}/_{\mathcal{E}} I \to \mathbf{C}/_{\mathcal{E}} X$ has a right adjoint $\forall_\phi$ and the Beck-Chevalley condition holds: for every pullback in $\mathbf{C}$ of the form*

$$\begin{tikzcd} X' \arrow[r, "v"] \arrow[d, "u^*(\phi)"'] \arrow[dr, phantom, "\lrcorner", very near start] & X \arrow[d, "\phi"] \\ J \arrow[r, "u"'] & I \end{tikzcd}$$

*the canonical natural transformation $v^* \forall_\phi \to \forall_{u^*(\phi)} u^*$ is an isomorphism.*

Existential and universal quantification are respectively the left and right adjoint to weakening, so we can express them using the following rules in the adjoint style:

$$(\exists)\ \frac{\Gamma, x : X : c, a : A : r \vdash_l \_ : B : r}{\Gamma, c : \exists x : X.A : r \vdash_l \_ : B : r}$$

$$(\forall)\ \frac{\Gamma, x : X : c \vdash_l \_ : A : r}{\Gamma \vdash_l \_ : \forall x : X.A : r}$$

These rules are not part of the calculus but have to be considered as a sort of meta-rules, used to stress the corrispondence between semantics and syntax.

**Remark 3.15** In the definition of $\exists$-quantifier we omit the condition on orthogonality (as for example in [Mog91a]) because the condition is satisfied when $\mathcal{E}$ is simple w.r.t. itself.

It remains to give the categorical structures for finite cardinals and sequences of fixed length.

In order to interpret finite cardinals we need the relation $<$ defined on the natural number object $N$, which defines a subobject $F$ of $N \times N$:

$$\begin{tikzcd}
1 \arrow[r, tail, "true"] & 2 \\
F \arrow[u, "!"] \arrow[r, tail, "m"] \arrow[dr, "f"] \arrow[ur, phantom, "\urcorner", very near start] & N \times N \arrow[u, "<"'] \arrow[d, "\pi_1"] \\
 & N
\end{tikzcd}$$

The map $f = \pi_1 \circ m : F \to N$, is the candidate for the interpretation of finite cardinal types.

**Definition 3.16** *We say that a display map category* $(\mathbf{C}, \mathcal{D})$ *with natural number object* $N$ *has*

**finite cardinals** *if the morphism* $f$ *defined above belongs to* $\mathbf{C}/_{\mathcal{D}}N$*.*

Since semantics is extensional, sequences $Seq(n, A)$ can be interpreted in the same way as $\Pi_{\_} : n.A$.

## 3.4 Models of S2

**Definition 3.17** *A categorical model for* $\mathbf{S2}_{basic}$ *is given by a category* $\mathbf{C}$ *with terminal object and two classes of display maps,* $\mathcal{D}^c$ *and* $\mathcal{D}^r$*, over* $\mathbf{C}$ *satisfying the following additional properties:*

1. $\mathbf{C}$ *is extensive;*

2. $\mathcal{D}^c$ *is independent from* $\mathcal{D}^r$*;*

3. *$\mathcal{D}^c$ has strong disjoint union, strong natural number objects and finite cardinals;*

4. *$\mathcal{D}^r$ has cartesian product;*

5. *there is $\exists$-quantifier with range in $\mathcal{D}^c$ and body $\mathcal{D}^r$;*

6. *there is $\forall$-quantifier with range given by finite cardinals and body $\mathcal{D}^r$.*

**Definition 3.18** *A categorical model for $\mathbf{S2}_{Seq}$ is obtained from a categorical model for $\mathbf{S2}_{basic}$ requiring the additional properties:*

1. *$\mathbf{D}^c$ has products (restricted to finite cardinals).*

**Definition 3.19** *A categorical model for $\mathbf{S2}_{ho}$ is obtained from a categorical model for $\mathbf{S2}_{basic}$ requiring the additional properties:*

1. *$\mathbf{D}^c$ has products;*

2. *there is $\forall$-quantifier with range in $\mathcal{D}^c$ and body $\mathcal{D}^r$.*

3. *$\mathbf{D}^r$ has exponentials;*

**Example 3.20** Examples of models of **S2** can be obtained from (split) fibrations, with display map structures.

Let $p : \mathbf{E} \to \mathbf{B}$ be a fibration with fibred terminal object, and let $\mathcal{D}$ be a class of display maps over $\mathbf{B}$. Moreover, for each $X \in \mathbf{B}$, let $\mathcal{E}_X$ be a class of display maps over $\mathbf{E}_X$ preserved by the reindexing functors.

Take $\mathbf{E}$ as the category of contexts and consider

- $\mathcal{D}^c = \{\bar{d}(A) : d^*A \to A | d : Y \to X \in \mathcal{D}, pA = X\}$
- $\mathcal{D}^r = \{f : B \to A | f \in \mathcal{E}_X, \text{for some } X \in \mathbf{B}\}$

The following proposition shows that $\mathcal{D}^c$ and $\mathcal{D}^r$ are classes of display maps.

**Proposition 3.21** *$\mathcal{D}^c$ and $\mathcal{D}^r$ are closed under pullbacks. Moreover $\mathcal{D}^c$ is independent from $\mathcal{D}^r$.*

**Proof.** $\mathcal{D}^c$ is closed under pullbacks since the next diagram is a pullback:

$$\begin{tikzcd}
d^*B \arrow[r, "\bar{d}(B)"] \arrow[d, "d^*f"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "f"] \\
d^*A \arrow[r, "\bar{d}(A)"'] & A
\end{tikzcd}$$

To show that $\mathcal{D}^r$ is closed under pullbacks it is sufficient to prove that the outermost square in the next diagram is a pullback. This follows from the fact that the two inner squares are pullbacks: the rightmost because of the previous item, the leftmost because $f'$ is a display map, since reindexing preserves display maps.

$$\begin{tikzcd}
\cdot \arrow[rr] \arrow[dr] \arrow[dd] & & B \arrow[dd, "f"] \\
& (pg)^*B \arrow[ur] & \\
\cdot \arrow[rr, "g"] \arrow[dr] & & A \\
& (pg)^*A \arrow[ur] \arrow[from=uu, crossing over, "f'"'] &
\end{tikzcd}$$

The independence requirement corresponds to showing that, given $e : B \to A$ in $\mathcal{D}^r$, the pullback functor $e^* : \mathbf{E}/_{\mathcal{D}^c}A \to \mathbf{E}/_{\mathcal{D}^c}B$ is an equivalence. It is sufficient to show that $e^*$ is full, faithful and essentially surjective. It is straightforward to see that the functor is surjective on objects. It is full and faithful because the elements in $\mathcal{D}^c$ are cartesian morphisms.

∎

The following lemma establishes the properties a fibration has to satisfy so that the total category is extensive.

**Lemma 3.22** *Let $p : \mathbf{E} \to \mathbf{B}$ be a fibration such that:*

- $\mathbf{B}$ *is extensive;*
- *$p$ has fibred coproducts plus left adjoints $\coprod_{in_0}$ and $\coprod_{in_1}$ to the reindexing functors along injections, satisfying Beck-Chevalley;*
- $\langle in_0^*, in_1^* \rangle : \mathbf{E}_{I+J} \to \mathbf{E}_I \times \mathbf{E}_J$, *and* $! : \mathbf{E}_0 \to 1$ *are equivalences.*

*Then* $\mathbf{E}$ *is extensive.*

**Proof.**

Firstly, $\mathbf{E}$ has coproducts defined as follows: given $X \in \mathbf{E}_I$ and $Y \in \mathbf{E}_J$ then their coproduct is given by the coproduct in the fibre $\mathbf{E}_{I+J}$ of $\coprod_{in_0} X$ and $\coprod_{in_1} Y$, i.e. $\coprod_{in_0} X +_{I+J} \coprod_{in_1} Y$, where $\coprod_u$ is the left adjoint to the reindexing functor $u^*$.

Then we note that $F : \mathbf{E}_{I_1} \times \mathbf{E}_{I_2} \to \mathbf{E}_{I_1+I_2}$, defined as $F(A_1, A_2) = \coprod_{in_1} A_1 +_{I_1+I_2} \coprod_{in_2} A_2$ and for $h_i : A_i \to B_i$ in $\mathbf{E}_{I_i}$, $F(h_1, h_2) = [in_1 \circ \coprod_{in_1} h_1, in_2 \circ \coprod_{in_2} h_2]$, is a left adjoint to $\langle in_0^*, in_1^* \rangle$. Since $F$ is a left adjoint of an equivalence functor, $F$ is an equivalence (follows by theorem [Mac71, Theorem 1, pag. 91]).

Now we need to show that the canonical functor $+ : \mathbf{E}/X_1 \times \mathbf{E}/X_2 \to \mathbf{E}/(X_1 + X_2)$ is an equivalence, that is it is full, faithful and essentially surjective (on objects).

We see first that $+$ is full. Let $f_i : A_i \to X_i$ and $g_i : B_i \to X_i$ respectively over $u_i : I_i \to K_i$ and $v_i : J_i \to K_i$, for $i = 1, 2$.

Let $h : +(f_1, f_2) \to +(g_1, g_2)$ be a morphism in $\mathbf{E}/(X_1 + X_2)$:

$$\begin{tikzcd}
A_1 + A_2 \arrow[rr, "h"] \arrow[dr] & & B_1 + B_2 \arrow[dl] \\
& X_1 + X_2 &
\end{tikzcd}$$

Now, we reindex all the object and morphisms of $\mathbf{E}$ in the fibre $I_1 + I_2 = p(A_1 + A_2)$.

Exploiting the equivalence of the functor $F$, we can find a pair of morphisms $\widehat{h_i} : A_i \to (p(h))^* B_i$, such that

$$\begin{tikzcd} A_i \arrow[rr, "\widehat{h_i}"] \arrow[dr] & & (p(h))^* B_i \arrow[dl] \\ & u^*(X_i) & \end{tikzcd}$$

and then the corresponding morphisms in $\mathbf{E}/X_i$ between $f_i$ and $g_i$.

We see next that $+$ is faithful. Let $f_i : A_i \to X_i$ and $g_i : B_i \to X_i$ respectively over $u_i : I_i \to K_i$ and $v_i : J_i \to K_i$, for $i = 1, 2$.

Let $h_i, l_i : f_i \to g_i$ be two morphisms in $\mathbf{E}/X_i$, respectively over $w_i, z_i$ in $\mathbf{B}$, such that $+(h_1, h_2) = +(l_1, l_2) : +(f_1, f_2) \to +(g_1, g_2)$.

Consider the corresponding vertical factors in $\mathbf{E}_{I_1+I_2}$ (denoted by $\widehat{-}$). Since $F$ is faithful and $+\widehat{(f_1, f_2)} = +(\widehat{f_1}, \widehat{f_2})$ for generic $f_i$, we have that $\widehat{h_i} = \widehat{l_i}$ and therefore $h_i = l_i$ (exploiting the fact that in $\mathbf{B}$, $+(z_1, z_2) = +(w_1, w_2)$ implies $z_i = w_i$).

We see finally that $+$ is surjective on objects.

Let $h : C \to X_1 + X_2$ be an object in $\mathbf{E}/X_1 + X_2$, and suppose it is over $u : I \to K_1 + K_2$ in $\mathbf{B}$. Consider the vertical factor $\widehat{h} : C \to u^*(X_1 + X_2)$ in $\mathbf{E}_I$. Since $\mathbf{B}$ is extensive, there are $I_1$ and $I_2$ such that $iso : I \cong I_1 + I_2$. Let $\widehat{h}' : C' \to iso^*(u^*(X_1 + X_2))$ be the reindexing of $\widehat{h}$ in $\mathbf{E}_{I_1+I_2}$. Since $F$ is surjective on objects $C' \cong C'_1 + C'_2$, and since $F$ is full, there are the two morphisms required.

∎

The next three propositions conclude the example, showing when we can obtain models for $\mathbf{S2}_{basic}$, $\mathbf{S2}_{Seq}$ and $\mathbf{S2}_{ho}$ respectively, from a fibration.

**Proposition 3.23** *Suppose the fibration $p : \mathbf{E} \to \mathbf{B}$ is s.t. :*

- *the base category $\mathbf{B}$ has a terminal object and is extensive;*
- *$p$ has fibred products;*
- *$p$ has fibred coproducts plus left adjoints $\coprod$ to the reindexing functors along injections satisfying Beck-Chevalley;*

- $\langle in_0^*, in_1^* \rangle : \mathbf{E}_{X+Y} \to \mathbf{E}_X \times \mathbf{E}_Y$ , *and* $! : \mathbf{E}_0 \to 1$ *are equivalences;*
- *there is a display structure* $\mathcal{D}$ *over* $\mathbf{B}$, *with unit, strong disjoint union, natural number object and finite cardinals;*
- *there are display structures* $\mathcal{E}_X$ *respectively on* $\mathbf{E}_X$ *consisting of projections;*
- $p$ *has* $\exists_d$ *and* $\forall_d$ *quantifier along display maps* $d$ *in* $\mathcal{D}$, *that is* $p$ *has left adjoints* $\exists$ *and right adjoints* $\forall$ *to the reindexing functors satisfying Beck-Chevalley along display maps in* $\mathcal{D}$.

*Then* $\mathbf{E}$ *with* $\mathcal{D}^c$ *and* $\mathcal{D}^r$ *is a model of* $\mathbf{S2}_{basic}$ *as defined in 3.17.*

**Proof.** We need to show that the properties that define a model for $\mathbf{S2}$ are satisfied.

1. $\mathbf{E}$ is extensive because of lemma 3.22.
2. $\mathcal{D}^c$ is independent from $\mathcal{D}^r$ because of proposition 3.21.
3. $\mathcal{D}^c$ has strong disjoint union, strong natural number objects and finite cardinals because $\mathcal{D}$ on $\mathbf{B}$ has the same structures.
4. $\mathcal{D}^r$ has cartesian product: straightforward.
5. There is $\exists$-quantifier with range in $\mathcal{D}^c$ and body $\mathcal{D}^r$. Let $\phi = \bar{d}(A) : d^*A \to A$ be a display map in $\mathcal{D}^c$ and let $\phi^* : \mathbf{C}/_{\mathcal{D}^r}A \to \mathbf{C}/_{\mathcal{D}^r}d^*A$ be the pullback functor. Define $\exists_\phi : \mathbf{C}/_{\mathcal{D}^r}d^*A \to \mathbf{C}/_{\mathcal{D}^r}A$ as follows $\exists_\phi(d^*A \times C \to d^*A) = A \times \exists_d C \to A$. It is a left adjoint to the pullback functor because

$$\begin{tikzcd} A \times \exists_d C \arrow[rr, "id \times f"] \arrow[dr] & & A \times B \arrow[dl] \\ & A & \end{tikzcd}$$

if and only if

$$\begin{tikzcd} d^*A \times C \arrow[rr, "id \times f"] \arrow[dr] & & d^*A \times d^*B \arrow[dl] \\ & d^*A & \end{tikzcd}$$

and this follow for the fact $\exists_d C \to B$ iff $C \to d^* B$ because the fibration has $\exists$-quantifier along display map in $\mathcal{D}$

6. there is $\forall$-quantifier with range in $\mathcal{D}^c$ and body $\mathcal{D}^r$. Similarly to previous case.

∎

**Proposition 3.24** *Suppose the fibration $p : \mathbf{E} \to \mathbf{B}$ is s.t. :*

- *satisfies the hypothese in Proposition 3.23;*
- *the display structure $\mathcal{D}$ over $\mathbf{B}$ has products.*

*Then $\mathbf{E}$ with $\mathcal{D}^c$ and $\mathcal{D}^r$ is a model of $\mathbf{S2}_{Seq}$ as defined in 3.18.*

**Proof.** Similarly to Proposition 3.23. ∎

**Proposition 3.25** *Suppose the fibration $p : \mathbf{E} \to \mathbf{B}$ is s.t. :*

- *satisfies the hypothese in Proposition 3.23;*
- *the display structure $\mathcal{D}$ over $\mathbf{B}$ has products;*
- *$p$ has fibred exponent;*

*Then $\mathbf{E}$ with $\mathcal{D}^c$ and $\mathcal{D}^r$ is a model of $\mathbf{S2}_{ho}$ as defined in 3.19.*

**Proof.** Similarly to Proposition 3.23. ∎

### 3.4.1 Model based on Fam construction

There is a simple way to construct a model of **S2** starting from any cartesian closed category **B** with small products and small sums (e.g. **Set** and **Cpo**) using the **Fam**-construction.

Let's consider the fibration $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$, with **B** any cartesian closed category with small products and small sums.

The category of contexts is thus given by $\mathbf{Fam}(\mathbf{B})$. The display maps in the base category are given by any morphisms in the base category, whereas the class of display maps in the fibers are given by projections.

The interpretation of judgments in $\mathbf{Fam}(\mathbf{B})$ becomes fairly simple to describe:

- $\Gamma \vdash$ is interpreted by an object $(A_i)_{i\in I}$ in $\mathbf{Fam}(\mathbf{B})$, namely $I \in \mathbf{Set}$ corresponds to $\Gamma_c$ and $(A_i)_{i\in I} \in \mathbf{B}^I$ corresponds to $\Gamma_r$;
- $\Gamma \vdash A : c$ is interpreted by a family $(X_i)_{i\in I}$ of sets;
- $\Gamma \vdash M : A : c$ is interpreted by a family $(x_i)_{i\in I}$ of elements, $x_i \in X_i$;
- $\Gamma \vdash B : r$ is interpreted by a family $b = (b_i)_{i\in I}$ of objects of $\mathbf{B}$;
- $\Gamma \vdash N : B : r$ is interpreted by a family $(f_i : a_i \to b_i)_{i\in I}$ of morphisms in $\mathbf{B}$.

Interpretation of $c$-types are given next, where $I \in \mathbf{Set}$ corresponds to $\Gamma_c$, $(X_i)_{i\in I}$ corresponds to $\Gamma \vdash A : c$ and $(Y_{(i,x)})_{i\in I, x\in X_i}$ corresponds to $\Gamma, x : A \vdash B : c$

- $\Gamma \vdash 1 : c$ is interpreted by the family $(\{*\})_{i\in I}$.
- $\Gamma \vdash \Sigma x : A.B : c$ is interpreted by the family $(\coprod_{x\in X_i} Y_{(i,x)})_{i\in I}$.
- $\Gamma \vdash A_1 + A_2 : c$ is interpreted by the family $(X_{1_i} + X_{2_i})_{i\in I}$.
- $\Gamma \vdash N : c$ is interpreted by the family $(N)_{i\in I}$, where $N$ is the set of natural numbers.
- $\Gamma \vdash \Pi x : A.B : c$ is interpreted by the family $(\prod_{x\in X_i} Y_{(i,x)})_{i\in I}$.
- $\Gamma \vdash F(n) : c$ is interpreted by the family $\langle \{0, \ldots, n_i - 1\} | i \in I\rangle$.
- $\Gamma \vdash Seq(n, A) : c$ is interpreted by the family $(X_i^{n_i})_{i\in I}$.

Interpretation of $r$-types are given next, where $\langle I, a\rangle$ corresponds to $\Gamma$, $(X_i)_{i\in I}$ corresponds to $\Gamma \vdash A : c$ and $(b_{(i,x)})_{i\in I, x\in X_i}$ corresponds to $\Gamma, x : A : c \vdash B : r$

- $\Gamma \vdash 1 : r$ is interpreted by the family $(1)_{i\in I}$, where 1 is the terminal object in $\mathbf{B}$.
- $\Gamma \vdash \exists x : A.B : r$ is interpreted by the family $(\coprod_{x\in X_i} b_{(i,x)})_{i\in I}$, where $\coprod_{x\in X_i} b_{(i,x)}$ is a small sum in $\mathbf{B}$.
- $\Gamma \vdash A_0 \times A_1 : r$ is interpreted by the family $(b_{1i} \times b_{2i})_{i\in I}$, where $b_{1i} \times b_{2i}$ is the cartesian product in $\mathbf{B}$.
- $\Gamma \vdash \forall x : A.B : r$ is interpreted by the family $(\prod_{x\in X_i} b_{(i,x)})_{i\in I}$, where $\prod_{x\in X_i} b_{(i,x)}$ is a small product in $\mathbf{B}$.

### 3.4.2 Adding collection types and computational monad

If we want to take into account computational aspects, in particular errors, we need to add a computational monad to the models of **S2**. Whereas we want to detect shape errors at compile-time, we put the other computational aspect at run-time.

Let $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$ be a model of **S2**.

**Definition 3.26 (Fibred computational monad)** *Let $S : \mathbf{B} \to \mathbf{B}$ be a computational monad on $\mathbf{B}$. We define the following pair of monads $(T^c, \eta^c, \mu^c)$ and $(T^r, \eta^r, \mu^r)$ such that $(T^c, T^r)$ is a 1-cell in the category of fibration* **Fib** *and $(\eta^c, \eta^r)$, $(\mu^c, \mu^r)$ are two 2-cells in* **Fib**.

*$T^c : \mathbf{Set} \to \mathbf{Set}$ is defined as $T^c(X) = X + 1$, the usual error monad on* **Set**.

*$T^r : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Fam}(\mathbf{B})$ is the monad on $\mathbf{Fam}(\mathbf{B})$ defined as*

$$T^r(A) = T^r((A_x)_{x\in X}) = [(SA_x)_{x\in X}, 1_{*\in 1}].$$

*The natural transformations associated are*

*$\eta^r_A : (A_x)_{x\in X} \to [(SA_x)_{x\in X}, 1_{*\in 1}]$ is defined as the pair*

- $in_0 : X \to X + 1$ *in* **Set**
- $\eta^S_{A_x} : A_x \to SA_x,\quad$ *for* $x \in X$.

*$\mu^r_A : [[(SSA_x)_{x\in X}, S1_{*\in 1}], 1_{*\in 1}] \to [(SA_x)_{x\in X}, 1_{*\in 1}]$ is defined as the pair*

- $[id, in_1] : (X + 1) + 1 \to X + 1$ *in* **Set**
- $\mu^S_{A_x} : SSA_x \to SA_x,\quad$ *for* $\langle 0, \langle 0, x\rangle\rangle \in (X + 1) + 1$
  $! : S1 \to 1,\quad$ *for* $\langle 0, \langle 1, *\rangle\rangle \in (X + 1) + 1$
  $id : 1 \to 1,\quad$ *for* $\langle 1, *\rangle \in (X + 1) + 1$

*The notion of error is given*

*$error^r_A : 1 \to [A', 1]$ is defined as the pair*

- $in_1 : 1 \to X + 1$ *in* **Set**
- $id : 1 \to 1,\quad$ *for* $* \in 1$.

The monad $T^r$ works as follows. Given a family $(A_x)_{x \in X}$, indexed over $X$, it returns a family indexed over $X+1$. The object over the element $\langle 1, * \rangle$ is the terminal object in $\mathbf{B}$. That is, in the case of error at compile-time, we have error also at run-time. The object over an element of the form $\langle 0, x \rangle$ is obtained by applying the computational monad $S$ on the corresponding object $A_x$.

List and vector types can be simply modelled in models for $\mathbf{S2}_{basic}$, since the models are extensional. Now we see the requirements we need for modelling a generic collection type.

In Section 2.3 we have seen the characterization of collections in terms of a shapely monad in a lextensive category. Now we generalize this characterization to a fibred setting. Instead of a functor (i.e. a morphism in $\mathbf{Cat}$), we consider a pair of functors (i.e. a morphism in the category of fibrations $\mathbf{Fib}$).

Let $\pi : \mathbf{Fam}(\mathbf{B}) \to Set$ be a model for $\mathbf{S2}$.

**Definition 3.27 (Fibred shapely monad)** *Let $(L^c, L^r) : \pi \to \pi$ be a morphism of fibrations. We require that $(L^c, \eta^c, \mu^c, nil^c, @^c) : \mathbf{Set} \to \mathbf{Set}$, and $(L^r, \eta^r, \mu^r, nil^r, @^r) : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Fam}(\mathbf{B})$ are shapely monads possibly with additional structure, given by nil and @, with $(\eta^c, \eta^r)$, $(\mu^c, \mu^r)$, $(nil^c, nil^r)$ and $(@^c, @^r)$ 2-cells in* $\mathbf{Fib}$.

Moreover, in order to model collection types, we require that $L^c$ and $L^r$ have traversals $(traverse^c, traverse^r)$ by the monad $T$ which are 2-cells in $\mathbf{Fib}$, and that $L^c 1$ and $L^r 1$ have decidable equality; that is, the shapely monads $L^c$ and $L^r$ satisfy the conditions in Section 2.4.

# Chapter 4

# Making phase-distinction explicit

In Chapter 2, we presented some high-level toy languages and gave a categorical model for such languages, showing some extensional and intensional examples. In the successive Chapter 3, we designed an intermediate language corresponding to a categorical account of phase-distinction, so that phase-distinction can be captured in the intermediate language. In this chapter we see that suitable translations of the toy languages into **S2** make phase-distinction explicit at syntactic level. From a semantic view, these translations correspond to interpreting the source languages in suitable categories obtained from a model of **S2**. For simplicity we consider models of **S2** obtained from fibrations of the form $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$.

We present two translations. The first one, called *simple*, translates the source languages at the top level of **S2**, without taking into account phase-distinction. This corresponds to an interpretation of the toy languages in **B**, that is the fiber $\mathbf{Fam}(\mathbf{B})_1$. The second one, called *mixed*, separates instead what can be computed at compile-time from what has to be computed at run-time. This corresponds to a more refined semantics of the toy language into $\mathbf{Fam}(\mathbf{B})$. If we want to take into account only compile-time, we can simply take the projection into the base category **Set**. Relations between the three interpretations can be established. These relations show that shape errors can be detected at compile-time.

The chapter is organized as follows. We postpone the definition of the simple and mixed translations to Section 4.4, whereas in the first sections we focus on the semantics view of shape error detection. Section 4.1 shows how we can obtain models of $\mathcal{NSC}$ and Vec starting from models of **S2**. The corresponding interpretations will be called respectively, mixed, simple and compile interpretations. Section 4.2 shows the shape-error detection result for $\mathcal{NSC}$. Section 4.3 shows the shape-error detection result for Vec. Section 4.5 summarized the contribution of the chapter.

## 4.1 From models of S2 to models of $\mathcal{NSC}$

We have said that translations of $\mathcal{NSC}$ and VEC into **S2** correspond to interpreting such languages in suitable categories obtained from a fibration that is a model of **S2**. In this section we see such categories. We restrict to the models of $\mathcal{NSC}$ and VEC based on categories with all finite limits.

From now on we suppose given $\mathbf{B}$, a cartesian closed category with small products and small sums, extensive and with a natural number object; and the fibration $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$. The fibration is a model of **S2** according to Section 3.4.1.

**Proposition 4.1** *Let $S : \mathbf{B} \to \mathbf{B}$ be a computational monad. Let $T : \mathbf{B} \to \mathbf{B}$ be the computational monad defined as $T(A) = S(A+1)$; and let $(T^c, T^r) : \pi \to \pi$ be the fibred computational monad defined as $T^r((A_x)_{x\in X}) = [(SA_x)_{x\in X}, 1_{*\in 1}]$ and $T^c(X) = X+1$. Let $L : \mathbf{B} \to \mathbf{B}$ be a shapely monad, and let $(L^c, L^r) : \pi \to \pi$ be a fibred shapely monad, satisfying the conditions for the intepretation of collection types in Section 2.4 Then the following are models of $\mathcal{NSC}$ as defined in Section 2.4.*

1. *The total category $\mathbf{Fam}(\mathbf{B})$, with the shapely monad $L^r : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Fam}(\mathbf{B})$ and with computational monad $T^r : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Fam}(\mathbf{B})$;*
2. *the base category $\mathbf{Set}$, with the shapely monad $L^c : \mathbf{Set} \to \mathbf{Set}$ and with computational monad $T^c : \mathbf{Set} \to \mathbf{Set}$; and*
3. *the fiber on 1, $\mathbf{Fam}(\mathbf{B})_1 \cong \mathbf{B}$ with shapely monad $L$ and with computational monad $T : \mathbf{B} \to \mathbf{B}$.*

*And we call them respectively, mixed, compile and simple models.*

**Proof.** It follows by lemma A.17 in Appendix A and by the properties of **Set**. Moreover, it is straightforward to see that the functors defined are monads. ∎

**Proposition 4.2** *Let $S : \mathbf{B} \to \mathbf{B}$ be a computational monad. Let $(T^c, T^r) : \pi \to \pi$ be the fibred computational monad defined as $T^r((A_x)_{x\in X}) = [(SA_x)_{x\in X}, 1_{*\in 1}]$ and $T^c(X) = X + 1$. Let $(V^c, V^r) : \pi \to \pi$ be the fibred shapely monad defined as $V^c(I) = N \times I$ and $V^r(X_i)_{i\in I}) = (X_i^m)_{\langle m,i\rangle \in N\times I}$. Then the following are models of* VEC *as defined in section 2.4.*

1. *The total category $\mathbf{Fam}(\mathbf{B})$, with shapely monad $V^r : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Fam}(\mathbf{B})$ and with computational monad $T^r : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Fam}(\mathbf{B})$;*

2. *the base category* **Set** $\cong$ **Fam(1)**, *with shapely monad* $V^c$ : **Set** $\rightarrow$ **Set** *and with computational monad* $T^c$ : **Set** $\rightarrow$ **Set**.

*And we call them respectively, mixed and compile.*

**Proof.** It follows by lemma A.17 in Appendix A and by the properties of **Set**. ∎

**Remark 4.3** Simple models of VEC in **B** can be obtained intepreting vectors as lists. So a simple model of VEC is given providing the list functor $L(X) = \coprod_{n\in N} X^n$ and the computational monad $T : \mathbf{B} \rightarrow \mathbf{B}$ defined as $T(A) = S(A+1)$.

**Remark 4.4** There are various way, starting from a model of $\mathcal{NSC}$, say **C**, to obtain models of **S2**. Possible candidates are the simple fibration and the codomain fibration. Another example could be given by fibration $\mathbf{C} \rightarrow 1 = \{*\}$.

## 4.2 Shape-error detection for $\mathcal{NSC}$

In this section we consider the language $\mathcal{NSC}$. First we show shape-error detection for the language with an instance of collection type, namely lists, and considering as computational monad the error monad. Then we show shape-error detection for a generic collection type interpreted as a shapely monad and considering a computational monad parameterized over another computational monad.

### 4.2.1 Simple case

In this subsection we fix the interpretations of the lists and the computational monads in the three models, compile, mixed and simple. The interpretation of the collection type is given respectively by the functors $L : \mathbf{B} \rightarrow \mathbf{B}$, $L^c : \mathbf{Set} \rightarrow \mathbf{Set}$ and $L^r : \mathbf{Fam(B)} \rightarrow \mathbf{Fam(B)}$. $L : \mathbf{B} \rightarrow \mathbf{B}$ is the shapely monad defined as $LX = \coprod_{n\in N} X^n$; $(L^c, L^r) : \pi \rightarrow \pi$ is the fibred shapely monad on fibration $\pi : \mathbf{Fam(B)} \rightarrow \mathbf{Set}$ defined as $L^c : \mathbf{Set} \rightarrow \mathbf{Set}$, $L^c(I) = \coprod_{n\in N} I^n$ and $L^r : \mathbf{Fam(B)} \rightarrow \mathbf{Fam(B)}$, $L^r((X_i)_{i\in I}) = (\prod_{j\in n} X_{vj})_{\langle n,v\rangle \in \coprod_{n\in N} I^n}$.

The computational monads are given by the monads $T : \mathbf{B} \rightarrow \mathbf{B}$, $T^c : \mathbf{Set} \rightarrow \mathbf{Set}$ and $T^r : \mathbf{Fam(B)} \rightarrow \mathbf{Fam(B)}$ defined as follows. $T : \mathbf{B} \rightarrow \mathbf{B}$ is the error monad $T(X) = X + 1$; $(T^c, T^r) : \pi \rightarrow \pi$ is the fibred computational monad defined as $T^c(I) = I + 1$ and $T^r((X_i)_{i\in I}) = [(X_i)_{i\in I} | 1_{*\in 1}]$.

We relate the three models of $\mathcal{NSC}$, that is the compile, mixed and simple ones.

The following lemma relates the categories **Set**, **Fam(B)** and **B**.

**Lemma 4.5** *Let* $\mathbf{B}$ *be a cartesian closed category with small products, and small sums. The following are full reflections*

$$\begin{tikzcd} \mathbf{Set} \arrow[r, hook, shift right, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathbf{Fam}(\mathbf{B}) \arrow[l, shift right, "\pi"'] \arrow[r, shift left, "\exists"] \arrow[r, phantom, "\perp"] & \mathbf{B} \arrow[l, hook, shift left] \end{tikzcd}$$

*Moreover, the functors* $\Delta : \mathbf{Set} \to \mathbf{Fam}(\mathbf{B})$, $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$ *and* $\exists : \mathbf{Fam}(\mathbf{B}) \to \mathbf{B}$ *preserve finite products small sums and natural number object (*$\Delta$ *and* $\pi$ *preserve also exponentials).*

**Proof.** The adjoint functors are given by

$$I \xleftarrow{\ \pi\ } (A_i)_{i\in I} \quad , \quad (A_i)_{i\in I} \xmapsto{\ \exists\ } \coprod_{i\in I} A_i$$

$$J \xmapsto[\Delta]{} (1)_{j\in J} \quad , \quad (B)_{*\in 1} \longleftarrow B$$

A simple check shows that all functors preserve finite products, all functors except $\mathbf{B} \hookrightarrow \mathbf{Fam}(\mathbf{B})$ preserve small sums, and the functors $\Delta$ and $\pi$ preserve exponentials. ∎

Since the interpretation of the toy languages is made into the Kleisli categories, we need to extend the functors $\pi$ and $\exists$ to the Kleisli categories.

**Lemma 4.6** *The functors* $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$ *and* $\exists : \mathbf{Fam}(\mathbf{B}) \to \mathbf{B}$ *defined as in Lemma 4.5, induce the functors* $\pi : \mathbf{Fam}(\mathbf{B})_{T^r} \to \mathbf{Set}_{T^c}$ *and* $\exists : \mathbf{Fam}(\mathbf{B})_{T^r} \to \mathbf{B}_T$*, between the Kleisli categories.*

**Proof.**

It is sufficient to show that for every object $(X_i)_{i\in I}$ in $\mathbf{Fam}(\mathbf{B})$ one has $\pi(T^r((X_i)_{i\in I})) \cong T^c(\pi((X_i)_{i\in I}))$ and $\exists(T^r((X_i)_{i\in I})) \cong T(\exists(\langle I, X_i\rangle))$.

We have:

$$\begin{array}{rcl} \pi(T^r((X_i)_{i\in I})) & \cong & \pi([(X_i)_{i\in I}|1_{*\in 1}]) \\ & \cong & I+1 \\ & \cong & T^c(I) \\ & \cong & T^c(\pi((X_i)_{i\in I})) \end{array}$$

$$\begin{aligned}
\exists(T^r((X_i)_{i\in I})) &\cong \exists([(X_i)_{i\in I}|1_{*\in 1}]) \\
&\cong (\coprod_{i\in I} X_i) + 1 \\
&\cong T(\coprod_{i\in I} X_i) \\
&\cong T(\exists((X_i)_{i\in I}))
\end{aligned}$$

$\blacksquare$

The next theorems relate the three models of $\mathcal{NSC}$.

**Theorem 4.7** *The following diagrams commute (up to a natural isomorphism)*

$$\begin{tikzcd}
 & \mathcal{NSC} \arrow[dl, "compile"'] \arrow[d, "mixed"] \arrow[dr, "simple"] & \\
\mathbf{Set}_{T^c} & \mathbf{Fam}(\mathbf{B})_{T^r} \arrow[l, "\pi"] \arrow[r, "\exists"'] & \mathbf{B}_T
\end{tikzcd}$$

**Proof.** From Lemma lemma:reflections and Lemma lemma:refl-kleisli, the interpretation of unit, products, sum commutes.

We need to show that the intepretation of the lists also commutes. We have that:

$$\exists(L^r((X_i)_{i\in I})) \cong L(\exists((X_i)_{i\in I}))$$

since

$$\begin{aligned}
\exists(L^r((X_i)_{i\in I})) &\cong \exists((\prod_{j\in n} X_{vj})_{\langle n,v\rangle\in\coprod_{n\in N} I^n}) \\
&\cong \coprod_{\langle n,v\rangle\in\coprod_{n\in N} I^n} (\prod_{j\in n} X_{vj}) \\
&\cong \coprod_{n\in N} (\coprod_{i\in I} X_i)^n \\
&\cong L(\coprod_{i\in I} X_i)) \\
&\cong L(\exists((X_i)_{i\in I}))
\end{aligned}$$

exploiting $\exists$ preserves finite products and small sums.

∎

**Lemma 4.8** *If the category* $\mathbf{B}$ *is extensive and non-trivial (i.e.* $0 \not\cong 1$*), then morphisms in* $\mathbf{Fam}(\mathbf{B})_{T^r}(1_{*\in 1}, (A_x)_{x\in X})$ *that are mapped by* $\exists$ *to* $err = in_1 \in \mathbf{B}_T(1, \coprod_{x\in X} A_x)$*, are only those isomorphic to* $\langle err^c, err^r\rangle = \langle in_1(*), id\rangle$*.*

**Proof.** Let $\langle f, g\rangle : 1_{*\in 1} \to [(A_x)_{x\in X}, 1_{*\in 1}]$ be a morphism in $\mathbf{Fam}(\mathbf{B})$. If $f = in_1$ then $\langle f, g\rangle = \langle err^c, err^r\rangle$. Suppose now that $f(*) = in_0(a)$, for some $a \in X$.

By hypothesis we have $in_a \circ g = in_1$. Moreover, since sums are disjoint, the square below is a pullback.

$$\begin{tikzcd}
1 \arrow[drr] \arrow[dr] \arrow[ddr, "g"'] & & \\
& 0 \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & 1 \arrow[d, "in_1"] \\
& A_a \arrow[r, "in_a"'] & \coprod A_x + 1
\end{tikzcd}$$

So we have that $0 \cong 1$. Contradiction. ∎

Thus one can conclude (when the hypotheses of the lemma are satisfied) that the interpretation of a closed expression of $\mathcal{NSC}$ is equal to error in the simple semantics if and only if it is in the mixed semantics.

The mixed interpretation of $\mathcal{NSC}$ is uninteresting when the interpretation of atomic types in $\Sigma$ are *trivial*, i.e. isomorphic to the terminal object.

**Theorem 4.9** *If the mixed interpretation of base types are trivial, then the following dia-*

*gram commutes (up to a natural isomorphism)*

$$\begin{tikzcd} & \mathcal{NSC} \arrow[dl, "compile"'] \arrow[d, "mixed"] \\ \mathbf{Set}_{T^c} \arrow[r, hook] & \mathbf{Fam}(\mathbf{B})_{T^r} \end{tikzcd}$$

**Remark 4.10** The syntactic counterpart of this theorem says that $x : \tau^c \vdash \tau^r \cong 1$ is provable (in extensional **S2**). Therefore, the run-time part of **S2** is not really used.

**Theorem 4.11** *The compile-time part, compile* $: \mathcal{NSC} \to \mathbf{Set}_{T^c}$ *of the mixed interpretation factors through the full sub-category of countable sets.*

**Remark 4.12** The syntactic counterpart of this result is much stronger, namely (in extensional **S2**) $\tau^c$ is provably isomorphic either to a finite cardinal or to the NNO. This means that the compile-time part of the translation uses very simple types (though the provable isomorphisms may get rather complex).

Theorem 4.7 (and Lemma 4.8) says that *(shape) errors* are detected at compile-time. Theorem 4.9 says that the run-time translation of types is *trivial*, when it is trivial for all base types. Theorem 4.11 says that the compile-time translation of types is very simple, i.e. (up to isomorphism) it is either a finite cardinal or the NNO.

### 4.2.2 General case

We show now the result for a generic collection type, defined in terms of a shapely monad and for a generic computation monad with errors at compile time.

Let $In : \mathbf{B} \to \mathbf{Fam}(\mathbf{B})$ be the inclusion functor which maps an object $X$ in $\mathbf{B}$ to the object $\langle 1, X \rangle$ in $\mathbf{Fam}(\mathbf{B})$.

From now on we suppose given the following monads for the interpretation of the collection types, that is, $L : \mathbf{B} \to \mathbf{B}$, a shapely monad, and $(L^c, L^r) : \pi \to \pi$, a fibred shapely monad. Moreover we suppose there exists a natural transformation $\rho : L^r \circ In \to In \circ L$.

Then we suppose given the following computational monads. $S : \mathbf{B} \to \mathbf{B}$, the monad $T : \mathbf{B} \to \mathbf{B}$ defined as $T(X) = S(X+1)$, and the fibred computational monad $(T^c, T^r) : \pi \to \pi$ defined by $T^c(I) = I + 1$ and $T^r((X_i)_{i \in I}) = [(SX_i)_{i \in I} | 1_{* \in 1}]$.

**Lemma 4.13** *There is a natural transformation $\sigma$ from $T^r \circ In : \mathbf{B} \to \mathbf{Fam(B)}$ and $In \circ T : \mathbf{B} \to \mathbf{Fam(B)}$, defined (see notation A.15, page 158) as:*

$\sigma_X = \langle u, (f_k)_{k\in 1+1}\rangle : [SX_{*\in 1}, 1_{*\in 1}] \to (S(X+1))_{*\in 1}$ *where*
$u : 1+1 \to 1$ *is given by* $u = [!, !]$,
$f_{in_0(*)} : SX \to S(X+1)$ *is given by* $f_{in_0(*)} = S(in_0)$ *and*
$f_{in_1(*)} : 1 \to S(X+1)$ *is given by* $f_{in_1(*)} = \eta^S_{X+1} \circ in_1$.

**Proof.**

Given $f : X \to Y$ in $\mathbf{B}$, it is straightforward to see that the following square

$$\begin{tikzcd}
{[SX_{*\in 1}, 1_{*\in 1}]} \arrow[r, "{\langle [!,!], S(in_0) | \eta^S_{X+1} \circ in_1 \rangle}"] \arrow[d, "{\langle id, S(f) | id \rangle}"'] & S(X+1)_{*\in 1} \arrow[d, "{\langle id, S([in_0 \circ f, in_1]) \rangle}"] \\
{[SY_{*\in 1}, 1_{*\in 1}]} \arrow[r, "{\langle [!,!], S(in_0) | \eta^S_{Y+1} \circ in_1 \rangle}"] & S(Y+1)_{*\in 1}
\end{tikzcd}$$

commutes.

∎

**Theorem 4.14** *1. There is a natural transformation $\alpha$ from the mixed interpretation (parameterized over the monad $T^r$) to the simple interpretation (parameterized over the monad $T$):*

$$\begin{tikzcd}
 & \mathcal{NSC} \arrow[ddl, "mixed(T^r)"'] \arrow[dr, "simple(T)"] & \\
 & & \mathbf{B} \arrow[d, hook, "In"] \\
\mathbf{Fam(B)} \arrow[rr, phantom, "\overset{\alpha}{\Rightarrow}"] & & \mathbf{Fam(B)}
\end{tikzcd}$$

2. *The following diagram commutes (up to a natural isomorphism)*

$$\begin{tikzcd} & \mathcal{NSC} \arrow[dl, "compile(T^c)"'] \arrow[d, "mixed(T^r)"] \\ \mathbf{Set} & \mathbf{Fam}(\mathbf{B}) \arrow[l, "\pi"] \end{tikzcd}$$

**Proof.**

1. First we see the types. Let $(X_i)_{i\in I}$ and $X$ be the mixed and the simple interpretation, respectively, of $\tau_0$; and let $(Y_j)_{j\in J}$ and $Y$ be the mixed and the simple interpretation, respectively, of $\tau_1$.

$$\begin{array}{lcl} \alpha_1 & = & id : 1_{*\in 1} \to 1_{*\in 1} \\ \alpha_N & = & \langle !, \langle n, *\rangle\rangle : (1)_{n\in N} \to (\coprod_{n\in N} 1)_{*\in 1} \\ \alpha_{\tau_0\times\tau_1} & = & \langle !, \alpha_{\tau_0} \times \alpha_{\tau_1}\rangle : (X_i \times Y_j)_{\langle i,j\rangle \in I\times J} \to (X\times Y)_{*\in 1} \\ \alpha_{\tau_0+\tau_1} & = & \langle !, in_0 \circ \alpha_{\tau_0} | in_1 \circ \alpha_{\tau_1}\rangle : [(X_i)_{i\in I} | (Y_j)_{j\in J}] \to (X+Y)_{*\in 1} \\ \alpha_{[\tau_0]} & = & \langle !, \rho_Y \circ L^r(\alpha_{\tau_0})\rangle : L^r((X_i)_{i\in I}) \to (LX)_{*\in 1} \end{array}$$

We look now at the terms.

A closed expression $\vdash t : \tau_0$ is interpreted as a morphism $f : 1 \to T(X)$ in the simple model and $f' : \langle 1, 1\rangle \to T^r(\langle I, X_i\rangle)$ in the mixed model.

$$\begin{tikzcd} (1)_{*\in 1} \arrow[r, "\alpha_1"] \arrow[d, "f'"'] & (1)_{*\in 1} \arrow[d, "f"] \\ T^r((X_i)_{i\in I}) \arrow[r, "\sigma \circ T^r(\alpha_{\tau_0})"] & (TX)_{*\in 1} \end{tikzcd}$$

The result is straightforward, and because $\sigma \circ T^r(\alpha_{\tau_0})$ preserves errors

2. Follows because $\pi$ is a fibration.

∎

To obtain a result similar to Lemma 4.8 for a monad parametrized on a computational monad $S$, we need to make some assumptions about the monad $S$. The following definition is adapted from the one in [Mog95].

**Definition 4.15** *Let $\mathcal{M}$ be the class of all monos over a category* $\mathbf{C}$ *with finite limits.*

*We say that* $T : \mathbf{C} \to \mathbf{C}$ *is* mono preserving *iff* $m \in \mathcal{M}$ *implies* $Tm \in \mathcal{M}$.

*We say that a natural transformation* $\sigma : S \to T$ *between (mono preserving) endofunctors is* $\mathcal{M}$-cartesian *iff for every* $m : A' \to A$ *in* $\mathcal{M}$

$$\begin{tikzcd} SA' \arrow[r, "\sigma_{A'}"] \arrow[d, hook, "Sm"'] \arrow[dr, phantom, "\lrcorner", very near start] & TA' \arrow[d, hook, "Tm"] \\ SA \arrow[r, "\sigma_A"] & TA \end{tikzcd}$$

The next lemma shows that shape-error can be detected at compile-time in the general case also.

**Lemma 4.16** *Let* $\mathbf{B}$ *be extensive and non-trivial (i.e. $0 \not\cong 1$). Let $\mathcal{M}$ be the class of all monos and suppose $S$ is mono preserving and $\eta : Id \to S$, $\mathcal{M}$-cartesian. Then the interpretation of a closed expression of $\mathcal{NSC}$ is equal to error in the simple semantics iff it is in the mixed semantics.*

**Proof.**

($\Leftarrow$) Follows from the definition of the natural transformation between mixed and simple semantics.

($\Rightarrow$) Let $(X_i)_{i\in I}$ be the interpretation of $\tau$ in $\mathbf{Fam}(\mathbf{B})$, and let $Y$ be the interpretation of $\tau$ in $\mathbf{B}$. Let $\langle f, g\rangle : (1)_{*\in 1} \to T^r((X_i)_{i\in I})$ be the interpretation of a term $t$. Since the simple interpretation of the closed expression $t$ is error, from Theorem 4.14 we have:

$$\begin{tikzcd} (1)_{*\in 1} \arrow[r, "\langle id, id\rangle"] \arrow[d, "\langle f, g\rangle"'] & (1)_{*\in 1} \arrow[d, "\langle id, inr\rangle"] \\ {[S(X_i)_{i\in I} | 1_{*\in 1}]} \arrow[r, "\sigma \circ T^r(\alpha_\tau)"'] & \langle 1, S(Y+1)_{*\in 1}\rangle \end{tikzcd}$$

The proof is by contradiction.

First note that the leftmost square below is a pullback, because sums are disjoint in **B**, and that the rightmost square is also a pullback because $\eta$ is $\mathcal{M}$-cartesian and $in_0 : Y \to Y + 1$ is a mono.

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & Y \arrow[r, "\eta"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & S(Y) \arrow[d] \\
1 \arrow[r] & Y + 1 \arrow[r, "\eta"] & S(Y + 1)
\end{tikzcd}$$

So the big square is a pullback.

Suppose now that $f(*) = in_0(x)$ (sums are disjoint in **Set**) for some $xinI$ and that $g : 1 \to S(X_x)$ according to $f$.

$$\begin{tikzcd}
1 \arrow[d, "g"'] \arrow[dr] \arrow[drr, "err^r"] & & \\
(X_x) \arrow[dr] & 0 \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & 1 \arrow[d, "\eta \circ in_1"] \\
& S(Y) \arrow[r] & S(Y + 1)
\end{tikzcd}$$

So we have that $0 \cong 1$. Contradiction.

∎

The assumptions on the monads $S$ are not particularly strict, since many computational monads, such as exceptions, side-effects, and complexity, satisfy them, as shown in [Mog95, Example 4.9].

## 4.3 Shape-error detection for VEC

We have seen in chapter 2 that VEC needs more refined categories (like arrow categories or **Fam** categories) in order to be able to express shapes and in particular that concatenation operation need a check on shapes. However, if we do not have a suitable category, we can always interpret vectors as lists. So, the simple interpretation of VEC corresponds to interpreting the vector type constructor, as list, whereas in the mixed interpretation we are able to interpret the collection types as vectors, following what was done in Chapter 2. Thus, the mixed interpretation can raise many more errors than the simple one. The approach seen for $\mathcal{NSC}$ is no longer valid and we have to use logical relations.

Our approach to logical relations follows [Pit96]. We introduce *relational structures* between two categories, and in terms of them we define *logical relations.*

**Definition 4.17 (Categorical logical relations)** *Given two categories* $\mathbf{C}_i$ *with additional structure (e.g. locos), a* relational structure *is a span*

$$\begin{tikzcd} \mathbf{C}_1 & \mathbf{R} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & \mathbf{C}_2 \end{tikzcd}$$

*consisting of a category* $\mathbf{R}$ *with additional structure and two structure preserving functors* $\pi_i : \mathbf{R} \to \mathbf{C}_i$*.*
*If* $\mathbf{C}_1 = \mathbf{C}_2 = \mathbf{C}$*, we say to have a relational structure* on $\mathbf{C}$.
*Given two interpretations,* $[\![-]\!]_i : \mathcal{L} \to \mathbf{C}_i$*, of a language* $\mathcal{L}$ *a* **R-logical relation** *between them is an interpretation* $[\![-]\!]_R : \mathcal{L} \to \mathbf{R}$ *s.t.*

$$\begin{tikzcd} & \mathcal{L} \arrow[dl, "{[\![-]\!]_1}"'] \arrow[d, "{[\![-]\!]_R}"] \arrow[dr, "{[\![-]\!]_2}"] & \\ \mathbf{C}_1 & \mathbf{R} \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & \mathbf{C}_2 \end{tikzcd}$$

When the additional structure on $\mathbf{C}_i$ includes a monad $T_i$, we must have a monad $T_R$ also on $\mathbf{R}$, and moreover the following diagrams must commute

$$\begin{tikzcd} \mathbf{C}_1 \arrow[d, "T_1"'] & \mathbf{R} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "T_R"] & \mathbf{C}_2 \arrow[d, "T_2"] \\ \mathbf{C}_1 & \mathbf{R} \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & \mathbf{C}_2 \end{tikzcd}$$

[Pit96] considers relational structures on one category, and in that setting $T_R$ is called the relational action of $T$ on $\mathbf{R}$.

Let $\mathbf{Fam(B)}^\wedge$ be the category of spans in $\mathbf{Fam(B)}$, or better the functor category from the category $\wedge$ and the category $\mathbf{Fam(B)}$ (see definition A.18, page 159). Consider the subcategory $\mathbf{R}$ of $\mathbf{Fam(B)}^\wedge$ defined as

$$\begin{tikzcd}
\mathbf{R} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{Fam(B)}^\wedge \arrow[d] \\
\mathbf{Fam(B)} \times \mathbf{B} \arrow[r, "Id \times In"'] & \mathbf{Fam(B)} \times \mathbf{Fam(B)}
\end{tikzcd}$$

Let $T_R : \mathbf{R} \to \mathbf{R}$ be the monad defined as follows: given a span

$$(X_i)_{i \in I} \xleftarrow{\langle h, (\beta_k)_{k \in K} \rangle} (R_k)_{k \in K} \xrightarrow{\langle !, (\gamma_k)_{k \in K} \rangle} (X)_{* \in 1}$$

$T_R$ returns the span

$$[S(X_i)_{i \in I}, 1_{* \in 1}] \xleftarrow{\langle u, (f_l)_{l \in K+1} \rangle} [(SR_k)_{k \in K}, S(X+1)_{* \in 1}] \xrightarrow{\langle v, (g_l)_{l \in K+1} \rangle} S(X+1)_{* \in 1}$$

where
$\langle u, (f_l)_{l \in K+1} \rangle = \langle [in_0 \circ h, in_1], S(\beta_k)_{k \in K} | !_{* \in 1} \rangle$
and
$\langle v, (g_l)_{l \in K+1} \rangle = \langle !, S(in_0 \circ \gamma_k)_{k \in K} | id_{* \in 1} \rangle$;

Suppose given the functors and fibred functors $In : \mathbf{B} \to \mathbf{Fam(B)}$, $S : \mathbf{B} \to \mathbf{B}$, $T : \mathbf{B} \to \mathbf{B}$, $(T^r, T^c) : \pi \to \pi$, defined as in Section 4.2.2. Let $L : \mathbf{B} \to \mathbf{B}$, $L(X) = \coprod_{n \in N} X^n$, be the list functor and let $(V^c, V^r) : \pi \to \pi$, $V^c(I) = N \times I$ and $V^r((X_i)_{i \in I}) = (X_i^m)_{\langle m, i \rangle \in N \times I}$ be the vector functor. Moreover let $\rho : V^r \circ In \to In \circ L$ be the natural transformation given by the unit $\eta$ of the adjunction $(\exists, In)$ in Lemma 4.5, more formally $\rho_X = \eta_{(X^n)_{n \in N}}$.

The next theorems relate the three models of VEC.

**Theorem 4.18** *1. There is a logical relation between the mixed and simple interpreta-*

*tions.*

$$\begin{tikzcd}
 & \textsc{Vec} \arrow[dl, "mixed"'] \arrow[d] \arrow[dr, "simple"] & \\
\mathbf{Fam}(\mathbf{B})_{T^r} & \mathbf{R}_{T_R} \arrow[l] \arrow[r] & \mathbf{B}_T
\end{tikzcd}$$

*2. The following diagram commutes (up to a natural isomorphism)*

$$\begin{tikzcd}
 & \textsc{Vec} \arrow[dl, "compile"'] \arrow[d, "mixed"] \\
\mathbf{Set}_{T^c} & \mathbf{Fam}(\mathbf{B})_{T^r} \arrow[l, "\pi"]
\end{tikzcd}$$

**Proof.**

1. Consider $\mathbf{B}$ as included in $\mathbf{Fam}(\mathbf{B})$.

   We need to give the interpretation of the language in $\mathbf{R}$. Intepretation of basic types are the following, given a type $\tau$ and its mixed and simple semantics, respectively $\langle I, X_i \rangle$ and $\langle 1, Y \rangle$, the corresponding span is :

$$(X_i)_{i\in I} \xleftarrow{id} (X_i)_{i\in I} \xrightarrow{\alpha_\tau} (Y)_{*\in 1}$$

   where $\alpha$ is the natural transformation of theorem 4.14.

   In general, given two spans,

$$X \xleftarrow{r_1} R \xrightarrow{r_2} Y$$

$$Z \xleftarrow{s_1} S \xrightarrow{s_2} W$$

   the product is given by

$$X \times Z \xleftarrow{r_1 \times s_1} R \times S \xrightarrow{r_2 \times s_2} Y \times W$$

and the exponential is given by

$$[X \to Z] \xleftarrow{p_1} U \xrightarrow{p_2} [Y \to W]$$

where $U$ is obtained by the following pullbacks

$$\begin{tikzcd}
 & & U \arrow[dl] \arrow[dr] & & \\
 & P \arrow[dl] \arrow[dr] & & Q \arrow[dl] \arrow[dr] & \\
{[X \to Z]} \arrow[dr] & & {[R \to S]} \arrow[dl] \arrow[dr] & & {[Y \to W]} \arrow[dl] \\
 & {[R \to Z]} & & {[R \to W]} &
\end{tikzcd}$$

For collection types, let

$$(X_i)_{i \in I} \xleftarrow{\langle h, (\beta_k)_{k \in K} \rangle} (R_k)_{k \in K} \xrightarrow{\langle !, (\gamma_k)_{k \in K} \rangle} (X)_{* \in 1}$$

be the span corresponding to the type $\tau$, then

$$V^r((X_i)_{i \in I}) \xleftarrow{V^r(\langle h, (\beta_k)_{k \in K} \rangle)} V^r((R_k)_{k \in K}) \xrightarrow{\rho \circ V^r(\langle !, (\gamma_k)_{k \in K} \rangle)} (LX)_{* \in 1}$$

is the relation action corresponding to the collection type.

2. Follows because $\pi$ is a fibration.

∎

The following result is weaker than the corresponding one for $\mathcal{NSC}$. For VEC we only have that if the interpretation of a closed expression of VEC is equal to error in the simple semantics, then is equal to error in the mixed semantics. The vice-versa does not hold. In fact the mixed semantics can capture more errors, since it is more refined. In particular the mixed semantics captures the error due to the concatenation of two arrays which entries have different shapes.

**Lemma 4.19** *Let* $\mathbf{B}$ *be extensive and non-trivial (i.e.* $0 \not\cong 1$*). Let* $\mathcal{M}$ *be the class of all monos and suppose* $S$ *is mono preserving and* $\eta : Id \to S$*,* $\mathcal{M}$*-cartesian. Then the interpretation of a closed expression of* VEC *is equal to error in the simple semantics only if it is in the mixed semantics.*

**Proof.**

Let

$$(X_i)_{i \in I} \xleftarrow{\langle h, (\beta_k)_{k \in K} \rangle} (R_k)_{k \in K} \xrightarrow{\langle !, (\gamma_k)_{k \in K} \rangle} (X)_{* \in 1}$$

be the interpretation of $\tau$ in $\mathbf{R}$. Let the following morphism in $\mathbf{R}$ be the interpretation of term $t$.

$$\begin{tikzcd}
1_{*\in 1} \arrow[d, "\langle _, _ \rangle"'] & 1_{*\in 1} \arrow[l] \arrow[r] \arrow[d, "\langle r, s \rangle"'] & 1_{*\in 1} \arrow[d, "\langle !, \eta \circ in_1 \rangle"'] \\
{[S(X_i)_{i \in I}, 1_{*\in 1}]} & {[(SR_k)_{k \in K}, S(X+1)_{*\in 1}]} \arrow[l, "\langle u, (f_l)_{l \in K+1} \rangle"'] \arrow[r, "\langle v, (g_l)_{l \in K+1} \rangle"] & S(X+1)_{*\in 1}
\end{tikzcd}$$

where
$\langle u, (f_l)_{l \in K+1} \rangle = \langle [in_0 \circ h, in_1], S(\beta_k)_{k \in K} | !_{*\in 1} \rangle$
and
$\langle v, (g_l)_{l \in K+1} \rangle = \langle !, S(in_0 \circ \gamma_k)_{k \in K} | id_{*\in 1} \rangle$.

If $r(*) = in_1(*)$, then $s : 1 \to S(X+1)$, so the mixed interpreation is given by $\langle in_1 \circ r, ! \circ s \rangle$ and corresponds to error.

Suppose now that $r(*) = in_0(a)$ (sums are disjoint in $\mathbf{Set}$) for some $a \in I$, and that $s : 1 \to S(R_a)$ according to $r$.

$$
\begin{tikzcd}
1 \arrow[d, "s"'] \arrow[dr] \arrow[drr] & & \\
S(R_a) \arrow[dr, "S(\gamma_a)"'] & 0 \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & 1 \arrow[d, "\eta \circ in_1"] \\
& S(X) \arrow[r, "S(in_0)"'] & S(X+1)
\end{tikzcd}
$$

The above pullback holds for the same reason as in Lemma 4.16.

So we have that $0 \cong 1$. Contradiction.

∎

## 4.4 Translations of $\mathcal{NSC}$ (and Vec) into S2

This section describes in detail the simple and the mixed translations of $\mathcal{NSC}$ in **S2** and thus points out the role of **S2** as intermediate language. In the translation of the collection type we use some particular instances of a shapely monad, such as the monad for the list type and the monad for vector type (in Vec). We use some auxiliary notation and notational conventions explained in Section 4.4.3.

### 4.4.1 The simple translation

Before describing the simple translation, we define the error monad in **S2**.

**Definition 4.20 (Error monad)** *Given $A : r$ the type $TA : r$ is given by $TA = A + 1$. The corresponding monad structure is defined by*

| | |
|---|---|
| $val$ | $A \to TA$ |
| | $val(x) = in_0(x)$ |
| $let$ | $(A \to TB), TA \to TB$ |
| | $let(f, in_0(x)) \;=\; f(x)$ <br> $let(f, in_1(*)) \;=\; in_1(*)$ |

*We write $[M]_T$ for $val(M)$ and $(\text{let } x{\Leftarrow}M \text{ in } N)$ for $let([x : A]N, M)$.*

More generally, we can define the following monad:

**Definition 4.21 (Error monad plus)** *Let $S$ be a computational monad. Given $A : r$ the type $TA : r$ is given by $TA = S(A + 1)$.*

*The corresponding monad structure is defined by*

| | |
|---|---|
| $val$ | $A \to TA$ |
| | $val(x) = val^S(in_0(x))$ |
| $let$ | $(A \to TB), TA \to TB$ |
| | $let(f, x) \;=\; let^S([f, val^S \circ in_1], x)$ |

The simple translation $\_^* : \mathcal{NSC} \to \mathbf{S2}$ corresponds to translating $\mathcal{NSC}$ into a simply typed lambda calculus with unit, sums, products, NNO and list objects (extended with the analogue $\Sigma^*$ of the $\mathcal{NSC}$-signature $\Sigma$). It has the following pattern:

- types $\vdash_{\mathcal{NSC}} \tau$ are translated to $r$-types $\vdash_{\mathbf{S2}} \tau^* : r$

| $\tau$ of $\mathcal{NSC}$ | $\tau^* : r$ of $\mathbf{S2}$ |
|---|---|
| $1$ | $1$ |
| $N$ | $\exists n : \mathbf{N}.1$ |
| $D$ | $D$ |
| $\tau_1 \times \tau_2$ | $\tau_1^* \times \tau_2^*$ |
| $\tau_1 + \tau_2$ | $\tau_1^* + \tau_2^*$ |
| $[\tau]$ | $\exists n : \mathbf{N}.n{\Rightarrow}\tau^*$ |
| arities of $\mathcal{NSC}$ | |
| $\overline{\tau} \to \tau$ | $\overline{\tau}^* \to T\tau^*$ |
| $\overline{\sigma}, \overline{\tau} \to \tau$ | $\overline{\sigma}^*, \overline{\tau}^* \to T\tau^*$ |

- terms $\overline{x} : \overline{\tau} \vdash_{\mathcal{NSC}} e : \tau$ are translated to terms $\overline{x} : \overline{\tau}^* : r \vdash_{\mathbf{S2}} e^* : T\tau^* : r$, where $T$ is the error monad on $r$-types.

  | $e : \tau$ of $\mathcal{NSC}$ | $e^* : T\tau^* : r$ of $\mathbf{S2}$ |
  |---|---|
  | $x$ | $[x]_T$ |
  | $op(\overline{e})$ | $\mathrm{let}\ \overline{x}\Leftarrow\overline{e}^*\ \mathrm{in}\ op^*(\overline{x})$ |
  | $c(\overline{f}, \overline{e})$ | $\mathrm{let}\ \overline{x}\Leftarrow\overline{e}^*\ \mathrm{in}\ c^*(\overline{f}^*, \overline{x})$ |
  | $\lambda\overline{x} : \overline{\tau}.e$ | $\lambda\overline{x} : \overline{\tau}^*.e^*$ |

  when a term-constructor $c$ cannot raise an error (i.e. is not marked by † in Figure 2.1), we translate $c(\overline{f}, \overline{e})$ to $\mathrm{let}\ \overline{x}\Leftarrow\overline{e}^*\ \mathrm{in}\ [c^*(\overline{f}^*, \overline{x})]_T$.

- term-constructors $c : \overline{\sigma}, \overline{\tau} \to \tau$ are translated to terms $\vdash_{\mathbf{S2}} c^* : \overline{\sigma}^*, \overline{\tau}^* \to T\tau^* : r$, or to $\vdash_{\mathbf{S2}} c^* : \overline{\sigma}^*, \overline{\tau}^* \to \tau^* : r$ when $c$ cannot raise an error.

Figure 4.1 gives $c^*$ for the term-constructors $c$ of $\mathcal{NSC}$ which can raise an error (and for *length* and *enum*). We leave readers to work out for themself the definition of $c^*$ for the other term-constructors. In Figure 4.1 we use ML-style notation for function definitions and other auxiliary notation for **S2**, which is defined in section 4.4.3.

**Remark 4.22** Given a $\mathcal{NSC}$-signature $\Sigma$, its analogue $\Sigma^*$ in **S2** is defined as follows:

- a constant type $D : r$ for each atomic type $D$ in $\Sigma$;
- a constant term $\overline{x} : \overline{\tau}^* : r \vdash op^*(\overline{x}) : T\tau^* : r$ for each operation $op : \overline{\tau} \to \tau$ in $\Sigma$.

Now we informally describe the correspondence between simple translation and simple interpretation, that is, the denotations in the fibration $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$ of types and terms of **S2** obtained as simple translations of types and terms of $\mathcal{NSC}$, are isomorphic to the denotations of the types and terms of $\mathcal{NSC}$ in $\mathbf{B}$ (according to Chapter 2). Types $\vdash_{\mathcal{NSC}} \tau$ are translated to $r$-types $\vdash_{\mathbf{S2}} \tau^* : r$, which are interpretated in families indexed on 1, that is in the fiber 1, which correspond to $\mathbf{B}$. It is straightforward to see that the interpretation of the translations corresponds to directly interpret $\mathcal{NSC}$ into $\mathbf{B}$. For example, the type $N$ of $\mathcal{NSC}$ is interpreted in the natural number object of $\mathbf{B}$. The simple translation in **S2** is $\exists n : N.1$ which is interpreted as $\coprod_{n\in N} 1$, isomorphic to the natural number object in $\mathbf{B}$.

### 4.4.2 The mixed translation

In this section also, before describing the mixed translation, we define the corresponding error monad in **S2**.

| $c^*$ | arity and ML-like definition of $c^*(\overline{f},\overline{x})$ |
|---|---|
| $\mathsf{err}^*$ | $T\tau$ |
| | $\mathsf{err}^* = in_1(*)$ |
| $\mathsf{at}^*$ | $(\tau_1 \to T\tau_2), \tau_1 \to T\tau_2$ |
| | $\mathsf{at}^*(f,x) = fx$ |
| $\mathsf{for}^*$ | $(\mathbf{N}, \tau \to T\tau), \tau, \mathbf{N} \to T\tau$ |
| | $\mathsf{for}^*(f,x,0) \quad = \quad [x]_T$ <br> $\mathsf{for}^*(f,x,sn) \quad = \quad \mathrm{let}\ y \Leftarrow \mathsf{for}^*(f,x,n)\ \mathrm{in}\ f(n,y)$ |
| $\mathsf{case}^*$ | $(\tau_1 \to T\tau), (\tau_2 \to T\tau), \tau_1 + \tau_2 \to T\tau$ |
| | $\mathsf{case}^*(f_0, f_1, in_i(x)) = f_i(x) \qquad (i = 0, 1)$ |
| $\mathsf{map}^*$ | $(\tau_1 \to T\tau_2), L\tau_1 \to T(L\tau_2)$ |
| | $\mathsf{map}^*(f, nil) \quad = \quad [nil]_T$ <br> $\mathsf{map}^*(f, h :: t) \quad = \quad \mathrm{let}\ x \Leftarrow f(h)\ \mathrm{in}\ \mathrm{let}\ l \Leftarrow \mathsf{map}^*(f,t)\ \mathrm{in}\ [x :: l]_T$ |
| $\mathsf{length}^*$ | $L\tau \to \mathbf{N}$ |
| | $\mathsf{length}^*(nil) \quad = \quad 0$ <br> $\mathsf{length}^*(h :: t) \quad = \quad s(\mathsf{length}^*(t))$ |
| $\mathsf{get}^*$ | $L\tau \to T\tau$ |
| | $\mathsf{get}^*(nil) \quad = \quad \mathsf{err}^*$ <br> $\mathsf{get}^*(h :: t) \quad = \quad \mathsf{case}^*([h]_T, \mathsf{err}^*, eq(0, \mathsf{length}^*(t)))$ <br> where $eq : \mathbf{N}, \mathbf{N} \to 1 + 1$ is equality for the NNO |
| $\mathsf{zip}^*$ | $L\tau_1, L\tau_2 \to T(L(\tau_1 \times \tau_2))$ |
| | $\mathsf{zip}^*(nil, nil) \quad = \quad [nil]_T$ <br> $\mathsf{zip}^*(nil, h_2 :: t_2) \quad = \quad \mathsf{err}^*$ <br> $\mathsf{zip}^*(h_1 :: t_1, nil) \quad = \quad \mathsf{err}^*$ <br> $\mathsf{zip}^*(h_1 :: t_1, h_2 :: t_2) \quad = \quad \mathrm{let}\ l \Leftarrow \mathsf{zip}^*(t_1, t_2)\ \mathrm{in}\ [\langle h_1, h_2 \rangle :: l]_T$ |
| $\mathsf{split}^*$ | $L\tau, L\mathbf{N} \to T(L(L\tau))$ |
| | $\mathsf{split}^*(nil, nil) \quad = \quad [nil]_T$ <br> $\mathsf{split}^*(h :: t, nil) \quad = \quad \mathsf{err}^*$ <br> $\mathsf{split}^*(x, 0 :: p) \quad = \quad \mathrm{let}\ l \Leftarrow \mathsf{split}^*(x,p)\ \mathrm{in}\ [nil :: l]_T$ <br> $\mathsf{split}^*(nil, sn :: p) \quad = \quad \mathsf{err}^*$ <br> $\mathsf{split}^*(h :: t, sn :: p) \quad = \quad \mathrm{let}\ l \Leftarrow \mathsf{split}^*(t, n :: p)\ \mathrm{in}\ cons'(x, l)$ <br> where $cons'(x, nil) = in_1(*) \mid cons'(x, h :: t) = [(x :: h) :: t]_T$ |

Figure 4.1: Simple translation of $\mathcal{NSC}$ term-constructors

**Definition 4.23 (Error monad on families of types)** *Given* $x : A : c \vdash A' : r$ *the family* $x : TA : c \vdash T'([x : A]A', x) : r$ *is given by* $TA = A + 1$ *and*

$$\begin{array}{lcl} T'([x : A]A', in_0(x)) & = & A'(x) \\ T'([x : A]A', in_1(*)) & = & 1 \end{array}$$

*We write* $T'(A, A', M)$ *for* $T'([x : A]A', M)$.

*The corresponding monad structure is defined by pair of compatible terms*

| | |
|---|---|
| $val$<br>$val'$ | $A \to TA$<br>$\forall x : A.A' \to T'(A, A', val(x))$ |
| | $val(x) = in_0(x)$<br>$val'(x, x') = x'$ |
| $let$<br>$let'$ | $(A \to TB), TA \to TB$<br>$\forall f, x : (A \to TB), TA,$<br>$(\forall x : A.A' \to T'(B, B', fx)), T'(A, A', x) \to T'(B, B', let(f, x))$ |
| | $let(f, in_0(x)) = f(x)$<br>$let'(f, in_0(x), f', x') = f'(x, x')$<br>$let(f, in_1(*)) = in_1(*)$<br>$let'(f, in_1(*), f', *) = *$ |

*We write* $[M]_T$ *for* $val(M)$, $(\mathrm{let}\, x \Leftarrow M \,\mathrm{in}\, N)$ *for* $let([x : A]N, M)$, $M'$ *for* $val'(M, M')$ *and* $(\mathrm{let}'\, x, x' \Leftarrow M, M' \,\mathrm{in}\, N')$ *for* $let'([x : A]N, M, [x : A, x' : A']N', M')$.

More generally we can define the following:

**Definition 4.24 (Error monad plus on families of types)** *Let* $S$ *be a computational monad. Given* $x : A : c \vdash A' : r$ *the family* $x : TA : c \vdash T'([x : A]A', x) : r$ *is given by* $TA = A + 1$ *and*

$$\begin{array}{lcl} T'([x : A]A', in_0(x)) & = & S(A'(x)) \\ T'([x : A]A', in_1(*)) & = & 1 \end{array}$$

*We write* $T'(A, A', M)$ *for* $T'([x : A]A', M)$.

**Remark 4.25** Unfortunately, in intensional **S2** it is not possible to mimic the definition of the monad $T$ from $S$ (the difficulty is in the definition $let$). The reason is lack of extensivity.

In defining the mixed translation of collection type $[\tau]$ as lists, we use the list type-constructor $(L, L')$ acting on families of $r$-types.

**Definition 4.26 (List objects for families of types)** *Given* $x : A : c \vdash A' : r$ *the family* $x : LA : c \vdash L'([x : A]A', x) : r$ *is given by* $LA = \Sigma n : \mathbf{N}.V(n, A)$ *and* $L'([x : A]A', \langle n, v \rangle) = \forall i : n.A'(vi)$. *We write* $L'(A, A', M)$ *for* $L'([x : A]A', M)$.

The alternative translation of types $[\tau]$ as vectors, uses the array type-constructor $(V, V')$ acting on families of $r$-types.

**Definition 4.27 (Array for families of types)** *Given* $x : A : c \vdash A' : r$ *the family* $x : VA : c \vdash V'([x : A]A', x) : r$ *is given by* $VA = \mathbf{N} \times A$ *and* $V'([x : A]A', \langle n, e\rangle) = A'(e)^n$. *We write* $V'(A, A', M)$ *for* $V'([x : A]A', M)$.

**Remark 4.28** A better choice is given by modifying the preceding definition for taking into account vectors of length zero. We could define $VA = 1 + \mathbf{N} \times A$ and $V'([x : A]A', \langle n, e\rangle) = 1|A'(e)^{sn}$.

The mixed translation highlights phase-distinction between compile-time and run-time, and exploits fully the features of **S2** (extended with the analogue of the $\mathcal{NSC}$-signature $\Sigma$). The mixed translation consists of a pair of translations $(\_^c, \_^r) : \mathcal{NSC} \to \mathbf{S2}$ s.t.

- types $\vdash_{\mathcal{NSC}} \tau$ are translated to families of $r$-types $x : \tau^c : c \vdash_{\mathbf{S2}} \tau^r(x) : r$

| $\tau$ of $\mathcal{NSC}$ | $x : \tau^c : c$ and | $\tau^r(x) : r$ of **S2** |
|---|---|---|
| $1$ | $\_ : 1$ | $1$ |
| $N$ | $\_ : \mathbf{N}$ | $1$ |
| $D$ | $\_ : 1$ | $D$ |
| $\tau_1 \times \tau_2$ | $\langle x_1, x_2\rangle : \tau_1^c \times \tau_2^c$ | $\tau_1^r(x_1) \times \tau_2^r(x_2)$ |
| $\tau_1 + \tau_2$ | $in_i(x_i) : \tau_1^c + \tau_2^c$ | $\tau_i^r(x_i)$ |
| $[\tau]$ | $\langle n, v\rangle : \Sigma n : \mathbf{N}.V(n, \tau^c)$ | $\forall i : n.\tau^r(vi)$ |
| arities of $\mathcal{NSC}$ | | |
| $\overline{\tau} \to \tau$ | $f : \overline{\tau}^c \to T\tau^c$ | $\forall \overline{x} : \overline{\tau}^c.\overline{\tau}^r \to T'(\tau^c, \tau^r, f(\overline{x}))$ |
| $\overline{\sigma}, \overline{\tau} \to \tau$ | $F : \overline{\sigma}^c, \overline{\tau}^c \to T\tau^c$ | $\forall \overline{f}, \overline{x} : \overline{\sigma}^c, \overline{\tau}^c.\overline{\sigma}^r, \overline{\tau}^r \to T'(\tau^c, \tau^r, F(\overline{f}, \overline{x}))$ |

  Alternatively, using arrays:

| $\tau$ | $x : \tau^c : c$ and | $\tau^r(x) : r$ of **S2** |
|---|---|---|
| $[\tau]$ | $\langle n, x\rangle : \mathbf{N} \times \tau^c$ | $\forall \_ : n.\tau^r(x)$ |

- terms $\overline{x} : \overline{\tau} \vdash_{\mathcal{NSC}} e : \tau$ are translated to pairs of *compatible* terms
  $\overline{x} : \overline{\tau}^c : c \vdash_{\mathbf{S2}} e^c : T\tau^c : c$ and
  $\overline{x} : \overline{\tau}^c : c, \overline{x}' : \overline{\tau}^r : r \vdash_{\mathbf{S2}} e^r : T'(\tau^c, \tau^r, e^c) : r$
  where $(T, T')$ is the error monad on families of $r$-types.

| $e:\tau$ of $\mathcal{NSC}$ | $e^c : T\tau^c : c$ and | $e^r : T'(\tau^c, \tau^r, e^c) : r$ of **S2** |
|---|---|---|
| $x$ | $[x]_T$ | $x'$ |
| $op(\overline{e})$ | $\mathrm{let}\,\overline{x}\Leftarrow\overline{e}^c\,\mathrm{in}\,op^c(\overline{x})$ | $\mathrm{let}'\,\overline{x},\overline{x}'\Leftarrow\overline{e}^c,\overline{e}^r\,\mathrm{in}\,op^r(\overline{x},\overline{x}')$ |
| $c(\overline{f},\overline{e})$ | $\mathrm{let}\,\overline{x}\Leftarrow\overline{e}^c\,\mathrm{in}\,c^c(\overline{f}^c,\overline{x})$ | $\mathrm{let}'\,\overline{x},\overline{x}'\Leftarrow\overline{e}^c,\overline{e}^r\,\mathrm{in}\,c^r(\overline{f}^c,\overline{f}^r,\overline{x},\overline{x}')$ |
| $\lambda\overline{x}:\overline{\tau}.e$ | $\lambda\overline{x}:\overline{\tau}^c.e^c$ | $\Lambda\overline{x}:\overline{\tau}^c.\lambda\overline{x}':\overline{\tau}^r.e^r$ |

when a term-constructor $c$ cannot raise an error, we translate $c(\overline{f},\overline{e})$ to $\mathrm{let}\,\overline{x}\Leftarrow\overline{e}^c\,\mathrm{in}\,[c^c(\overline{f}^c,\overline{x})]_T$ and $\mathrm{let}'\,\overline{x},\overline{x}'\Leftarrow\overline{e}^c,\overline{e}^r\,\mathrm{in}\,c^r(\overline{f}^c,\overline{f}^r,\overline{x},\overline{x}')$.

- term-constructors $c:\overline{\sigma},\overline{\tau}\to\tau$ are translated to pairs of *compatible* terms
  $\vdash_{\mathbf{S2}} c^c : \overline{\sigma}^c,\overline{\tau}^c \to T\tau^c : c$ and
  $\vdash_{\mathbf{S2}} c^r : \forall\overline{f},\overline{x}:\overline{\sigma}^c,\overline{\tau}^c.\overline{\sigma}^r,\overline{\tau}^r \to T'(\tau^c,\tau^r,c^c(\overline{f},\overline{x})) : r$, or to
  $\vdash_{\mathbf{S2}} c^c : \overline{\sigma}^c,\overline{\tau}^c \to \tau^c : c$ and
  $\vdash_{\mathbf{S2}} c^r : \forall\overline{f},\overline{x}:\overline{\sigma}^c,\overline{\tau}^c.\overline{\sigma}^r,\overline{\tau}^r \to \tau^r(c^c(\overline{f},\overline{x})) : r$ when $c$ cannot raise an error.

Figure 4.2 and 4.3 give $c^c$ and $c^r$ for the term-constructors $c$ of $\mathcal{NSC}$ which can raise an error (and for length), using the list type-constructor to interpret the finite sequence types.

A sketch of the translation using arrays is given in figure 4.5. It looks simpler except in the case of concatenation (in which we have to check if the shape is the same).

The tables are organized as follows:

- For each term-constructor $c$ of $\mathcal{NSC}$ we write

  | | |
  |---|---|
  | $c^c$ | $\overline{f},\overline{x}:\overline{\sigma},\overline{\tau}\to T(\tau)$ |
  | $c^r$ | $\overline{\sigma}',\overline{\tau}'\to T'(\tau,\tau',c^c(\overline{f},\overline{x}))$ |

  to mean that $c^c:\overline{\sigma},\overline{\tau}\to T(\tau)$ and $c^r:\forall\overline{f},\overline{x}:\overline{\sigma},\overline{\tau}.\overline{\sigma}',\overline{\tau}'\to T'(\tau,\tau',c^c(\overline{f},\overline{x}))$ .
- Each case of the ML-like definition of $c^c$ is immediately followed by the corresponding case for $c^r$.
- Arguments of trivial run-time type (i.e. isomorphic to 1) are omitted. This happens in the definition of for and split.
- The definition of $c^r$ is omitted when the type of the result is trivial. This happens in the definition of length.

**Remark 4.29** Given a $\mathcal{NSC}$-signature $\Sigma$, its analogue $\Sigma^{c,r}$ in **S2** is defined as follows:

- a constant type $D:r$ for each atomic type $D$ in $\Sigma$;

- a pair of compatible constant terms $\overline{x} : \overline{\tau}^c : c \vdash op^c(\overline{x}) : T\tau^c : c$ and $\overline{x} : \overline{\tau}^c : c, \overline{x}' : \overline{\tau}^r : r \vdash op^r(\overline{x}, \overline{x}') : T'(\tau, \tau', op^c(\overline{x})) : r$ for each $op : \overline{\tau} \to \tau$ in $\Sigma$.

At the semantic level this translation of $\Sigma$ imposes strong restrictions. For instance, consider the translation of $op : D, D \to 1 + 1$. This is given by $op^c : 1 + 1$ and $op^r : D, D \to 1$ (when ignoring arguments of trivial type). Therefore, we cannot interpret $op$ as equality on $D$, because both $op^c$ and $op^r$ are constant. The mixed translation can only cope with *shapely* operations (see [Jay95b]), where the shape of the result is determined uniquely by the shape of the arguments. However, in **S2** one can give a type to non-shapely operations, for instance $op : D, D \to (\exists x : 1 + 1.1)$.

Now we informally describe the correspondence between mixed translation and mixed interpretation, that is, the denotations in the fibration $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$ of types and terms of **S2** obtained as mixed translations of types and terms of $\mathcal{NSC}$, are isomorphic to the denotations of the types and terms of $\mathcal{NSC}$ in $\mathbf{Fam}(\mathbf{B})$ (according to Chapter 2). Types $\vdash_{\mathcal{NSC}} \tau$ are translated to families of $r$-types $x : \tau^c : c \vdash_{\mathbf{S2}} \tau^r(x) : r$. It is straightforward to see that the interpretation of the translation corresponds to directly interpret $\mathcal{NSC}$ into $\mathbf{Fam}(\mathbf{B})$. For example, the type $N$ of $\mathcal{NSC}$ is interpreted as $(1)_{n \in N}$ in $\mathbf{Fam}(\mathbf{B})$. The mixed translation in **S2** is $x : N : c \vdash_{\mathbf{S2}} 1 : r$ which is interpreted as $(1)_{n \in N}$. The type of vectors $[\tau]$ of $\mathcal{NSC}$ is interpreted as $V((X_i)_{i \in I}) = (X_i^m)_{\langle m,i \rangle \in N \times I}$, where $(X_i)_{i \in I}$ is the interpretation of $\tau$. The mixed translation in **S2** is $\langle n, x \rangle : N \times \tau^c : c \vdash_{\mathbf{S2}} \forall\_ : n.\tau^r(x) : r$ which is interpreted as $(X_i^m)_{\langle m,i \rangle \in N \times I}$.

### 4.4.3 Auxiliary notation and notational conventions

This section introduces auxiliary notations and notational conventions for **S2**.

**Auxiliary notation for types.**

- $A \Rightarrow B$ stands for $\forall\_ : A.B$
- $A \to B$ stands for $\Pi\_ : A.B$ when $A$ and $B$ have sort $c$
- $A \times B$ stands for $\Sigma\_ : A.B$ when $A$ and $B$ have sort $c$
- $A^n$ stands for $V(n, A)$
- $\mathbf{N}^r$ stands for $\exists n : \mathbf{N}.1$ (the NNO of sort $r$)
- $L(A)$ stands for $\Sigma n : \mathbf{N}.V(n, A)$ (the list object of sort $c$)

| $c$ | arity and ML-like definition of $c^c(\overline{f},\overline{x})$ and $c^r(\overline{f},\overline{f}',\overline{x},\overline{x}')$ |
|---|---|
| $\mathsf{err}^c$<br>$\mathsf{err}^r$ | $T\tau$<br>$T'(\tau,\tau',err^c)$ |
| | $\mathsf{err}^c = in_1(*)$<br>$\mathsf{err}^r = *$ |
| $\mathsf{at}^c$<br>$\mathsf{at}^r$ | $f,x:(\tau_1\to T\tau_2),\tau_1\to T\tau_2$<br>$(\forall x:\tau_1.\tau_1'\to T'(\tau_2,\tau_2',fx)),\tau_1'\to T'(\tau_2,\tau_2',\mathsf{at}^c(f,x))$ |
| | $\mathsf{at}^c(f,x)=fx$<br>$\mathsf{at}^r(f,x,f',x')=f'(x,x')$ |
| $\mathsf{for}^c$<br>$\mathsf{for}^r$ | $f,x,n:(\mathbf{N},\tau\to T\tau),\tau,\mathbf{N}\to T\tau$<br>$(\forall n,x:\mathbf{N},\tau.\tau'\to T'(\tau,\tau',f(n,x))),\tau'(x)\to T'(\tau,\tau',\mathsf{for}^c(f,x,n))$ |
| | $\mathsf{for}^c(f,x,0) = [x]_T$<br>$\mathsf{for}^r(f,x,0,f',x') = x'$<br>$\mathsf{for}^c(f,x,sn) = \mathrm{let}\ y\Leftarrow\mathsf{for}^c(f,x,n)\ \mathrm{in}\ f(n,y)$<br>$\mathsf{for}^r(f,x,sn,f',x') = \mathrm{let}'\ y,y'\Leftarrow\mathsf{for}^c(f,x,n),\mathsf{for}^r(f,x,n,f',x')\ \mathrm{in}\ f'(n,y,y')$ |
| $\mathsf{case}^c$<br>$\mathsf{case}^r$ | $f_1,f_2,in_i(x):(\tau_1\to T\tau),(\tau_2\to T\tau),\tau_1+\tau_2\to T\tau$<br>$(\forall y:\tau_1.\tau_1'\to T'(\tau,\tau',f_1y)),(\forall y:\tau_2.\tau_2'\to T'(\tau,\tau',f_2y)),\tau_i'(x)\to$<br>$\to T'(\tau,\tau',\mathsf{case}^c(f_1,f_2,x))$ |
| | $\mathsf{case}^c(f_0,f_1,in_i(x))=f_i(x)\qquad(i=0,1)$<br>$\mathsf{case}^r(f_0,f_1,in_i(x),f_1',f_2',x')=f_i'(x,x')\qquad(i=0,1)$ |
| $\mathsf{map}^c$<br>$\mathsf{map}^r$ | $f,l:(\tau_1\to T\tau_2),L\tau_1\to T(L\tau_2)$<br>$\forall x:\tau_1.\tau_1'\to T'(\tau_2,\tau_2',fx),L'(\tau_1,\tau_1',l)\to T'([x:L\tau_2]L'(\tau_2,\tau_2',x),\mathsf{map}^c(f,l))$ |
| | $\mathsf{map}^c(f,nil) = [nil]_T$<br>$\mathsf{map}^r(f,nil,f',[]) = []$<br>$\mathsf{map}^c(f,h::t) = \mathrm{let}\ y\Leftarrow f(h)\ \mathrm{in}\ \mathrm{let}\ z\Leftarrow\mathsf{map}^c(f,t)\ \mathrm{in}\ [y::z]_T$<br>$\mathsf{map}^r(f,h::t,f',[h',t']) = \mathrm{let}'\ y,y'\Leftarrow f(h),f'(h,h')\ \mathrm{in}$<br>$\qquad\mathrm{let}'\ z,z'\Leftarrow\mathsf{map}^c(f,t),\mathsf{map}^r(f,t,f',t')\ \mathrm{in}\ [y',z']$ |
| $\mathsf{length}^c$<br>$\mathsf{length}^r$ | $l:L\tau\to\mathbf{N}$<br>$L'(\tau,\tau',l)\to 1$ |
| | $\mathsf{length}^c(nil) = 0$<br>$\mathsf{length}^c(h::t) = s(\mathsf{length}^c(t))$ |
| $\mathsf{get}^c$<br>$\mathsf{get}^r$ | $l:L\tau\to T\tau$<br>$L'(\tau,\tau',l)\to T'(\tau,\tau',\mathsf{get}^c(l))$ |
| | $\mathsf{get}^c(nil) = \mathsf{err}^c$<br>$\mathsf{get}^r(nil,[]) = \mathsf{err}^r$<br>$\mathsf{get}^c(h::t) = \mathsf{case}^c([h]_T,\mathsf{err}^c,\mathrm{eq}^c(0,\mathsf{length}^c(t)))$<br>$\mathsf{get}^r(h::t,[h',t']) = \mathsf{case}^r([h]_T,\mathsf{err}^c,\mathrm{eq}^c(0,\mathsf{length}^c(t)),h',\mathsf{err}^r,*)$<br>where $\mathrm{eq}^c:\mathbf{N},\mathbf{N}\to 1+1$ is equality for the NNO |

Figure 4.2: Mixed translation of $\mathcal{NSC}$ term-constructors

| | | | |
|---|---|---|---|
| $\mathsf{zip}^c$ | $l_1, l_2 : L\tau_1, L\tau_2 \to T(L(\tau_1 \times \tau_2))$ | | |
| $\mathsf{zip}^r$ | $L'(\tau_1, \tau_1', l_1), L'(\tau_2, \tau_2', l_2) \to T'([x : L(\tau_1 \times \tau_2)]L'(\tau_1 \times \tau_2, \tau_1' \times \tau_2', x), \mathsf{zip}^c(l_1, l_2))$ | | |
| | $\mathsf{zip}^c(nil, nil)$ | $=$ | $[nil]_T$ |
| | $\mathsf{zip}^r(nil, nil, [], [])$ | $=$ | $[]$ |
| | $\mathsf{zip}^c(nil, h_2 :: t_2)$ | $=$ | $\mathsf{err}^c$ |
| | $\mathsf{zip}^r(nil, h_2 :: t_2, \_, \_)$ | $=$ | $\mathsf{err}^r$ |
| | $\mathsf{zip}^c(h_1 :: t_1, nil)$ | $=$ | $\mathsf{err}^c$ |
| | $\mathsf{zip}^r(h_1 :: t_1, nil, \_, \_)$ | $=$ | $\mathsf{err}^r$ |
| | $\mathsf{zip}^c(h_1 :: t_1, h_2 :: t_2)$ | $=$ | $\mathrm{let}\, y \Leftarrow \mathsf{zip}^c(t_1, t_2) \,\mathrm{in}\, [\langle h_1, h_2 \rangle :: y]_T$ |
| | $\mathsf{zip}^r(h_1 :: t_1, h_2 :: t_2, [h_1', t_1'], [h_2', t_2'])$ | $=$ | $\mathrm{let}'\, y, y' \Leftarrow \mathsf{zip}^c(t_1, t_2), \mathsf{zip}^r(t_1, t_2, t_1', t_2') \,\mathrm{in}$ $[\langle h_1', h_2' \rangle, y']$ |
| $\mathsf{split}^c$ | $l, k : L\tau, L\mathbf{N} \to T(L(L\tau))$ | | |
| $\mathsf{split}^r$ | $L'(\tau, \tau', l) \to T'([x : L(L\tau)]L'([y : L\tau]L'(\tau, \tau', y), x), \mathsf{split}^c(l, k))$ | | |
| | $\mathsf{split}^c(nil, nil)$ | $=$ | $[nil]_T$ |
| | $\mathsf{split}^r(nil, nil, [])$ | $=$ | $[]$ |
| | $\mathsf{split}^c(h :: t, nil)$ | $=$ | $\mathsf{err}^c$ |
| | $\mathsf{split}^r(h :: t, nil, [h', t'])$ | $=$ | $\mathsf{err}^r$ |
| | $\mathsf{split}^c(x, 0 :: p)$ | $=$ | $\mathrm{let}\, y \Leftarrow \mathsf{split}^c(x, p) \,\mathrm{in}\, [nil :: y]_T$ |
| | $\mathsf{split}^r(x, 0 :: p, x')$ | $=$ | $\mathrm{let}'\, y, y' \Leftarrow \mathsf{split}^c(x, p), \mathsf{split}^r(x, p, x') \,\mathrm{in}\, [[], y']$ |
| | $\mathsf{split}^c(nil, sn :: p)$ | $=$ | $\mathsf{err}^c$ |
| | $\mathsf{split}^r(nil, sn :: p, [])$ | $=$ | $\mathsf{err}^r$ |
| | $\mathsf{split}^c(h :: t, sn :: p)$ | $=$ | $\mathrm{let}\, y \Leftarrow \mathsf{split}^c(t, n :: p) \,\mathrm{in}\, cons'^c(h, y)$ |
| | $\mathsf{split}^r(h :: t, sn :: p, [h', t'])$ | $=$ | $\mathrm{let}'\, y, y' \Leftarrow \mathsf{split}^c(t, n :: p), \mathsf{split}^r(t, n :: p, t') \,\mathrm{in}$ $cons'^r(h, y, h', y')$ |
| | where $cons'^c(x, nil) = \mathsf{err}^c \mid cons'^c(x, h :: t) = [(x :: h) :: t]_T$ | | |
| | where $cons'^r(x, nil, x', []) = \mathsf{err}^r \mid cons'^r(x, h :: t, x', [h', t']) = [[x', h'], t']$ | | |

Figure 4.3: Mixed translation of $\mathcal{NSC}$ term-constructors (cont.)

| $c$ | arity and ML-like definition of $c^c(\overline{f}, \overline{x})$ and $c^r(\overline{f}, \overline{f}', \overline{x}, \overline{x}')$ |
|---|---|
| $\mathsf{map}^c$ | $f, l : (\tau_1 \to T\tau_2), V\tau_1 \to T(V\tau_2)$ |
| $\mathsf{map}^r$ | $\forall x : \tau_1.\tau_1' \to T'(\tau_2, \tau_2', fx), V'(\tau_1, \tau_1', l) \to T'([x : V\tau_2]V'(\tau_2, \tau_2', x), \mathsf{map}^c(f, l))$ |
| | $\mathsf{map}^c(f, \langle n, e\rangle) \quad = \quad \mathrm{let}\, y \Leftarrow f(e)\, \mathrm{in}\, [\langle n, y\rangle]_T$ |
| | $\mathsf{map}^r(f, \langle n, e\rangle, f', u) \quad = \quad \mathrm{let}'\, y, y' \Leftarrow f(e), \_\, \mathrm{in}\, f' \circ u$ |
| $\mathsf{length}^c$ | $l : L\tau \to \mathbf{N}$ |
| $\mathsf{length}^r$ | $L'(\tau, \tau', l) \to 1$ |
| | $\mathsf{length}^c(\langle n, x\rangle) \quad = \quad n$ |
| $\mathsf{get}^c$ | $l : V\tau \to T\tau$ |
| $\mathsf{get}^r$ | $V'(\tau, \tau', l) \to T'(\tau, \tau', \mathsf{get}^c(l))$ |
| | $\mathsf{get}^c(\langle n, e\rangle) \quad = \quad \mathsf{case}^c([e]_T, \mathsf{err}^c, \mathsf{eq}^c(1, n))$ |
| | $\mathsf{get}^r(\langle n, e\rangle, u') \quad = \quad \mathsf{case}^r([e]_T, \mathsf{err}^c, \mathsf{eq}^c(1, n), u', \mathsf{err}^r, *)$ |
| | where $\mathsf{eq}^c : \mathbf{N}, \mathbf{N} \to 1 + 1$ is equality for the NNO |

Figure 4.4: Mixed translation of $\mathcal{NSC}$ term-constructors - ARRAYS

| | |
|---|---|
| $\mathsf{zip}^c$ | $l_1, l_2 : L\tau_1, L\tau_2 \to T(L(\tau_1 \times \tau_2))$ |
| $\mathsf{zip}^r$ | $L'(\tau_1, \tau_1', l_1), L'(\tau_2, \tau_2', l_2) \to T'([x : L(\tau_1 \times \tau_2)]L'(\tau_1 \times \tau_2, \tau_1' \times \tau_2', x), \mathsf{zip}^c(l_1, l_2))$ |
| | $\mathsf{zip}^c(\langle m, x\rangle, \langle n, y\rangle) \quad = \quad \mathsf{case}^c([\langle x, y\rangle]_T, \mathsf{err}^c, \mathsf{eq}^c(m, n))$ |
| | $\mathsf{zip}^r(\langle m, x\rangle, \langle n, y\rangle, g, h) \quad = \quad \mathsf{case}^r([\langle x, y\rangle]_T, \mathsf{err}^c, \mathsf{eq}^c(m, n), \lambda i : n.\langle gi, hi\rangle, \mathsf{err}^r, *)$ |
| $\mathsf{split}^c$ | $l, k : L\tau, L\mathbf{N} \to T(L(L\tau))$ |
| $\mathsf{split}^r$ | $L'(\tau, \tau', l) \to T'([x : L(L\tau)]L'([y : L\tau]L'(\tau, \tau', y), x), \mathsf{split}^c(l, k))$ |
| | $\mathsf{split}^c(\langle l, u\rangle, \langle m, n\rangle) \quad = \quad \mathsf{case}^c([\langle m, \langle n, u\rangle\rangle]_T, \mathsf{err}^c, \mathsf{eq}^c(l, m * n))$ |
| | $\mathsf{split}^r(\langle l, u\rangle, \langle m, n\rangle, g, \_) \quad =$ |
| | $\quad \mathsf{case}^c([\langle m, \langle n, u\rangle\rangle]_T, \mathsf{err}^c, \mathsf{eq}^c(l, m * n), \lambda i : m, j : n.g(i * (j + 1)), \mathsf{err}^r, *)$ |

Figure 4.5: Mixed translation of $\mathcal{NSC}$ term-constructors ARRAYS (cont.)

- $L^r(A)$ stands for $\exists n : \mathbf{N}.n \Rightarrow A$ (the list object of sort $r$)

When there is no ambiguity with sorts the superscript $r$ is omitted.

**ML-style notation for function definitions.**

- $f(\langle x, y\rangle) = M(x, y)$ stands for $f(z) = R^{\Sigma}([x : A, y : B]M, z)$ or $R^{\exists}([x : A, y : B]M, z)$ or $M(\pi_0(z), \pi_1(z))$ depending on the domain of $f$
- $f(in_i(x)) = M_i(x) \quad (i = 0, 1)$ stands for $f(z) = R^{+}([x : A_0]M_0, [x : A_1]M_1, z)$
- $f(*) = M$ stands for $f(z) = R^1(M, z)$
- $\left\{\begin{array}{lcl} f(0) & = & M_0 \\ f(sn) & = & M_s(n, f(n)) \end{array}\right.$ stands for $f(z) = R^{\mathbf{N}}(M_0, [n : \mathbf{N}, x : A(n)]M_s, z)$
- $(M_0, [x : n]M_s)$ stands for $f(z : sn) = R^s(M_0, [x : n]M_s, z)$ and $()$ stands for $R^0$
- $f([]) = M$ stands for $f(z) = R^{V_0}(M, z)$

  $f([x, v]) = M$ stands for $f(z) = R^{V_s}([x : A, v : V(n, A)]M, z)$ when $n$ is clear from the context.
- array selection $select : \Pi n : \mathbf{N}.V(n, A) \to n \to A$ is given by

  $$\begin{array}{lcl} select(0, []) & = & () \\ select(sn, [a, v]) & = & (a, select(n, v)) \end{array}$$

  and we write $MN$ for $select(n, M, N)$ when $M : V(n, A) : c$ and $N : n : c$.

**Auxiliary notation for derived universal objects.**

- NNO $\mathbf{N}^r$ of sort $r$

| | |
|---|---|
| $0^r$ | $\mathbf{N}^r$ |
| | $0^r = \langle 0, *\rangle$ |
| $s^r$ | $\mathbf{N}^r \to \mathbf{N}^r$ |
| | $s^r(\langle n, *\rangle) = \langle sn, *\rangle$ |
| $R^{\mathbf{N}^r}$ | $A, (\mathbf{N}^r, A \to A), \mathbf{N}^r \to A$ |
| | $R^{\mathbf{N}^r}(x, f, \langle 0, *\rangle) = x$ <br> $R^{\mathbf{N}^r}(x, f, \langle sn, *\rangle) = f(n, R^{\mathbf{N}^r}(x, f, \langle n, *\rangle))$ |

- list object $L(A)$ of sort $c$

| | |
|---|---|
| $nil$ | $L(A)$ |
| | $0 = \langle 0, [] \rangle$ |
| $cons$ | $A, L(A) \to L(A)$ |
| | $cons(a, \langle n, v \rangle) = \langle sn, [a, v] \rangle$ |
| $R^L$ | $B([]), (l : L(A), a : A, B(l) \to B([a, l])), l : L(A) \to B(l)$ |
| | $R^L(b, f, \langle 0, [] \rangle) = b$ <br> $R^L(b, f, \langle sn, [a, v] \rangle) = f(\langle n, v \rangle, a, R^L(b, f, \langle n, v \rangle))$ |

we may write $M :: N$ for $cons(M, N)$

- list object $L^r(A)$ of sort $r$

| | |
|---|---|
| $nil^r$ | $L^r(A)$ |
| | $0^r = \langle 0, () \rangle$ |
| $cons^r$ | $A, L^r(A) \to L^r(A)$ |
| | $cons^r(a, \langle n, l \rangle) = \langle sn, (a, l) \rangle$ |
| $R^{L^r}$ | $B, (L^r(A), A, B \to B), L^r(A) \to B$ |
| | $R^{L^r}(b, f, \langle 0, \_ \rangle) = b$ <br> $R^{L^r}(b, f, \langle sn, l \rangle) = f(\langle n, l_s \rangle, l_0, R^{L^r}(b, f, \langle n, l_s \rangle))$ <br> where $l_0 = l(0)$ and $l_s = \Lambda i : n.l(si)$ |

we may write $M :: N$ for $cons^r(M, N)$

ML-style notation for function definitions will be used also for these derived types. When there is no ambiguity, we may drop the superscript $r$.

## 4.5 Summary

In this chapter we have:

1. shown intepretations of $\mathcal{NSC}$ and VEC in $\mathbf{B}$ (simple interpretation), $\mathbf{Fam}(\mathbf{B})$ (mixed interpretation) and $\mathbf{Set}$ (compile interpretation);
2. related these interpretations, obtaining shape error detection at compile-time (from a semantic point of view);
3. defined two translations of $\mathcal{NSC}$ and VEC into $\mathbf{S2}$, the simple one and the mixed one;
4. considered a model of $\mathbf{S2}$ given by $\pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}$ and seen that the interpretations in this model of the simple and mixed translations correspond to the simple

and mixed interpretations; since such a corrispondence, we can move the result from the semantic level to the syntactic one

Graphically:

$$
\begin{tikzcd}
\mathcal{NSC} \arrow[r, shift left, "\text{translations}"] \arrow[r, shift right] \arrow[d, shift right, "\text{intepretations}"'] \arrow[d, shift left] & \mathbf{S2} \arrow[d, "\text{intepretation}"] \\
\mathbf{B}, \mathbf{Fam}(\mathbf{B}) \arrow[r, dash, "\text{correspondence}"] & \pi : \mathbf{Fam}(\mathbf{B}) \to \mathbf{Set}
\end{tikzcd}
$$

# Part II

# Polymorphism

# Chapter 5

# Functor-based polymorphism

The interplay between type theory, programming language semantics and category theory is now well established. Two of the strongest examples of this interaction are the representation of function types as exponential objects in a cartesian closed category [LS86] and the description of polymorphic terms as natural transformations (e.g. [BFSS90]). For example, the operation of appending lists can be represented as a natural transformation $L \times L \Rightarrow L$ where $L : \mathcal{D} \to \mathcal{D}$ is the list functor on some category $\mathcal{D}$. Of course, these natural transformations must have associated functors for their domain and codomain. System F supports a notion of *expressible functor*, i.e. a type constructor and a corresponding action on functions [RP90], but such encodings are rather unsatisfactory ([GLT89, Section 15.1.1]). In particular, the action of a functor on morphisms, its *mapping*, must be defined anew for each choice of type constructor.

A better approach, primarily advocated by adherents of the Bird-Meertens style (e.g. [MFP91, MH95, Jeu95]), is to give a combinator for mapping, whose type can be expressed as:

$$\text{map} : \forall F : 1.\forall X, Y.(X \to Y) \to FX \to FY.$$

That is, for any functor $F : 1$ (i.e. taking one argument), types $X$ and $Y$, and any morphism $f : X \to Y$ we have

$$\text{map } F\ X\ Y\ f : FX \to FY$$

whose action is to take each datum of type $X$ in $FX$ and apply $f$ to it. Unfortunately, the existence of this type does not solve the problem of realizing this high-level algorithm, as there remains the question of how to *find* the data.

Naturally, one can use the functor to determine the algorithm. There are basically two ways to do this. One method is to have the user specify the mapping algorithm, say by instantiating a *constructor class* [Jon95]. In this particular case, the functor and type arguments are suppressed to obtain map $f\ a$, since the choice of functor can be inferred from the type of $a$. Unfortunately, it follows that if $F$ and $G$ are functors such that $FX$ and $GY$ are intended to be the same for some types $X$ and $Y$ then a dummy constructor must be introduced to distinguish them.

The other method is to generate the mapping algorithm automatically, from the structure of the functor. This results in a small loss of flexibility, but saves the user from supplying repetitious algorithms. Charity [CF92] encodes this directly. Jeuring [Jeu95] uses a pre-processor to determine the appropriate Haskell code for mapping and other *polytypic* operations. *Intensional polymorphism* [HM95] is a general technique for describing type-dependent operations in an extension of ML, designed to obtain more efficient compilation. Although in the same spirit as the other approaches, the lack of sum types makes it hard to make direct comparisons.

Perhaps surprisingly, there is a generous class of covariant functors for which it is possible to describe a mapping algorithm that is independent of the choice of functor, i.e. which support *parametric functorial polymorphism.* The first such algorithms (for polymorphic folding) were produced for a small experimental language **P2** [Jay95a]. Polymorphic mapping for the *covariant* type system is developed in [Jay97]. This paper presents an extension of Hindley-Milner called Functorial ML, or FML, which supports parametric functorial polymorphism.

For example, it supports:

$$\begin{aligned} \mathrm{map}\ f\ (\mathrm{cons}\ h\ t) &\rightarrow_* \mathrm{cons}\ (f\ h)\ (\mathrm{map}\ f\ t) \\ \mathrm{map}\ f\ (\mathrm{leaf}\ x) &\rightarrow_* \mathrm{leaf}\ (f\ x) \end{aligned}$$

where cons is the usual list constructor, and leaf is the leaf constructor for binary trees with labeled leaves. It is important to note that these evaluations are not achieved by pattern matching on primitive constructors of user defined data types, such as cons and leaf, but by hard-wired reductions on a fixed set of language primitives, which is used in the reduction to find the data in a uniform way. We can also use mapping within a let-construct:

$$\mathrm{let}\ g = \mathrm{map}\ f\ \mathrm{in}\ \langle g\ (\mathrm{cons}\ h\ t), g\ (\mathrm{leaf}\ x)\rangle$$

where $\langle -, - \rangle$ is the pairing for binary products. The polymorphic mapping allows us to support polymorphic folding, too, as will be shown in the body of the paper. Functors of many variables are also catered for. For example, we have

$$\mathrm{map}_2\ f\ g\ (\mathrm{in}_0\ t) \rightarrow \mathrm{in}_0\ (f\ t) \tag{5.1}$$

$$\mathrm{map}_2\ f\ g\ (\mathrm{pair}\ s\ t) \rightarrow \mathrm{pair}\ (f\ s)\ (g\ t). \tag{5.2}$$

where $\mathrm{in}_0$ is the left inclusion to a binary sum, Thus, $\mathrm{map}_2\ f\ g$ is equally able to act on values whose type is a sum or product, etc.

Shape theory [Jay95b] provides the basis for these algorithms. It is a new approach to data types based on the idea of decomposing values into their shape (or data *structure*) and the data which is stored within them. The data structure corresponds to the type constructor, or functor, whose argument is the type of the data. Thus *shape polymorphism* is closely linked to functorial polymorphism, as distinct from the *data polymorphism* of operations like append. Thus, the data-shape decomposition supports uniform mechanisms for storing data within a shape, which are exploited by our mapping algorithm.

To see how this works, consider the projection functors $\Pi_i^m : \mathcal{D}^m \to \mathcal{D}$ which pick out the $i$th argument from $m$. It is tempting to identify $\Pi_0^2(X, X)$ and $\Pi_1^2(X, X)$ with $X$ but this would obliterate the shape-data distinction. Rather, these types are "isomorphic", a situation captured by terms

$$\mathrm{tag}_{2,j} : X_j \to \Pi_j^2(X_0, X_1)$$

(for $j \in 2$) and their inverses. Now given $x : X$ we have the reductions:

$$\mathrm{map}_2\ f_0\ f_1\ (\mathrm{tag}_{2,j}\ x) \to \mathrm{tag}_{2,j}\ (f_j\ x).$$

In other words, the isomorphisms are used to determine where to find the data associated to each argument of the functor. Similarly, we have isomorphisms to disambiguate composite functors, e.g.

$$\mathrm{pack}_{1,1} : F(G(X)) \to F\langle G\rangle^1(X).$$

Its source is the functor $F$ applied to $G(X)$ while its target is the composite functor $F\langle G\rangle^1$ applied to $X$. The corresponding reduction for map is:

$$\mathrm{map}\ f\ (\mathrm{pack}_{1,1}\ t) \to \mathrm{pack}_{1,1}\ (\mathrm{map}\ (\mathrm{map}\ f)\ t)$$

whose corresponding diagram is:

$$\begin{tikzcd}
F(G(X)) \arrow[r, "\mathrm{pack}_{1,1}"] \arrow[d, "\mathrm{map}\ (\mathrm{map}\ f)"'] & F\langle G\rangle^1(X) \arrow[d, "\mathrm{map}\ f"] \\
F(G(Y)) \arrow[r, "\mathrm{pack}_{1,1}"'] & F\langle G\rangle^1(Y).
\end{tikzcd}$$

These isomorphisms may be viewed as a systematic method for resolving the ambiguities otherwise addressed by the dummy constructors mentioned above. The other approaches use implicit substitution to handle functor composition, making a uniform algorithm impossible.

The significance of these isomorphisms becomes particularly clear in certain application contexts. For instance, in distributed or parallel computing the shape-data distinction can be used to describe data distributions [Jay95c] in which case the isomorphisms represent redistributions of the data. Also, such isomorphisms between different composites are central to bicategories [Ben67].

The polymorphism of the mapping algorithm can be captured in a system that supports inference of functors as well as types. We work with an extension of the Hindley-Milner types which supports a syntactic class of functors, as well as those of types and type schema. Another possibility is to identify types and type schema (so extending system F with functors). Such a system is used in the proof of strong normalization. Again, one could identify types with functors of arity 0, at the cost of introducing another pair of isomorphisms. We emphasize the tri-partite division for reasons of clarity, and to obtain type inference.

In this paper we will focus on FML with untyped terms, i.e. à la Curry (in the terminology of [Bar92]), to emphasize parametric functorial polymorphism. However, FML à la Church is very important, too: it allows a more aggressive use of type information (as advocated in [HM95]) and it is more suitable for a semantic investigation.

The chapter is structured as follows. Section 5.1 introduces the functors, their syntax and semantics. Section 5.2 introduces the basic term structure and proves elementary properties, relative to a set of term constants. Section 5.3 considers a variety of functor-polymorphic constants motivated by the desired choice of functor semantics. It also introduces some syntactic sugar for monads. This is used in Section 5.4 where shape polymorphic programs for matching and unification are constructed.

Section 5.5 looks at extensions to the basic type system. As well as considering quantification by functor arities, it considers recursion over functor construction, which leads to polytypism. This can be used to define the various constants introduced earlier, provided that the semantics of functors is suitably constrained. Section 5.6 contrasts FML with **PolyP**, a polytypic language currently under development [JJ97]. Section 5.7 concludes the body of the chapter.

The Appendix B contains a proof of strong normalization for FML obtained by establishing the property for a related language $FF_{shape}$ which supports all the combinators previously introduced, but is typed à la Church.

## 5.1 Functors, types and type schema

### 5.1.1 Functors

Only elementary concepts of category theory will be required in this paper, as found in almost any reference work, e.g. [Mac71, BW90]. Consider a fixed category, $\mathcal{D}$. A *(covariant) functor* is a structure-preserving morphism of categories, i.e. it maps objects to objects and morphisms to morphisms, so as to preserve the sources and targets of the morphisms, the composition and identities. The identity arrow on an object $A$ may be written $\mathrm{id}_A$. The symbols $m$ and $n$ will denote natural numbers throughout this paper. The category $\mathcal{D}^m$ has objects given by $m$-tuples of objects of $\mathcal{D}$ and arrows given $m$-tuples of arrows of $\mathcal{D}$ with the obvious point-wise composition and identities. A functor $F : \mathcal{D}^m \to \mathcal{D}$ is said to be a *functor on* $\mathcal{D}$ *of arity* $m$ which may be written as $F : m$ (as $\mathcal{D}$ is understood from the context). The action of such a functor on a sequence of arrows $f_i : X_i \to Y_i$ for $i \in m$ will be written as

$$\mathrm{map}_m\ f_0\ \ldots f_{m-1} : F(X_0, \ldots, X_{m-1}) \to F(Y_0, \ldots Y_{m-1})\ .$$

Here are some elementary examples and constructions. The binary functors $+, \times : 2$ represent sums and products, and $1 : 0$ is the functor of no arguments that produces the terminal object, corresponding to the unit type. Let $i$ range over $m = \{0, \ldots, m-1\}$, also written $i : m$. The $i$th *projection functor* of $m$ arguments is $\Pi_i^m : m$. A sequence $G_i : n$ of $m$ functors, all of arity $n$, may be written as $G_{i:m}$ or even $\overline{G}$ when the choice of $m$ is either clear from the context, or irrelevant. Similar notation will be used for other sequences below, of types, etc. If $F : m$ then

$$\mathcal{D}^n \xrightarrow{\langle\overline{G}\rangle} \mathcal{D}^m \xrightarrow{F} \mathcal{D}$$

written $F\langle\overline{G}\rangle$ is their *composite*. If $F : m+1$ is a functor then $\mu F : m$ is its *initial algebra functor* which represents the functor whose action on the tuple $\overline{X}$ yields the initial $F(\overline{X}, -)$-algebra with action

$$\mathrm{wrap}_m : F(\overline{X}, \mu F(\overline{X})) \to \mu F(\overline{X})$$

which gives a minimal solution of the domain equation $F(\overline{X}, Y) \cong Y$.

Functoriality is fundamental to the interpretation of the algebras whose initiality we are considering. For example, given $F : 1$ and an algebra $f : F(Y) \to Y$ the algebra homomorphism $(\mathrm{fold}_0\ f) : \mu F() \to Y$ makes the following square commute:

$$\begin{tikzcd} F(\mu F()) \arrow[r, "\mathrm{wrap}_0"] \arrow[d, "\mathrm{map}_1\ (\mathrm{fold}_0\ f)"'] & \mu F() \arrow[d, "\mathrm{fold}_0\ f"] \\ F(Y) \arrow[r, "f"'] & Y \end{tikzcd}$$

This is the basis for the corresponding term reduction

$$\mathrm{fold}_0\ f\ (\mathrm{wrap}_0\ t) > f\ (\mathrm{map}_1(\mathrm{fold}_0\ f)t)$$

which shows that $\mathrm{fold}_0\ f$ acts by recursively mapping itself across all of the substructures of $\mathrm{wrap}_0\ t$ and then applying $f$ to the result.

The usual presentations of $\mathrm{fold}_0$ expand this definition for each choice of functor, typically by a case analysis on the functor constructors.

Now consider initial algebras for $F : 2$. Each $f : X \to Y$ determines an algebra

$$\mathrm{wrap}_1 \circ (\mathrm{map}_2\ f\ \mathrm{id}) : F(X, \mu F(Y)) \to \mu F(Y)$$

whose corresponding homomorphism $\mu F(X) \to \mu F(Y)$ defines the functoriality of $\mu F$. Thus, the mapping on $F : 2$ defines the initial algebra structure on $\mu F(X)$ which in turn supports the functoriality of $\mu F$. Thus, despite the delicate interplay between mapping and folding, it is clear that mapping, i.e. functoriality, is the the fundamental concept, and further, that it cannot be captured by the purely type-based notions of introduction and elimination rules. It also emphasizes the need to consider all functors together, rather than separating them according into arities, as occurred in the earlier work of one of us [Jay95a] or in the study of polytypism, e.g. [JJ97].

The class of functors obtained by closing up under the constructions above have been widely studied in the context of initial algebra theory, starting with [GTWW77] and more recently as the *regular functors* of [Mee96].

### 5.1.2 Syntax

These constructions motivate the choice of functors in the following description of the raw syntax for *functors* $F$, *types* $\tau$ and *type schema* $\sigma$ in FML:

$$\begin{array}{rcl} F, G & ::= & X \mid C \mid \Pi_i^m \mid F\langle \overline{G} \rangle^n \mid \mu F \\ \tau & ::= & X \mid F(\overline{\tau}) \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall X : \mathrm{T}.\sigma \mid \forall X : m.\sigma. \end{array}$$

In addition to the constructions $\Pi_i^m$,$F\langle\overline{G}\rangle^n$, and $\mu F$, described in Section 5.1.1, functors include variables $X$ and constant functors $C$. Types are variables, functor applications, and function types. The application of a functor to a tuple $\tau_{i:m}$ of types represents the action of the categorical functor on objects. Note that the usual ancillary type constructors, such as products and sums, have been treated as functors. The key exception is the function type constructor, which is contravariant in its first argument.

Type schema include the types, and are closed under universal quantification over type variables and over functors of given arity. The former quantification is familiar from Hindley-Milner and is used to express data polymorphism; the latter quantification will allow us to express functorial polymorphism.

**Notation 5.1** We adopt the following notational conventions throughout the paper. $X$ and $Y$ range over functor and type variables. $C$ ranges over functor constants, in particular $+, \times$ and 1. The symbols $F$ and $G$ range over functors (though sometimes are used as functor variables), $\tau$ ranges over types, and $\sigma$ ranges over type schema. A type $\tau$ may be distinguished from functors or type schema by giving it the fixed arity $\tau : \mathrm{T}$. We write $\tau_1 \times \tau_2$ for $\times(\tau_1, \tau_2)$ and $\tau_1 + \tau_2$ for $+(\tau_1, \tau_2)$ and 1 for the type $1()$. Composition binds tighter than the initial algebra constructor $\mu$.

A *type context* $\Delta$ is a sequence of type variables with assigned arities (either $X : n$ or $X : \mathrm{T}$) with no repetition of variables. We may identify $\Delta$ with a partial function from functor and type variables to arities, and write $\mathrm{DV}(\Delta)$ for its domain. Also, each functor constant $C$ has an associated arity $n_C$ fixed upon its introduction.

For each of the syntactic categories above we define corresponding judgments for asserting well-formedness as follows:

- $\Delta \vdash$ means that $\Delta$ is a well-formed typed context.
- $\Delta \vdash F : n$ means that $F$ is a functor of arity $n$ in context $\Delta$.
- $\Delta \vdash \tau : \mathrm{T}$ means that $\tau$ is a type in context $\Delta$.
- $\Delta \vdash \sigma$ means that $\sigma$ is a type schema in context $\Delta$.

The corresponding inference rules are given in Figure 5.1 (where the symbol $i$ ranges over $m$). The formation rules for functors express the constraints on arities implicit in the category theory. In general, we may use the symbol $J$ to represent judgments.

The *free variables* of a functor are defined in the usual way, and the functors are defined to be equivalence classes of well-formed functor expressions under $\alpha$-conversion. Types and

**Contexts**

$$\text{empty}\ \frac{}{\emptyset \vdash} \qquad \text{functor}\ \frac{\Delta \vdash}{\Delta, X : n \vdash}\ X \notin \mathrm{DV}(\Delta)$$

$$\text{type}\ \frac{\Delta \vdash}{\Delta, X : \mathrm{T} \vdash}\ X \notin \mathrm{DV}(\Delta)$$

**Functors**

$$X\ \frac{\Delta \vdash}{\Delta \vdash X : m}\ m = \Delta(X) \qquad C\ \frac{\Delta \vdash}{\Delta \vdash C : n_C}$$

$$\text{FPi}\ \frac{\Delta \vdash}{\Delta \vdash \Pi_i^m : m} \qquad \text{Fcomp}\ \frac{\Delta \vdash F : m \quad \Delta \vdash G_0 : n\ \ldots\ \Delta \vdash G_{m-1} : n}{\Delta \vdash F\langle G_{i:m}\rangle^n : n}$$

$$\text{Fmu}\ \frac{\Delta \vdash F : m+1}{\Delta \vdash \mu F : m}$$

**Types**

$$X\ \frac{\Delta \vdash}{\Delta \vdash X : \mathrm{T}}\ \mathrm{T} = \Delta(X) \qquad \text{Fapp}\ \frac{\Delta \vdash F : m \quad \Delta \vdash \tau_0 : \mathrm{T}\ \ldots\ \Delta \vdash \tau_{m-1} : \mathrm{T}}{\Delta \vdash F(\tau_{i:m}) : \mathrm{T}}$$

$$\rightarrow\ \frac{\Delta \vdash \tau_1 : \mathrm{T} \quad \Delta \vdash \tau_2 : \mathrm{T}}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \mathrm{T}}$$

**Schema**

$$\tau\ \frac{\Delta \vdash \tau : \mathrm{T}}{\Delta \vdash \tau} \qquad \forall_m\ \frac{\Delta, Y : m \vdash [Y/X]\sigma}{\Delta \vdash \forall X : m.\sigma}\ Y \notin \mathrm{DV}(\Delta)$$

$$\forall\ \frac{\Delta, Y : \mathrm{T} \vdash [Y/X]\sigma}{\Delta \vdash \forall X : \mathrm{T}.\sigma}\ Y \notin \mathrm{DV}(\Delta)$$

Figure 5.1: FML types

schema are defined similarly. The quantification $\forall\Delta.\sigma$ of a type scheme $\sigma$ by a type context $\Delta$ is defined as follows:

$$\begin{array}{rcl} \forall.\,\sigma & = & \sigma \\ \forall(\Delta', X : n).\,\sigma & = & \forall\Delta'.(\forall X : n.\,\sigma) \\ \forall(\Delta', X : \mathrm{T}).\,\sigma & = & \forall\Delta'.(\forall X : \mathrm{T}.\,\sigma) \end{array}$$

**Lemma 5.2** *1. Uniqueness of derivation: each judgment $\Delta \vdash J$ has at most one derivation (up to $\alpha$-conversion).*

*2. Uniqueness of arity: if $\Delta \vdash F : n_j$ is derivable for $j \in 2$ then $n_0 = n_1$.*

**Proof.** For the first, use induction on the size of the derivation of $\Delta \vdash J$. For the second, use induction on the structure of $F$. ∎

### 5.1.3 Examples

Many of the usual data type constructors can be presented as functors. For example, consider a datatype definition

$$\mathrm{Maybe}_d\ \tau = ok\ \tau \mid fail.$$

We can regard this as specifying the type $\tau + 1$ or, more formally, $+(\tau, 1())$ i.e. the plus functor applied to the two types $\tau$ and $1()$. Note that 1 is a functor of arity 0, not a type, and so must be applied to 0 arguments to obtain a type. To express this as a functor of $\tau$ requires several abstractions. First, the type $\tau$ must be replaced by $\mathrm{Id}(\tau)$ where Id is the *identity functor* $\Pi^1_0$. Then, to respect arities, the type $1()$ must also be presented as a functor of arity 1 that ignores its argument, namely $1\langle\rangle^1(\tau)$. This analysis yields

$$+(\mathrm{Id}(\tau), 1\langle\rangle^1(\tau))$$

i.e. the functor $+$ applied to two types built from $\tau$. Finally, we can replace this by a functor of one variable, namely

$$+\langle \mathrm{Id}, 1\langle\rangle^1\rangle^1(\tau)$$

or

$$\mathrm{Maybe}_F = +\langle \mathrm{Id}, 1\langle\rangle^1\rangle^1.$$

Note that $\mathrm{Maybe}_F(\tau)$ and $\tau + 1$ are not identical types. They are obtained by applying functors of different arities, namely 1 and 2 respectively. This distinction is fundamental since the meaning of terms below will typically depend on the separation of the functor from its arguments.

The list functor shows the use of recursion in functor definition. This functor corresponds to the datatype definition $L_dX = nil \mid cons\ X \times L_dX$. We must construct the functor that maps types $X$ and $Y$ to $1 + X \times Y$. This is

$$+\langle 1\langle\rangle^2, \times\rangle^2$$

Now the list functor $L$ is obtained by constructing the corresponding initial algebra functor (which maps $X$ to $\mu Y.1 + X \times Y$)

$$L = \mu + \langle 1\langle\rangle^2, \times\rangle^2$$

Similarly, the datatype $T_dX = leaf\ X \mid node\ T_dX \times T_dX$ of binary trees with labeled leaves has corresponding functor

$$T = \mu + \langle \Pi^2_0, \times\langle \Pi^2_1, \Pi^2_1\rangle^2\rangle^2$$

and the datatype $R_dX = leaf\ X \mid node\ L_d(R_dX)$ of rose trees (having finite lists of branches) has corresponding functor

$$R = \mu + \langle \Pi^2_0, L\langle \Pi^2_1\rangle^2\rangle^2$$

### 5.1.4 Substitutions and unification

A *substitution* is a partial function $S$ from functor and type variables to expressions for functors and types, respectively. The notation $S : \Delta_1 \to \Delta_2$ means that the $\Delta_i$ are well-formed contexts, $\mathrm{DV}(\Delta_1)$ is included in the domain of $S$ and for each $X \in \mathrm{DV}(\Delta_1)$ we have

$$\Delta_2 \vdash SX : \Delta_1(X).$$

The action of such an $S$ extends homomorphically to any expression that is well-formed in context $\Delta_1$ (including those to be defined below). If $R : \Delta_2 \to \Delta_3$ is another substitution then their *composite* substitution $R\ S$ has action given by $(R\ S)X = R(SX)$.

$S$ is a *renaming* if it is an injective function from variables to variables. Then for each type context $\Delta$ we have another such $\Delta_S$ obtained by applying $S$ homomorphically to $\Delta$.

**Lemma 5.3** *1. Renaming: if $S$ is a renaming then $\Delta \vdash J$ implies $\Delta_S \vdash S(J)$.*

*2. Thinning: $\Delta_1, \Delta_2 \vdash J$ implies $\Delta_1, X : a, \Delta_2 \vdash J$ provided $X \notin \mathrm{DV}(\Delta_1, \Delta_2)$ and $a$ is either $T$ or an arity $m$.*

*3. Substitution: if $S : \Delta_1 \to \Delta_2$ is a substitution then $\Delta_1 \vdash J$ implies $\Delta_2 \vdash S(J)$.*

**Proof.** Each of the proofs is by induction on the derivation of the premise. ∎

Let $\Delta \vdash J_j : a$ be well-formed functors or types having the same arity $a$ for $j \in 2$. A *unifier* for $(\Delta, J_0, J_1)$ is a pair $(\Delta', S)$ such that $S : \Delta \to \Delta'$ is a substitution and $S(J_0) = S(J_1)$. Their *most general unifier* $\mathcal{U}(\Delta, J_0, J_1)$ is a unifier $(\Delta', S)$ such that if $(\Delta'', S')$ is any other unifier for them then there is a substitution $R : \Delta' \to \Delta''$ such that $S'(X) = R\ S(X)$ for all $X \in \mathrm{DV}(\Delta)$.

**Lemma 5.4** *If* $(\Delta, J_0, J_1)$ *has a unifier then it has a most general unifier.*

**Proof.** Standard. Note that the introduction of functors does not require higher order unification since, for example, $\Pi_i^m(\overline{X})$ and $X_i$ do not have a unifier. ∎

### 5.1.5 Semantics

A completely general semantic treatment would axiomatize the assumptions made about the underlying category, starting from, say, a *locos* [Coc90] as used in developing the foundations of shape theory [Jay95b]. A successful treatment should allow the results to be extended to various Kleisli categories built over the base, to represent computational monads [Mog91b, Wad93]. Another possibility, not explored here, is to interpret FML-functors as morphisms in a bicategory [Ben67]. However, one already gains significant insights from set-theoretic models. We will also introduce domain-theoretic models (i.e. types as $\omega$-cpos) to address some issues obscured in the set-theoretic treatment.

The simpler interpretations are *extensional.* The interpretation of functor application to types will be by applying (semantic) functors to objects. Thus they identify $\Pi_i^m(\overline{X})$ with $X_i$ and $F\langle\overline{G}\rangle^m(\overline{X})$ with $F(\overline{G(\overline{X})})$ so that the canonical isomorphisms become equalities. To interpret the quantification occurring in type schema one should, say, use a set theory with universes, and interpret types as small sets. However, we will focus only on the novel part of the semantics, i.e. the interpretation FML functors, and so may safely ignore size issues.

Other, *intensional* models are more cumbersome, but able to represent the isomorphisms explicitly, so that functor composition is associative only up to isomorphism. These intensional interpretations motivate the introduction of FML constants corresponding to mediating isomorphisms.

Having fixed the choice of category, and thus the interpretation of types, the basic question is then: which functors? It is relatively easy to identify classes of functors closed under composition, so that the key semantic point is that they be closed under the formation of initial algebras. Various alternatives will be given.

#### 5.1.5.1 Set-theoretic semantics

Let **Set** be the category of (small) sets and functions. The obvious interpretation of an FML functor of arity $m$ is as a functor $F : \mathbf{Set}^m \to \mathbf{Set}$. But the interpretation of $\mu F$ is problematic, since there are functors which do not have an initial algebra for cardinality reasons, e.g. the power set functor $P(X) = \{X' | X' \subseteq X\}$. One may avoid this problem by interpreting FML functors within a suitably restricted class of functors on **Set**. We consider three possible choices: $\omega$-colimit preserving functors (where "$\omega$-colimit" means "colimit of an $\omega$-chain"); functors shapely over lists; and, regular functors.

**$\omega$-colimit preserving functors.** In any cartesian closed category $\mathcal{D}$ with small coproducts and $\omega$-colimits the class of functors on $\mathcal{D}$ preserving $\omega$-colimits enjoys the following properties:

- closure under composition;
- if $F : m+1$ preserves $\omega$-colimits, then the the initial algebra functor $\mu F$ exists and preserves $\omega$-colimits;
- if $F_i : m$ preserves $\omega$-colimits for each $i \in I$, then their coproducts, given by the functor $F(\overline{X}) = \coprod_{i \in I} F_i(\overline{X})$ does too;
- constant functors, projection functors $\Pi_i^m$ and the binary product functor $\times$ preserve $\omega$-colimits.

Thus, the class of $\omega$-colimit preserving functors on **Set** is suitable for interpreting FML-functors. An example of $\omega$-colimit preserving functor (not definable in FML) is the finite power set functor $\mathcal{P}_f$, while non-examples include functors of the form $F(X) = X^B$ where $B$ is infinite.

**Functors shapely over lists.** A *functor shapely over lists* is a functor $F : \mathbf{Set}^m \to \mathbf{Set}$ equipped with a cartesian natural transformation

$$\delta_{\overline{X}} : F(X_i)_{i:m} \to \prod_{i:m} LX_i$$

i.e a natural transformation whose defining commuting squares are all pullbacks. Such a functor is determined, up to isomorphism, by its *object of shapes* $S = F(\overline{1})$ and its *arity* $E = \delta_{\overline{1}} : S \to N^m$ (where $N = L1$ is the natural numbers object) which specifies how much data of each type is required for each shape. An alternative definition of functor shapely over lists requires a cartesian natural transformation

$$\delta_{\overline{X}} : F(X_i)_{i:m} \to L(\coprod_{i:m} X_i)$$

The two definitions coincide when $m = 1$, and are *interchangable* for all purposes. In the rest of the paper we stick with the first definition.

In any locos, the class of functors shapely over lists is closed under the construction of initial algebra functors, and so is suitable for interpreting FML-functors [Jay95b]. Furthermore, functors shapely over lists preserve all pullbacks (since the list functor does), in particular the pullback

$$\begin{tikzcd} A \times B \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d] \\ B \arrow[r] & 1 \end{tikzcd}$$

that defines products. Thus one has an isomorphism called zip from the canonical pullback $FA \times_{\#} FB$ of the corresponding cospan into $F1$ to $F(A \times B)$. It takes a pair of values of the same shape and produces a shape filled with pairs. Therefore, this interpretation is useful to validate the extension of FML with shapely constants.

An example of functor shapely over lists is the matrix functor where $\mathrm{Matrix}(A)$ represents the matrices with entries of type $A$. Here $\delta$ produces a list of the entries, say, row by row. If we identify $\mathrm{Matrix}(1)$ with $N \times N$ then the arity function is simply multiplication. The list functor preserves all $\omega$-colimits. In **Set** such colimits are preserved by pulling back, so that all functors shapely over list have this property, too. That these functors form a proper subclass of the $\omega$-colimit preserving functors is illustrated by $\mathcal{P}_f$ whose object of shapes $\mathcal{P}_f(1) = 2$ is too small to represent all possible shapes of finite sets.

**Inductive functors.** The formal grammar for FML functor expressions (without functor variables) determines a class of functors, which we close under functor isomorphisms (to avoid making canonical choices) to obtain the *inductive functors*. The properties of this class have been studied by various authors interested in initial algebras, most recently by those investigating polytypism. Meertens [Mee96] dubbed them the *regular functors*, though: this name has been used for a slightly larger class elsewhere in the community [JJ96]; it may clash with its use in the study of functors on regular categories; and, it contradicts the convention that trees may be thought of as irregular arrays.

A simple inductive argument shows that all inductive functors are shapely over lists. The converse fails since there are only countably many inductive functors (up to isomorphism) whereas the functors shapely over lists are indexed by the uncountable collection of functions into (powers of) $N$. For example, the matrix functor does not appear to be inductive.

**Representations of functors shapely over lists.** Since coproducts in **Set** are stable under pullbacks, functors shapely over lists can be decomposed into dependent sums of products, with one summand per shape:

$$F(\overline{X}) = \coprod_{s:S}(\prod_{i:m} X_i^{E(s,i)})$$

which can be represented by the pair $[\![F]\!] = \langle S : \mathbf{Set}, E : S \to N^m\rangle$. This explicit separation of the shape $S$ from the data $\overline{X}$ provides a more intensional semantics of FML functors, which we now present in detail.

- $[\![\Pi_i^m]\!]$: $S = 1$ and $E(0,j) =$ (if $i = j$ then 1 else 0)
- $[\![F\langle\overline{G}\rangle^n]\!]$: $S = \coprod_{s:S_F}(\prod_{i:m} S_{G_i}^{E_F(s,i)})$ and $E\ (\langle s, f\rangle, j) = \coprod_{i:m}(\coprod_{e:E_F(s,i)} E_{G_i}(f\ i\ e, j))$

  where $[\![F]\!] = \langle S_F, E_F\rangle$ and $[\![G_i]\!] = \langle S_{G_i}, E_{G_i}\rangle$
- $\mu F$: $S = \mu S. \coprod_{s:S_F} S^{E_F(s,m)}$ and $E(\langle s, f\rangle, i) = E_F(s,i) + \coprod_{e:E_F(s,m)} E(f\ e, i)$

  where $[\![F]\!] = \langle S_F, E_F\rangle$. Here $S$ can be viewed as the set of finite trees with nodes labeled by elements of $S_F$ and branching determined by labels, while $E(\_, i)$ is a weight function (defined by induction on the structure of finite trees) such that a node labeled by $s$ contributes $E_F(s,i)$ to the weight of the whole tree.

This interpretation is useful to motivate the introduction of canonical isomorphisms in FML. In fact, in this interpretation $\Pi_i^m(\overline{X}) \neq X_i$ and $F\langle\overline{G}\rangle^m(\overline{X}) \neq F(\overline{G(\overline{X})})$, but they are canonically isomorphic.

There is an alternative representation of functors shapely over lists as formal power series

$$F(\overline{X}) = \coprod_{n:N^m} C(n) \times \prod_{i:m} X_i^{n(i)}$$

for some $C : N^m \to \mathbf{Set}$ that establishes a link to Joyal's theory of species [Joy81].

**Closed term model.** Another intensional semantics is to interpret FML functors in the algebra of closed functor expressions, which can be seen as syntactic representations of inductive functors. This algebra is simply the closed term algebra for the following many-sorted algebraic signature

- $\Omega_m$ sort, for each $m \in N$
- $1 : \Omega_0$
- $\times, + : \Omega_2$

- $\Pi_i^m : \Omega_m$, for each $m \in N$ and $i \in m$
- $\circ_{m,n} : \Omega_m, \Omega_n^m \to \Omega_n$, for each $m, n \in N$
- $\mu_m : \Omega_{m+1} \to \Omega_m$, for each $m \in N$.

Here $\circ$ is used to represent composition. This interpretation can be used to support case analysis (or primitive recursion) on functor expressions, as discussed in Section 5.6 below.

#### 5.1.5.2 Domain-theoretic semantics

Now let us consider semantics in the category **Cpo** of $\omega$-cpos and $\omega$-continuous functions. While many features are shared with the set-theoretic semantics, the domain-theoretic models expose some issues that are relevant for both the general semantics, and the implementation of FML. First, terms should be interpreted by continuous functions. In particular, the interpretation of the mapping combinator should be continuous. Therefore, the interpretation of an FML functor of arity $m$ should be a *strong functor* $F : \mathbf{Cpo}^m \to \mathbf{Cpo}$, i.e. the action of $F$ on morphisms must be *internalizable*, which in the case of **Cpo** means $\omega$-continuous. Note that in **Set** every functor is strong, while in **Cpo** there is at most one way of internalizing the action of a functor on morphisms (but this is not true in general).

Second, we *could* require that $F$ preserve decidable objects, i.e. objects equipped with an equality arrow $\mathrm{eq} : X \times X \to 1 + 1$ making the following diagram a pullback:

$$\begin{tikzcd} X \times X \arrow[r, "\langle \mathrm{id}, \mathrm{id} \rangle"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & X \times X \arrow[d, dashed, "eq"] \\ 1 \arrow[r, "true"'] & 1 + 1. \end{tikzcd}$$

In particular, this implies that $F(\overline{1})$ is decidable, so that equality of shapes can be tested. In **Set** every object has decidable equality, while in **Cpo** the objects with decidable equality are exactly the flat cpos (i.e. those cpos where the partial order coincides with the equality). They form a full reflective sub-category of **Cpo** isomorphic to **Set**.

As above, the interpretation of initial algebras requires consideration of restricted classes of (strong) functors, to which we now turn.

**$\omega$-colimit preserving strong functors.** Let $F : m$ be a functor that preserves $\omega$-colimits. If $F$ is strong (respectively, preserves decidable objects) then so is its initial algebra functor. Thus, the strong functors, and their sub-class of functors preserving decidable objects, both form a suitable class for interpreting FML-functors. Among the strong functors which preserve $\omega$-colimits but not flat cpos there are: the constant functors corresponding to non-flat cpos, and the lifting functor $FX = X_\perp$.

**Functors shapely over lists.** The functors shapely over lists are always strong, and so form models as before. Such a functor $F$ preserves decidable objects iff $F(\overline{1})$ is decidable.

**Representations of functors shapely over lists.** Let $F$ be shapely over lists in **Cpo**. Since list objects decompose into countable many disjoint components, one for each possible length, the same is true for $F$. It follows that the lifting functor is not shapely over lists. More generally, $F$'s object of shapes $S$ can be decomposed into a set $U$ of connected components, i.e. represented as the colimit of a diagram $C : U \to \mathbf{Cpo}$. Thus, $F$ can be represented (up to natural isomorphism) by

$$F(\overline{X}) = \coprod_{u:U} C(u) \times (\prod_{i:m} X_i^{E(u,i)})$$

for some set $U$, and $E : U \to N^m$. In these terms, preservation of decidable objects amounts to requiring that each connected component be a singleton, i.e. $C(u) = 1$ for all $u$. Therefore, these functors have the same representations adopted for functors on **Set** shapely over lists. Then the intensional semantics in **Cpo** can be taken verbatim from **Set**, the only difference being that a pair $\langle S, E : S \to N^m \rangle$ now represents a strong functor on **Cpo** extending the functor on **Set** with the same representation.

**Inductive functors and closed term models.** These interpretations follow the same pattern as in **Set**.

## 5.2 Basic terms

This section introduces the basic term structure of FML, delaying the study of the combinators until the next section. As it is the novel combinators that give FML its flavor, this section is rather routine, but this makes the proofs routine, too. Further, all results are stable under change of the set of constants, both for functors and terms.

The terms are presented here à la Curry, i.e. with no explicit type or functor information. Explicitly typed terms, à la Church, will be considered in Section 5.5. There are two ways of assigning types to terms in the Hindley-Milner type system, either type schema or types [Tof88]. The former has separate rules for abstracting and instantiating type variables, whereas the latter combines these with the rules for typing variables, combinators and the let-construct. We will present both versions, show them equivalent (Lemma 5.9), and then present a modified version of Milner's algorithm W for inferring types (see [Mil78, Tof88]).

### 5.2.1 Syntax

The raw syntax for FML terms à la Curry is the same as that for the Hindley-Milner type system:

$$t ::= x \mid c \mid \lambda x.t \mid t_1\ t_2 \mid \text{let } x = t_1 \text{ in } t_2$$

Each combinator $c$ has an associated type schema $\sigma_c$ which will be define in the following section.

A *term context* is a sequence of $x : \sigma$ with no repetitions of term variables $x$. Define $\mathrm{DV}(\Gamma)$ to be the set of free term variables in $\Gamma$. A *context* $\Delta; \Gamma$ consists of a type context and a term context.

**Notation 5.5** The following notational conventions will be maintained throughout this paper. $x$ and $y$ range over term variables, $c$ ranges over combinators, and $t$ ranges over terms. $\Gamma$ ranges over term contexts. The usual conventions of $\lambda$-calculus concerning grouping of declared and bound variables apply (see [Bar84]). $\mathrm{FV}(t)$ is the set of free variables of $t$ and $[e/x]e'$ is the substitution of $e$ for $x$ in $e'$.

The judgment form $\Delta; \Gamma \vdash$ means that $\Delta; \Gamma$ is a well-formed context. The corresponding inference rules are

$$\text{empty}\ \frac{\Delta \vdash}{\Delta; \emptyset \vdash} \qquad \text{term}\ \frac{\Delta; \Gamma \vdash \quad \Delta \vdash \sigma}{\Delta; \Gamma, x : \sigma \vdash}\ x \notin \mathrm{DV}(\Gamma)$$

**Lemma 5.6** $\Delta; \Gamma \vdash$ *implies* $\Delta \vdash \Gamma(x)$ *for any* $x \in \mathrm{DV}(\Delta)$*.*

**Proof.** By induction on the structure of its premise. ∎

Let $\Delta_j; \Gamma_j$ be well-formed contexts for $j = 1, 2$. Define $S : \Delta_1; \Gamma_1 \to \Delta_2; \Gamma_2$ to mean that $S : \Delta_1 \to \Delta_2$ is a substitution, and that $\Gamma_1(x) = \sigma$ implies

$$\Delta_2; \Gamma_2 \vdash x : S(\sigma).$$

The inference rules for assigning type schema are given in Figure 5.2.

$$x \; \frac{\Delta; \Gamma \vdash}{\Delta; \Gamma \vdash x : \sigma} \; \sigma = \Gamma(x)$$

$$c \; \frac{\Delta; \Gamma \vdash}{\Delta; \Gamma \vdash c : \sigma_c}$$

$$\text{app} \; \frac{\Delta; \Gamma \vdash t : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash t_1 : \tau_1}{\Delta; \Gamma \vdash t \; t_1 : \tau_2}$$

$$\lambda \; \frac{\Delta; \Gamma, y : \tau_1 \vdash [y/x]t : \tau_2}{\Delta; \Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \; y \notin \mathrm{DV}(\Gamma)$$

$$\text{let} \; \frac{\Delta; \Gamma \vdash t_1 : \sigma_1 \quad \Delta; \Gamma, y : \sigma_1 \vdash [y/x]t_2 : \sigma_2}{\Delta; \Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : \sigma_2} \; y \notin \mathrm{DV}(\Gamma)$$

$$\text{App} \; \frac{\Delta; \Gamma \vdash t : \forall X : \mathrm{T}.\sigma \quad \Delta \vdash \tau : \mathrm{T}}{\Delta; \Gamma \vdash t : [\tau/X]\sigma}$$

$$\Lambda \; \frac{\Delta, Y : \mathrm{T}; \Gamma \vdash t : [Y/X]\sigma}{\Delta; \Gamma \vdash t : \forall X : \mathrm{T}.\sigma} \; Y \notin \mathrm{DV}(\Delta)$$

$$\text{App}_n \; \frac{\Delta; \Gamma \vdash t : \forall X : n.\sigma \quad \Delta \vdash F : n}{\Delta; \Gamma \vdash t : [F/X]\sigma}$$

$$\Lambda_n \; \frac{\Delta, Y : n; \Gamma \vdash t : [Y/X]\sigma}{\Delta; \Gamma \vdash t : \forall X : n.\sigma} \; Y \notin \mathrm{DV}(\Delta)$$

Figure 5.2: Assigning schema

$$x \; \frac{\Delta;\Gamma \vdash \quad S : \Delta' \to \Delta}{\Delta;\Gamma \vdash x : S(\tau)} \; \Gamma(x) = \forall\Delta'.\tau$$

$$c \; \frac{\Delta;\Gamma \vdash \quad S : \Delta' \to \Delta}{\Delta;\Gamma \vdash c : S(\tau)} \; \sigma_c = \forall\Delta'.\tau$$

$$\lambda \; \frac{\Delta;\Gamma, y : \tau_1 \vdash [y/x]t : \tau_2}{\Delta;\Gamma \vdash (\lambda x.t) : \tau_1 \to \tau_2} \; y \notin \mathrm{DV}(\Gamma)$$

$$\mathrm{app} \; \frac{\Delta;\Gamma \vdash t : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash t_1 : \tau_1}{\Delta;\Gamma \vdash (t\ t_1) : \tau_2}$$

$$\mathrm{let} \; \frac{\Delta,\Delta';\Gamma \vdash t_1 : \tau_1 \quad \Delta;\Gamma, y : (\forall\Delta'.\tau_1) \vdash [y/x]t_2 : \tau_2}{\Delta;\Gamma \vdash (\mathrm{let}\ x = t_1\ \mathrm{in}\ t_2) : \tau_2} \; y \notin \mathrm{DV}(\Gamma)$$

Figure 5.3: Assigning types

**Lemma 5.7** *The system defined in Figure 5.2 satisfies the following properties*

1. *If* $S : \Delta_1 \to \Delta_2$ *is a substitution then* $\Delta_1;\Gamma \vdash$ *implies* $\Delta_2; S(\Gamma) \vdash$.
2. *Well-typing:* $\Delta;\Gamma \vdash t : \sigma$ *implies* $\Delta \vdash \sigma$.
3. *If* $S$ *is a renaming of* $\Delta$ *then* $\Delta;\Gamma \vdash J$ *implies* $\Delta_S; S(\Gamma) \vdash S(J)$.
4. *Type substitution: if* $S : \Delta_1 \to \Delta_2$ *is a substitution then* $\Delta_1;\Gamma \vdash t : \sigma$ *implies* $\Delta_2; S(\Gamma) \vdash t : S(\sigma)$.
5. *Thinning:* $\Delta;\Gamma_1,\Gamma_2 \vdash J$ *implies* $\Delta;\Gamma_1, x : \sigma, \Gamma_2 \vdash J$ *for any* $x \notin \mathrm{DV}(\Gamma_1,\Gamma_2)$.
6. *Term substitution: if* $\Delta_1,\Delta;\Gamma_1 \vdash t : \sigma$ *then* $\Delta_1,\Delta_2;\Gamma_1, x : \sigma, \Gamma_2 \vdash t' : \sigma'$ *implies* $\Delta_1,\Delta_2;\Gamma_1,\Gamma_2 \vdash [t/x]t' : \sigma'$.

**Proof.** Each statement is proved by induction on the structure of its premise, in some cases using earlier statements in the lemma. ∎

The inference rules for assigning types (rather than schema) are given in Figure 5.3.

When assigning types, the definition of substitutions between contexts must be modified slightly, as follows. Let $\Delta_j;\Gamma_j$ be well-formed contexts for $j = 1, 2$. Define $S : \Delta_1;\Gamma_1 \to \Delta_2;\Gamma_2$ to mean that $S : \Delta_1 \to \Delta_2$ is a substitution, and that

$$\Delta_2, \Delta; \Gamma_2 \vdash x : S(\tau)$$

whenever $\Gamma_1(x) = \forall\Delta.\tau$. Note that $\alpha$-conversion is used to ensure that $\Delta_2$ and $\Delta$ have no variables in common.

**Lemma 5.8** *The system of Figure 5.3 satisfies the following properties.*

1. *Well-typing:* $\Delta; \Gamma \vdash t : \tau$ *implies* $\Delta \vdash \tau : \mathrm{T}$.
2. *If* $S : \Delta_1 \to \Delta_2$ *is a substitution then* $\Delta_1; \Gamma \vdash$ *implies* $\Delta_2; S(\Gamma) \vdash$.
3. *Type substitution: if* $S : \Delta_1 \to \Delta_2$ *is a substitution then* $\Delta_1; \Gamma \vdash t : \tau$ *implies* $\Delta_2; S(\Gamma) \vdash t : S(\tau)$.
4. *If* $S$ *is a renaming of* $\Delta$ *then* $\Delta; \Gamma \vdash J$ *implies* $\Delta_S; S(\Gamma) \vdash S(J)$.
5. *Thinning:* $\Delta; \Gamma_1, \Gamma_2 \vdash J$ *implies* $\Delta; \Gamma_1, x : \sigma, \Gamma_2 \vdash J$ *for any* $x \notin \mathrm{DV}(\Gamma_1, \Gamma_2)$.
6. *Term substitution: if* $\Delta_1, \Delta; \Gamma_1 \vdash t : \tau$ *then* $\Delta_1, \Delta_2; \Gamma_1, x : (\forall\Delta.\tau), \Gamma_2 \vdash t' : \tau'$ *implies* $\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash [t/x]t' : \tau'$.

**Proof.** Each statement is proved by induction on the structure of its premise, in some cases using earlier statements in the lemma. ∎

The following lemma compares inference of types with inference of schema. To distinguish them we use the notation $\Delta; \Gamma \vdash_\tau t : \tau$ for the former and $\Delta; \Gamma \vdash_\sigma t : \sigma$ for the latter.

**Lemma 5.9** $\Delta; \Gamma \vdash_\tau t : \tau$ *implies* $\Delta; \Gamma \vdash_\sigma t : \tau$. *Conversely,* $\Delta; \Gamma \vdash_\sigma t : \forall\Delta'.\tau$ *implies* $\Delta, \Delta'; \Gamma \vdash_\tau t : \tau$ *(assuming that $\alpha$-conversion is used to ensure well-formedness of the latter context).*

**Proof.** The proof is by straightforward induction on the structure of the proofs of the judgments. ∎

### 5.2.2 Type inference algorithm

Let $\Delta_1$ be a type context, $\Gamma$ be a term context and $t$ be a term. A *typing* for $(\Delta_1, \Gamma, t)$ consists of a triple $(\Delta_2, S, \tau)$ such that $S : \Delta_1 \to \Delta_2$ is a substitution and

$$\Delta_2; S(\Gamma) \vdash t : \tau.$$

- $W(\Delta, \Gamma, x) = (\Delta\ \Delta_1, \mathrm{id}, \tau)$, where $\Gamma(x) = \forall\Delta_1.\tau$
- $W(\Delta, \Gamma, c) = (\Delta\ \Delta_1, \mathrm{id}, \tau)$, where $\sigma_c = \forall\Delta_1.\tau$
- $W(\Delta, \Gamma, \lambda x.t) = (\Delta_1, S, SX \to \tau_2)$, where

$$(\Delta_1, S, \tau_2) = W(\Delta\ X : \mathrm{T}, \Gamma\ x : X, t)$$

- $W(\Delta, \Gamma, t\ t_1) = (\Delta_3, U\ R\ S, UX)$, where

$$\begin{array}{rcl}(\Delta_1, S, \tau) & = & W(\Delta, \Gamma, t) \\ (\Delta_2, R, \tau_1) & = & W(\Delta_1, S(\Gamma), t_1) \\ (\Delta_3, U) & = & \mathcal{U}(\Delta_2\ X : \mathrm{T}, R(\tau), \tau_1 \to X)\end{array}$$

- $W(\Delta, \Gamma, \mathrm{let}\ x = t_1\ \mathrm{in}\ t_2) = (\Delta_4, R\ S, \tau_2)$, where

$$\begin{array}{rcl}(\Delta_1, S, \tau_1) & = & W(\Delta, \Gamma, t_1) \\ \Delta_2 & = & \Delta_1\lceil(\cup\{\mathrm{FV}(SX) | X \in \mathrm{DV}(\Delta)\}) \\ \Delta_3 & = & \Delta_1 - \Delta_2 \\ (\Delta_4, R, \tau_2) & = & W(\Delta_2, S(\Gamma)\ x : \forall\Delta_3.\tau_1, t_2)\end{array}$$

  By definition $\Delta_2$ is the smallest sub-context of $\Delta_1$ such that $S : \Delta \to \Delta_2$, so that we will obtain $R\ S : \Delta \to \Delta_4$ as required.

Figure 5.4: Algorithm W

A *most general typing* for $(\Delta_1, \Gamma, t)$ is a typing as above such that if $(\Delta_2', S', \tau')$ is any other typing for it then there is a substitution $R : \Delta_2 \to \Delta_2'$ such that $R\ S = S'$ on $\mathrm{DV}(\Delta_1)$ and $R(\tau) = \tau'$.

Milner's algorithm $W$ can be modified to produce a most general typing for our terms, whenever any typing exists. In the description of the algorithm we assume that bound variables are renamed to avoid clashes, and fresh variables are introduced whenever needed.

**Theorem 5.10** *Let $\Delta_1; \Gamma$ be a well-formed context.*

1. *Soundness: if $W(\Delta_1, \Gamma, t) = (\Delta_2, S, \tau)$ then $S : \Delta_1 \to \Delta_2$ and $\Delta_2; S(\Gamma) \vdash t : \tau$.*
2. *Completeness: if $S' : \Delta_1 \to \Delta_3$ and $\Delta_3; S'(\Gamma) \vdash t : \tau'$, then ($W$ succeeds and) there exists a substitution $R : \Delta_2 \to \Delta_3$ such that $S' = R\ S$ on $\mathrm{DV}(\Delta_1)$ and $\tau' = R(\tau)$.*

**Proof.** *Both statements are proved by induction on the structure of $t$ (see [Tof88]) and use type substitution (see Lemma 5.8).* ∎

## 5.3 Constants and reductions

FML is a general framework for discussing functor-based polymorphism, whose detailed structure will be reflected in the choice of constants, or (combinators) of the language. These constants come in families indexed by functor arities. We will begin with the constants for mapping, as these are central to the functorial approach. To this can be added constants that correspond to introduction and elimination rules for the functor and type constructors of the language, in the style advocated in Martin-Löf type theory [NPS90]. The choice of constants and their names will be made to reflect this fact. Afterwards, we will introduce a comprehensive system of syntactic sugar that simplifies the process of constructing terms. While the fundamentals are unchanged from those of [BJM96] the surface syntax is changed substantially.

As the FML functors can be interpreted as being shapely over lists, we can introduce more constants motivated by such an interpretation. For example, we can introduce constants that allow the extraction of a data list from a value of functor type. Also, since the FML functors all have decidable objects of shapes, we can support the zipping of values that have the same shape, e.g. lists having the same length. Further, in Section 5.5 we introduce case analysis over functors when defining polytypic terms, since this is validated by the inductive functor interpretation of FML functors.

It is convenient to present the reduction rules with the constants. These are used to generate a rewriting system in the usual way, by applying reductions to arbitrary sub-terms.

**Beta-reduction.** Of course, the basic language has the standard reduction rules of the $\lambda$-calculus with the let construct for local definitions:

$$
\begin{array}{rcl}
(\lambda x.t_2)\ t_1 & > & [t_1/x]t_2 \\
\mathrm{let}\ x = t_1\ \mathrm{in}\ t_2 & > & [t_1/x]t_2
\end{array}
$$

### 5.3.1 Basic constants and $\mathrm{FML}_{basic}$

In this section we fix a set of basic combinators, and call the resulting calculus $\mathrm{FML}_{basic}$. The key polytypic combinator is mapping.

**Mapping.** There is one family of constants that express the action of functors on morphisms of a category, namely

$$\mathrm{map}_m : \forall F : m.\forall X_{i:m}, Y_{i:m} : \mathrm{T}.(X_i \to Y_i)_{i:m} \to F(\overline{X}) \to F(\overline{Y})$$

Of course, providing reduction rules for $\mathrm{map}_m$ is by no means trivial, as explained in the introduction. In fact it has no reduction rules of its own, but only interacts with the functor-based term constructors.

**Term constructors.** For each functor constructor, we give the corresponding term constructors:

$$\begin{array}{rcl}
\mathrm{intro}^1 & : & 1 \\
\mathrm{intro}^x & : & \forall X_0, X_1 : \mathrm{T}.\ X_0 \to X_1 \to X_0 \times X_1 \\
\mathrm{intro}^+_j & : & \forall X_0, X_1 : \mathrm{T}.\ X_j \to X_0 + X_1 \qquad \text{for } j : 2 \\
\mathrm{intro}^\Pi_{m,i} & : & \forall X_{j:m} : \mathrm{T}.\ X_i \to \Pi^m_i(\overline{X}) \\
\mathrm{intro}^\circ_{m,n} & : & \forall F : m.\forall G_{i:m} : n.\forall X_{j:n} : \mathrm{T}.\ F(G_i(\overline{X})_{i:m}) \to F\langle\overline{G}\rangle^n(\overline{X}) \\
\mathrm{intro}^\mu_m & : & \forall F : m+1.\forall X_{i:m} : \mathrm{T}.\ F(\overline{X}, \mu F(\overline{X})) \to \mu F(\overline{X}).
\end{array}$$

**Mapping.** Mapping preserves the structure of its data argument, i.e. it commutes with the functor-based term constructors. The reductions are:

$$\begin{array}{rcl}
\mathrm{map}_0\ \mathrm{intro}^1 & > & \mathrm{intro}^1 \\
\mathrm{map}_2\ f_0\ f_1\ (\mathrm{intro}^x\ t_0\ t_1) & > & \mathrm{intro}^x\ (f_0\ t_0)\ (f_1\ t_1) \\
\mathrm{map}_2\ f_0\ f_1\ (\mathrm{intro}^+_j\ t) & > & \mathrm{intro}^+_j\ (f_j\ t) \\
\mathrm{map}_m\ f_{k:m}\ (\mathrm{intro}^\Pi_{m,i}\ t) & > & \mathrm{intro}^\Pi_{m,i}\ (f_i\ t) \\
\mathrm{map}_n\ f_{k:n}\ (\mathrm{intro}^\circ_{m,n}\ t) & > & \mathrm{intro}^\circ_{m,n}\ (\mathrm{map}_m\ (\mathrm{map}_n\ \overline{f})_{i:m}\ t) \\
\mathrm{map}_m\ f_{i:m}\ (\mathrm{intro}^\mu_m\ t) & > & \mathrm{intro}^\mu_m\ (\mathrm{map}_{m+1}\ \overline{f}\ (\mathrm{map}_m\ \overline{f})\ t).
\end{array}$$

The rule for composite functors asserts that mapping over a composite functor, say $F\langle G\rangle^1$ is given by mapping with respect to $F$ the result of mapping with respect to $G$. Also, in the rule for initial algebra functors, $\mathrm{map}_m$ reduces to an expression involving $\mathrm{map}_{m+1}$ i.e. depends on the functoriality of the original functor, as described in Section 5.1.

**Term Destructors.** According to type theory, given the constructors for a type, there is a canonical way to derive its destructors and computational rules:

$$\mathrm{elim}^1 \ :\ \forall X : \mathrm{T}.\ X \to 1 \to X$$

$$
\begin{array}{rcl@{\qquad}rcl}
\text{apply } f\ x & = & f\ x & \text{case } t \text{ of} & = & \text{elim}^{+}\ (\lambda x.t_0)\ (\lambda y.t_1)\ t \\
\text{id} & = & \lambda x.x & \text{in}_0\ x \Rightarrow t_0 & & \\
g \circ f & = & \lambda x.g\ (f\ x) & \text{in}_1\ y \Rightarrow t_1 & & \\
\text{un} & = & \text{intro}^{1} & \text{tag}_{m,i} & = & \text{intro}^{\Pi}_{m,i} \\
\text{bang } x & = & \text{un} & \text{untag}_{m,i} & = & \text{elim}^{\Pi}_{m,i}\ \text{id} \\
\text{pair} & = & \text{intro}^{\times} & \text{pack}_{m,n} & = & \text{intro}^{\circ}_{m,n} \\
\langle t_0, t_1 \rangle & = & \text{intro}^{\times}\ t_0\ t_1 & \text{unpack}_{m,n} & = & \text{elim}^{\circ}_{m,n}\ \text{id} \\
\pi_0 & = & \text{elim}^{\times}\ (\lambda x, y.x) & \text{wrap}_m & = & \text{intro}^{\mu}_{m} \\
\pi_1 & = & \text{elim}^{\times}\ (\lambda x, y.y) & \text{fold}_m & = & \text{elim}^{\mu}_{m} \\
\text{in}_j & = & \text{intro}^{+}_{j} & \text{unwrap}_m & = & \text{fold}_m\ (\text{map}_{m+1}\ \overline{\text{id}}\ \text{wrap}_m)
\end{array}
$$

Figure 5.5: Syntactic sugar

$$
\begin{array}{rcl}
\text{elim}^{\times} & : & \forall X_0, X_1, Y : \mathrm{T}.\ (X_0 \to X_1 \to Y) \to X_0 \times X_1 \to Y \\
\text{elim}^{+} & : & \forall X_0, X_1, Y : \mathrm{T}.\ (X_0 \to Y) \to (X_1 \to Y) \to X_0 + X_1 \to Y \\
\text{elim}^{\Pi}_{m,i} & : & \forall X_{j:m}, Y : \mathrm{T}.\ (X_i \to Y) \to \Pi^{m}_{i}(\overline{X}) \to Y \\
\text{elim}^{\circ}_{m,n} & : & \forall F : m.\forall G_{i:m} : n.\forall X_{j:n}, Y : \mathrm{T}.(F(G_i(\overline{X})_{i:m}) \to Y) \to F\langle \overline{G} \rangle^{n}(\overline{X}) \to Y \\
\text{elim}^{\mu}_{m} & : & \forall F : m+1.\forall X_{i:m}, Y : \mathrm{T}.\ (F(\overline{X}, Y) \to Y) \to \mu F(\overline{X}) \to Y.
\end{array}
$$

Each canonical destructor has a corresponding reduction rule:

$$
\begin{array}{rcl}
\text{elim}^{1}\ t\ \text{intro}^{1} & > & t \\
\text{elim}^{\times}\ f\ (\text{intro}^{\times}\ t_0\ t_1) & > & f\ t_0\ t_1 \\
\text{elim}^{+}\ f_0\ f_1\ (\text{intro}^{+}_{j}\ t) & > & f_j\ t \\
\text{elim}^{\Pi}_{m,i}\ f\ (\text{intro}^{\Pi}_{m,i}\ t) & > & f\ t \\
\text{elim}^{\circ}_{m,n}\ f\ (\text{intro}^{\circ}_{m,n}\ t) & > & f\ t \\
\text{elim}^{\mu}_{m}\ f\ (\text{intro}^{\mu}_{m}\ t) & > & f\ (\text{map}_{m+1}\ (\lambda x.x)_{i:m}\ (\text{elim}^{\mu}_{m}\ f)\ t).
\end{array}
$$

Most of these rules follow the familiar structure, returning a strictly simpler expression. The last of them is different, as the reduction of a fold introduces a mapping, as explained in Section 5.1.

### 5.3.2 Syntactic sugar

We introduces more familiar synonyms for some of the constants, and some handier syntax for some common patterns in Figure 5.5. The declaration $f\ x = t$ is syntactic sugar for the declaration $f = \lambda x.t$. Multiple argument parameters are handled similarly. Composition associates to the right.

#### 5.3.2.1 Monads

In ML and FML a monad $M$ can be described by giving a type expression in one free variable $X : \mathrm{T} \vdash M(X) : \mathrm{T}$ and polymorphic terms of suitable type

$$\begin{array}{rcl} \mathit{val} & : & \forall X : \mathrm{T}.X \to M(X) \\ \mathit{let} & : & \forall X, Y : \mathrm{T}.(X \to M(Y)) \to M(X) \to M(Y) \end{array}$$

Note the use of slanted font to distinguish *let* from the let-construct let ... in ... of FML. In general we do not require that $M(X)$ be represented by an FML functor, since that would not allow the use of function types.

A very simple example is the Maybe monad, which is specified by

$$\begin{array}{rcl} \mathrm{Maybe}(X) & = & X + 1 \\ \mathit{val} & = & \lambda x.\ \mathrm{in}_0\ x \\ \mathit{let} & = & \lambda f, c.\ \mathrm{elim}^+\ f\ \mathrm{in}_1\ c \\ \mathit{fail} & = & \mathrm{in}_1\ \mathrm{un} \end{array}$$

where the constant *fail* represents failure.

Another example is the *parsing monad* $P_S(X)$. This is actually a family of monads parameterized with respect to a type variable $S$.

$$\begin{array}{rcl} P_S(X) & = & S \to (X \times S) + 1 \\ \mathit{val} & = & \lambda x, s.\ \mathrm{in}_0\ \langle x, s \rangle \\ \mathit{let} & = & \lambda f, c, s.\ \mathrm{elim}^+\ (\mathrm{elim}^x\ (\lambda x, s. f\ x\ s))\ \mathrm{in}_1\ (c\ s) \\ \mathit{fail} & = & \lambda s.\ \mathrm{in}_1\ \mathrm{un} \end{array}$$

$S$ should be thought of as a state, which may change during the computation. Note that $P_S(X)$, unlike Maybe, cannot be represented by an FML functor.

We introduce some syntactic sugar for the parsing monads, which will be used in Section 5.3.3

$$\begin{array}{rcl} [t] & = & \mathit{val}\ t \\ \mathrm{let}\ x \Leftarrow t_1\ \mathrm{in}\ t_2 & = & \mathit{let}\ (\lambda x.t_2)\ t_1 \end{array}$$

Maybe($X$) and $P_1(X)$ are isomorphic, so we write inverse mediating isomorphisms *pars2maybe* : $P_1(X) \to \mathrm{Maybe}(X)$ and *maybe2pars* : $\mathrm{Maybe}(X) \to P_1(X)$.

$$
\begin{tikzcd}
1() \arrow[r, "\mathrm{pack}_{0,2}"] \arrow[d, "\mathrm{map}_0"] & 1\langle\rangle^2(X, LX) \arrow[r, "\mathrm{in}_0"] \arrow[d, "g"'] & F(X) \arrow[r, "\mathrm{pack}_{2,2}"] \arrow[d, "\mathrm{map}_2\ g\ g"'] & Lu(X, LX) \arrow[r, "\mathrm{wrap}_1"] \arrow[d, "g"'] & LX \arrow[d, "\mathrm{map}_1\ f"'] \\
1() \arrow[r, "\mathrm{pack}_{0,2}"] & 1\langle\rangle^2(Y, LY) \arrow[r, "\mathrm{in}_0"] & F(Y) \arrow[r, "\mathrm{pack}_{2,2}"] & Lu(Y, LY) \arrow[r, "\mathrm{wrap}_1"] & LY
\end{tikzcd}
$$

Figure 5.6: $\mathrm{map}_1 f$ nil

#### 5.3.2.2 Lists

Now let us consider some syntactic sugar for the list functor, $L = \mu Lu$ where $Lu = +\langle 1\langle\rangle^2, \times\rangle^2$. For any type $X$ we have the constructors

$$
\begin{array}{rcl}
\mathrm{nil} & = & (\mathrm{wrap}_1 \circ \mathrm{pack}_{2,2} \circ \mathrm{in}_0 \circ \mathrm{pack}_{0,2})\ \mathrm{un} \\
\mathrm{cons}\ x\ y & = & (\mathrm{wrap}_1 \circ \mathrm{pack}_{2,2} \circ \mathrm{in}_1)\ \langle x, y\rangle \\
\mathrm{single}\ x & = & \mathrm{cons}\ x\ \mathrm{nil} \\
\mathrm{append}\ x\ y & = & \mathrm{fold}_1\ ((\mathrm{elim}^+\ (\lambda x.\ \mathrm{id}) \\
 & & (\lambda y, z.\ \mathrm{cons}\ (\pi_0\ y)\ (\pi_1\ y\ z))) \circ \mathrm{unpack}_{2,2})\ x\ y \\
\mathrm{flatten}\ x & = & \mathrm{fold}_1\ (\lambda y, z.\ \mathrm{append}\ y\ z)\ x.
\end{array}
$$

The composite applied to un in defining nil is displayed as the top line of Figure 5.6. Let us see how the usual pattern-matching reductions for mapping and folding over lists can be recovered as composite reductions. Other inductive types are handled similarly. Let $f : X \to Y$ be a morphism. Then $\mathrm{map}_1\ f$ nil reduces to nil by five map-reductions, as diagrammed in Figure 5.6, where $g = \mathrm{map}_2\ f\ (\mathrm{map}_1\ f)$ and $F(X) = 1\langle\rangle^2(X, LX) + X \times LX$. Similarly,

$$
\begin{array}{rcl}
\mathrm{map}_1\ f\ (\mathrm{cons}\ h\ t) & \to & (\mathrm{wrap}_1 \circ \mathrm{pack}_{2,2} \circ \mathrm{in}_1)\ (\mathrm{map}_2\ f\ (\mathrm{map}_1\ f)\ \langle h, t\rangle) \\
 & \to & \mathrm{cons}\ (f\ h)\ (\mathrm{map}_1\ f\ t).
\end{array}
$$

Now consider folding over a list. Given terms $d : D$ and $g : X \times D \to D$ their (ordinary) list fold is given by

$$f = (\mathrm{elim}^+\ (\lambda x.d)\ g) \circ \mathrm{unpack}_{2,2} : Lu(X, D) \to D.$$

Hence, given $h : X$ and $t : LX$ we have:

$$\mathrm{fold}_1\ f\ \mathrm{nil} \quad \to \quad f(\mathrm{map}_2\ \mathrm{id}\ (\mathrm{fold}_1\ f)\ ((\mathrm{pack}_{2,2} \circ \mathrm{in}_0 \circ \mathrm{pack}_{0,2})\ \mathrm{un}))$$

$$
\begin{aligned}
& \to && f(\mathrm{pack}_{2,2}(\mathrm{in}_0(\mathrm{pack}_{0,2}\ \mathrm{un}))) \\
& \to && d \\
\mathrm{fold}_1\ f\ (\mathrm{cons}\ h\ t) & \to && f\ (\mathrm{map}_2\ \mathrm{id}\ (\mathrm{fold}_1\ f)\ ((\mathrm{pack}_{2,2} \circ \mathrm{in}_1\ (\mathrm{pair}\ h\ t)))) \\
& \to && f\ ((\mathrm{pack}_{2,2} \circ \mathrm{in}_1\ (\mathrm{pair}\ h\ (\mathrm{fold}_1\ f\ t)))) \\
& \to && g\ (\mathrm{pair}\ h\ (\mathrm{fold}_1\ f\ t)).
\end{aligned}
$$

### 5.3.3 Shape-based constants and $\mathrm{FML}_{shape}$

The shape-based combinators are motivated by the interpretation of FML functors as functors shapely over lists and with a decidable shape. In this case the semantics is used both to justify the addition of a polytypic combinator, and to decide when we have introduced enough combinators. We call $\mathrm{FML}_{shape}$ the extension of $\mathrm{FML}_{basic}$ with the additional combinators ltraverse and zipop.

**Traversals.** The FML functors are all *shapely over lists*. In particular, the ordering of the list entries determines an ordering on the data in the structure, which can be used to define traversals. A full treatment of the semantics and uses of traversals lies beyond the scope of this paper, but we introduce them now as a convenient generalization of the data extraction constant which follows.

Traversal is the process of visiting each node in a data structure, and performing some action. If we do not wish to limit the nature of this action in advance, then we could represent it using a generic computational monad [Mog91b, Wad93]

$$
\begin{aligned}
\mathrm{ltraverse}_m \quad : \quad & \forall M : Monad . \forall F : m . \forall X_{i:m}, Y_{i:m} : \mathrm{T}. \\
& \overline{(X \to MY)} \to F(\overline{X}) \to M(F(\overline{Y})).
\end{aligned}
$$

However, this would require an extension of FML with qualified kinds. Instead, we will parameterize $\mathrm{ltraverse}_m$ with respect to parsing monads $P_S(X) = S \to (X \times S) + 1$ and make use of the syntactic sugar introduced in Section 5.3.2.1. The constants for *left traversal* are

$$
\mathrm{ltraverse}_m : \forall S : \mathrm{T}. \forall F : m . \forall X_{i:m}, Y_{i:m} : \mathrm{T}.\ \overline{(X \to P_S(Y))} \to F(\overline{X}) \to P_S(F(\overline{Y}))
$$

whose evaluation rules are given by

$$
\begin{aligned}
\mathrm{ltraverse}_0\ \mathrm{un} \quad & > \quad [\mathrm{un}] \\
\mathrm{ltraverse}_2\ f_0\ f_1\ \langle t_0, t_1 \rangle \quad & > \quad \mathrm{let}\, y_0 \Leftarrow f_0\ t_0 \,\mathrm{in} \\
& \qquad \mathrm{let}\, y_1 \Leftarrow f_1\ t_1 \,\mathrm{in}\, [\langle y_0, y_1 \rangle] \\
\mathrm{ltraverse}_2\ f_0\ f_1\ (\mathrm{in}_j\ t) \quad & > \quad \mathrm{let}\, y \Leftarrow f_j\ t \,\mathrm{in}\, [\mathrm{in}_j\ y]
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathrm{ltraverse}_m\ f_{i:m}\ (\mathrm{tag}_{m,i}\ t) & > & \mathrm{let}\, y \Leftarrow f_i\ t\, \mathrm{in}\, [\mathrm{tag}_{m,i}\ y] \\
\mathrm{ltraverse}_n\ f_{i:n}\ (\mathrm{pack}_{m,n}\ t) & > & \mathrm{let}\, y \Leftarrow \mathrm{ltraverse}_m\ (\mathrm{ltraverse}_n\ \overline{f})_{i:m}\ t\, \mathrm{in}\, [\mathrm{pack}_{m,n}\ y] \\
\mathrm{ltraverse}_m\ f_{i:m}\ (\mathrm{wrap}_m\ t) & > & \mathrm{let}\, y \Leftarrow \mathrm{ltraverse}_{m+1}\, \overline{f}\ (\mathrm{ltraverse}_m\ \overline{f})\ t\, \mathrm{in}\, [\mathrm{wrap}_m\ y]
\end{array}
$$

Using the normal presentation of initial algebras (in which the recursion parameter is never to the left of an ordinary parameter in a product) left traversal corresponds to top-down traversal of the resulting tree. Then bottom-up traversal is obtained by performing a *right traversal* $\mathrm{rtraverse}_m$ which has the same type and reduction rules as ltraverse except that the rule for pairs has been changed to

$$
\begin{array}{rcl}
\mathrm{rtraverse}_2\ f_0\ f_1\ \langle t_0, t_1\rangle & > & \mathrm{let}\, y_1 \Leftarrow f_1\ t_1\, \mathrm{in} \\
 & & \mathrm{let}\, y_0 \Leftarrow f_0\ t_0\, \mathrm{in}\, [\langle y_0, y_1\rangle]
\end{array}
$$

so that the recursion parameter is traversed before the others.

A specialization of ltraverse yields the operations for extracting data

$$\mathrm{extract}_{m,i} : \forall F : m. \forall X_{i:m} : \mathrm{T}.\ F(\overline{X}) \to L(X_i)$$

which is defined using the monad $P_{LX_i}$

$$
\begin{array}{rl}
\mathrm{extract}_{m,i}\ t = & \mathrm{case}\ \mathrm{ltraverse}_m \overline{f}\ t\ \mathrm{nil}\ \mathrm{of} \\
 & \mathrm{in}_0\ x \Rightarrow \pi_1\ x \\
 & \mathrm{in}_1\ x \Rightarrow \mathrm{nil}
\end{array}
$$

where $f_j : X_j \to P_{LX_i} 1$ is given by $f_j\ x\ l = [\langle \mathrm{un}, \mathrm{cons}\ x\ l\rangle]$ when $j = i$, and $f_j\ x = [\mathrm{un}]$ otherwise.

**Zipping.** The constants

$$
\begin{array}{rcl}
\mathrm{zipop}_m & : & \forall S : \mathrm{T}. \forall F : m. \forall X_{i:m}, Y_{i:m}, Z_{i:m} : \mathrm{T}. \\
 & & (X_i \to Y_i \to P_S(Z_i))_{i:m} \to F(\overline{X}) \to F(\overline{Y}) \to P_S(F(\overline{Z}))
\end{array}
$$

are used to represent the zipping of two values having the same shape, and then applying the given functions to pairs of values. The reduction rules for zipop are given by

$$
\begin{array}{rcl}
\mathrm{zipop}_0\ \mathrm{un}\ \mathrm{un} & > & [\mathrm{un}] \\
\mathrm{zipop}_2\ f_0\ f_1\ \langle s_0, s_1\rangle\ \langle t_0, t_1\rangle & > & \mathrm{let}\, z_0 \Leftarrow f_0\ s_0\ t_0\, \mathrm{in} \\
 & & \mathrm{let}\, z_1 \Leftarrow f_1\ s_1\ t_1\, \mathrm{in}\, [\langle z_0, z_1\rangle] \\
\mathrm{zipop}_2\ f_0\ f_1\ (\mathrm{in}_j\ s)\ (\mathrm{in}_j\ t) & > & \mathrm{let}\, z \Leftarrow f_j\ s\ t\, \mathrm{in}\, [\mathrm{in}_j\ z] \\
\mathrm{zipop}_2\ f_0\ f_1\ (\mathrm{in}_0\ s)\ (\mathrm{in}_1\ t) & > & \mathit{fail}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{zipop}_2\ f_0\ f_1\ (\mathrm{in}_1\ s)\ (\mathrm{in}_0\ t) & > & \mathit{fail} \\
\mathrm{zipop}_m\ f_{i:m}\ (\mathrm{tag}_{m,i}\ s)\ (\mathrm{tag}_{m,i}\ t) & > & \mathrm{let}\, z \Leftarrow f_i\ s\ t\, \mathrm{in}\, [\mathrm{tag}_{m,i}\ z] \\
\mathrm{zipop}_n\ f_{i:n}\ (\mathrm{pack}_{m,n}\ s)\ (\mathrm{pack}_{m,n}\ t) & > & \mathrm{let}\, z \Leftarrow \mathrm{zipop}_m\ (\mathrm{zipop}_n\ \overline{f})_{i:m}\ s\ t\, \mathrm{in} \\
 & & [\mathrm{pack}_{m,n}\ z] \\
\mathrm{zipop}_m\ f_{i:m}\ (\mathrm{wrap}_m\ s)\ (\mathrm{wrap}_m\ t) & > & \mathrm{let}\, z \Leftarrow \mathrm{zipop}_{m+1}\ \overline{f}\ (\mathrm{zipop}_m\ \overline{f})\ s\ t\, \mathrm{in} \\
 & & [\mathrm{wrap}_m\ z]
\end{array}
$$

Note that if we instantiate the parsing monad with 1, we obtain a type constructor isomorphic to the Maybe monad (see Section 5.3.2.1). Since it is sometimes useful to work directly with Maybe, we introduce the function $\mathrm{zipop}'_m$:

$$
\begin{array}{rcl}
\mathrm{zipop}'_m & : & \forall F : m.\forall X_{i:m}, Y_{i:m}, Z_{i:m} : \mathrm{T}. \\
 & & (X_i \to Y_i \to \mathrm{Maybe}(Z_i))_{i:m} \to F(\overline{X}) \to F(\overline{Y}) \to \mathrm{Maybe}(F(\overline{Z}))
\end{array}
$$

defined as $\mathrm{zipop}'_m\ f\ s\ t = \mathit{pars2maybe}\,(\mathrm{zipop}_m\ (\lambda x, y.\mathit{maybe2pars}\,(f\ x\ y))\ s\ t)$.

From $\mathrm{zipop}'$ we can define

$$
\begin{array}{rcl}
\mathrm{zip}_m & = & \mathrm{zipop}'_m\ (\lambda x, y.\mathrm{in}_0(\mathrm{pair}\ x\ y))_{i:m} \\
 & : & \forall F : m.\forall X_{i:m} : \mathrm{T}.\forall Y_{i:m} : \mathrm{T}.F(\overline{X}) \to F(\overline{Y}) \to \mathrm{Maybe}(F(\overline{X \times Y}))
\end{array}
$$

Further, we can define a polymorphic equality. Assume that there is an equality test $\mathrm{eq}_i : X_i \to X_i \to 1 + 1$ for each $X_i$ then define

$$
\mathrm{eq}_m\ x\ y \quad = \quad \mathrm{let}\, z \Leftarrow (\mathrm{zipop}'_m\ \mathrm{eq}_{i:m}\ x\ y)\, \mathrm{in}\, \mathrm{in}_0\ \mathrm{un}
$$

Setting all $X_i$ to be 1 with the obvious equality, we obtain a test for shape equality $F(\overline{1}) \to F(\overline{1}) \to 1 + 1$.

**Remark 5.11** We could have parameterized ltraverse and zipop with respect to any monad (with a *fail* operation) expressible in FML.

### 5.3.4 Properties of reduction

Let $\mathrm{F}\beta$ be the rewriting system generated by the above rules (with, or without ltraverse or zipop).

**Theorem 5.12 (SR)** *Let $t > t'$. If $\Delta; \Gamma \vdash t : \sigma$ then $\Delta; \Gamma \vdash t' : \sigma$.*

**Proof.** Without loss of generality, one can assume that the reduction is basic and perform the proof by case analysis. In each case one has to analyze only the last rules in the derivation of the premise, using Lemma 5.8 to handle term substitutions. ∎

**Theorem 5.13 (CR)** *F$\beta$ on untyped terms is Church-Rosser.*

**Proof.** Standard. The combinator reductions rules for F$\beta$ are left-linear and non overlapping, and one can apply the result in [Acz78] (see also [Klo80]). ∎

**Corollary 5.14 (CR)** *F$\beta$ on typable terms is Church-Rosser.*

**Proof.** Immediate from SR and CR on untyped terms. ∎

**Theorem 5.15 (SN)** *If $\Delta; \Gamma \vdash t : \sigma$, then $t$ is strongly normalizing.*

**Proof.** We prove SN for a system more powerful than FML, *functorial* F (briefly FF), which can type every term typable in FML, therefore SN for FF trivially implies SN for FML. The proof follows [Men91] and uses semantic techniques (reducibility candidates). The details are available in the Appendix B. ∎

## 5.4 Extended examples

In this section we present some examples to show the expressiveness of FML. FML has been presented as an extension of ML but it is rather awkward in comparison with ML because we have to deal explicitly with term constructors and destructors corresponding to functor constructors and we have to use the constant $\mathrm{fold}_m$ to implement primitive recursion. However we could introduce also in FML some of the nice features of ML, such as datatype definitions and pattern matching definitions, and take advantage of functors without losing the programmability. In this way we can point out when FML features, in particular functors, are really necessary.

So we give two versions of the examples. The first one uses ML-like datatype definitions, pattern matching for function definitions and FML functors only when strictly needed. The second one uses only "pure FML" and thus shows how to deal with term constructors and destructors. In each case, programs are constructed top-down, i.e. giving auxiliary functions after their use. Each function is presented by giving its type, value and description.

**Syntactic sugar.** In addition to the syntactic sugar defined in section 5.3.2, we use the following syntax:

- $M(X) = \text{Maybe}(X)$ with the usual operations on error monad: $\text{ok}\ x = \text{in}_0\ x$, $\text{fail} = \text{in}_1\ \text{un}$.
- $\mathit{Bool} = M(1)$ with $\top = \text{ok un}$, $\bot = \text{fail}$, $\text{if}\ b\ \text{then}\ t_0\ \text{else}\ t_1 = \text{elim}^+(\text{elim}^1\ t_0)(\text{elim}^1\ t_1)\ b$.
- $X^2 = X \times X$.

### 5.4.1 One-side matching

Consider the constructor of a recursively defined algebra, i.e. of initial algebras for a functor. Such an algebra of closed terms can then be extended with an anonymous variable or placeholder called Any to create open terms. Every occurrence of Any in a term behaves like a fresh variable. The one-sided matching problem is to determine whether there is a substitution of terms for placeholders which creates a match between an open term and a closed term. Note that different occurrences of the placeholder in the same term can match different terms, so there is no need to consider substitutions.

Fix $m$, let $F$ be a functor of arity $m+1$ and let $X_{i\in m}$ be $m$ equality types, that is, for each $i \in m$ there exists a function $eq_i : X_i \to X_i \to \mathit{Bool}$.

#### 5.4.1.1 One-side matching — ML-like version

Define the algebra of closed terms by

$$G = \text{Cns}\ \mathit{of}\ F(\overline{X}, G).$$

Then the algebra of open terms with placeholder Any is given by

$$G' = \text{Any} \mid \text{Cns}'\ \mathit{of}\ F(\overline{X}, G').$$

The functor $F$ can be seen as a constructor for terms. So $G$ is the type of terms constructed from $F$ and $G'$ is the type of terms constructed from $F$ and the placeholder Any.

The function

$\mathsf{match} : G' \to G \to \mathit{Bool}$

$\mathsf{match}\ \text{Any}\ (\text{Cns}\ w) = \top$

$\mathsf{match}\ (\mathrm{Cns}'\ w')\ (\mathrm{Cns}\ w) = \mathsf{maybeBang}\ (\mathrm{zipop}'_{m+1}\ \mathrm{eq}_{i:m}\ \mathsf{match}\ w'\ w)$

takes a term of type $G'$ and a term of type $G$. If the first term is Any then it matches any term. Otherwise the function checks if the structures of the two terms agree. Here

$$\begin{aligned}\mathrm{zipop}'_{m+1} \ :\ & (X_i \to X_i \to M(1))_{i:m} \to (G' \to G \to M(1)) \to \\ & F(\overline{X}, G') \to F(\overline{X}, G) \to M(F(\overline{1}, 1))\end{aligned}$$

checks for equality of the constructors, and matching of the recursive sub-terms. Recall that $\mathit{Bool} = M(1)$ and that $\top$ is identified with (ok un) and $\bot$ with fail. The sub-term $(\mathrm{zipop}'_{m+1}\ \mathrm{eq}_{i:m}\ \mathsf{match}\ w'\ w)$ fails as soon as an equality test $\mathrm{eq}_i$ or a recursive call to $\mathsf{match}$ fails. Otherwise it returns the common shape of $w$ and $w'$, of type $F(\overline{1}, 1)$.

The function

$\mathsf{maybeBang} : \forall X : \mathrm{T}.M(X) \to M(1)$

$$\begin{aligned}\mathsf{maybeBang}\ z = \ & \mathrm{case}\ z\ \mathrm{of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{in}_0\ \mathrm{un} \\ & \mathrm{in}_1\ y \Rightarrow \mathrm{fail}\end{aligned}$$

is required to get the right type after applying $\mathrm{zipop}'_{m+1}$.

Note that in ML, one should write a specialization of $\mathrm{zipop}'$ for each particular choice of the functor $F$.

#### 5.4.1.2 One-side matching — pure FML version

The data types $G$ and $G'$ defined above can be defined in FML respectively as $\mu F(\overline{X})$ and $\mu F'(\overline{X})$ where

$$F' = +\langle 1\langle\rangle^{m+1}, F\rangle^{m+1} : m+1.$$

The functor polymorphic function

$\mathsf{match} : \mu F'(\overline{X}) \to \mu F(\overline{X}) \to \mathit{Bool}$

$\mathsf{match} = \mathrm{fold}_m\ (\mathsf{f} \circ \mathsf{unft})$

is defined using induction ($\mathrm{fold}_m$) on the first argument.

$\mathsf{unft} : F'(\overline{X}, Y) \to 1 + F(\overline{X}, Y)$

$$\begin{aligned}\mathsf{unft}\ z = \ & \mathrm{case}\ \mathrm{unpack}_{2,m+1}\ z\ \mathrm{of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{unpack}_{0,m+1}\ x \\ & \mathrm{in}_1\ y \Rightarrow y\end{aligned}$$

This coerces the type $F'(\overline{X}, Y)$ to the type $1 + F(\overline{X}, Y)$.

$\mathsf{f} : 1 + F(\overline{X}, \mu F(\overline{X}) \to \mathit{Bool}) \to \mu F(\overline{X}) \to \mathit{Bool}$

$$\begin{aligned}\mathsf{f}\ z\ w = \ & \text{case } z \text{ of} \\ & \text{in}_0\ x \Rightarrow \top \\ & \text{in}_1\ y \Rightarrow \mathsf{maybeBang}(\text{zipop}'_{m+1}\ \text{eq}_{i:m}\ \text{apply}\ y\ (\text{unwrap}_m\ w))\end{aligned}$$

This is the core of the function $\mathsf{match}$.

The function $\mathsf{maybeBang}$ is defined as in the ML-like version.

### 5.4.2 Unification

In unification literature, a term is usually defined to be either a variable or an application of a constructor to zero or more terms and possibly to some parameters. At the level of types, we identify the term constructor with a functor $F$ of arity $m+1$, the variables with a type $V$ and the parameters with some types $X_{i \in m}$. We suppose that all types $V, \overline{X}$ support equality, that is, there exist a function $eq_V : V \to V \to \mathit{Bool}$ and for each $i \in m$ there is a function $\text{eq}_i : X_i \to X_i \to \mathit{Bool}$. We now describe a functional unification algorithm that works for all such $F, V$ and $\overline{X}$. A list of the functions used is given in Figures 5.7, 5.8.

#### 5.4.2.1 Unification — ML-like version

In this section we describe the unification algorithm using ML-like datatypes for the type of terms and the type of lists:

- $LX = \text{Nil} \mid \text{Cons } \mathit{of}\ X * LX$
- $\mathit{Term} = \text{Var } \mathit{of}\ V \mid \text{Cns } \mathit{of}\ F(\overline{X}, \mathit{Term})$

We use the following abbreviations: let $\mathit{Pair} = \mathit{Term}^2$ and let $\mathit{Pairs} = L(\mathit{Pair})$. Moreover let $\mathit{Subst} = V \to \mathit{Term}$ be the type of substitutions, that is the functions from variables to terms. Also, $\mathsf{id}_{subst}$ is the identity substitution, defined as $\mathsf{id}_{subst}\ v = \mathsf{var2term}\ v$.

The function

$\mathsf{unify} : \mathit{Pairs} \to M(\mathit{Subst})$

$\mathsf{unify}\ \langle t_1, t_2 \rangle = \mathsf{unifyl}\ (\text{single}\langle t_1, t_2 \rangle)\ \mathsf{id}_{subst}$

takes two terms and gives back the most general unifier if it exists. It is implemented using:

$\mathsf{unifyl} : PairsFV, \overline{X} \to M(Subst)$

$\mathsf{unifyl}\ (\mathrm{Nil})\ s = \mathrm{ok}\ s$

$\mathsf{unifyl}\ (\mathrm{Cons}\ y\ ys)\ s = \mathrm{let}\ z \Leftarrow \mathsf{step}\langle \mathrm{Cons}\ y\ ys, s\rangle\ \mathrm{in}\ \mathsf{unifyl}\ (\pi_0\ z)(\pi_1\ z)$

which takes a list of pair of terms and updates the current substitution, given as an extra argument.

**The step function.** The core of the unification algorithm is the function step which is iterated until the list of "pairs of terms" to be unified is empty. The termination of the algorithm is guaranteed since on each pass either the number of free variables in the list of pairs decreases, or the number of constructors in the list of pairs decreases but the number of variables doesn't increase.

$\mathsf{step} : Pairs \times Subst \to M(Pairs \times Subst)$

$\mathsf{step}\ \langle \mathrm{Nil}, s\rangle = \mathrm{ok}\ \langle \mathrm{Nil}, s\rangle$

$\mathsf{step}\ \langle \mathrm{Cons}\ y\ ys, s\rangle = \mathsf{Cases}\ y\ ys\ s$

This function takes a list of pairs of terms and a substitution. If the list is empty then it does nothing. If the list has a head, then it decomposes the head in such a way it is possible to distinguish four cases depending on whether or not the terms are variables.

The function

$\mathsf{Cases} : Pair \to Pairs \to Subst \to M(Pairs \times Subst)$

$\mathsf{Cases}\ \langle \mathrm{Var}\ v_0, \mathrm{Var}\ v_1\rangle\ l\ s = \mathsf{VarVarCase}\ v_0\ v_1\ \langle l, s\rangle$

$\mathsf{Cases}\ \langle \mathrm{Var}\ v_0, \mathrm{Cns}\ w_1\rangle\ l\ s = \mathsf{VarCnsCase}\ v_0\ w_1\ \langle l, s\rangle$

$\mathsf{Cases}\ \langle \mathrm{Cns}\ w_0, \mathrm{Cns}\ v_1\rangle\ l\ s = \mathsf{VarCnsCase}\ v_1\ w_0\ \langle l, s\rangle$

$\mathsf{Cases}\ \langle \mathrm{Var}\ w_0, \mathrm{Cns}\ w_1\rangle\ l\ s = \mathsf{CnsCnsCase}\ w_0\ w_1\ \langle l, s\rangle$

determines the cases, which are handled below.

$\mathsf{VarVarCase} : V \to V \to Pairs \times Subst \to M(Pairs \times Subst)$

$$\begin{aligned}\mathsf{VarVarCase}\ v_0\ v_1\ \langle l, s\rangle \;=\;& \mathrm{if}\ \mathrm{eq}_V\ v_0\ v_1\\ & \mathrm{then}\ \mathrm{ok}\ \langle l, s\rangle\\ & \mathrm{else}\ \mathrm{ok}\ (\mathsf{subUpdate}\ v_0\ (\mathrm{Var}\ v_1)\ l\ s)\end{aligned}$$

The function VarVarCase deals with the case in which both components are variables. If they are equal it does nothing, otherwise it augments the substitution and applies the new substitution to all the terms in the list.

$\mathsf{VarCnsCase} : V \rightarrow F(\overline{X}, \mathit{Term}) \rightarrow \mathit{Pairs} \times \mathit{Subst} \rightarrow M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{VarCnsCase}\ v\ w\ \langle l, s\rangle \ = \ & \text{if } \mathsf{occCheck}\ v\ (\mathrm{Cns}\ w)\\ & \text{then fail}\\ & \text{else ok } (\mathsf{subUpdate}\ v\ (\mathrm{Cns}\ w)\ l\ s)\end{aligned}$$

If either component is a variable then VarCnsCase does the usual occurs check, then it augments the substitution and applies the new substitution to all the terms in the list.

$\mathsf{CnsCnsCase} : F(\overline{X}, \mathit{Term}) \rightarrow F(\overline{X}, \mathit{Term}) \rightarrow \mathit{Pairs} \times \mathit{Subst} \rightarrow M(\mathit{Pairs} \times \mathit{Subst})$

$\mathsf{CnsCnsCase}\ w_0\ w_1\ \langle l, s\rangle \ =$

$\text{let } z \Leftarrow (\mathrm{zipop}'_{m+1}\ \mathrm{eq}_{i:m}\ \mathrm{pair}\ w_0\ w_1) \text{ in ok } \langle \mathrm{append}\ (\mathrm{extract}_{m+1,m}\ z)\ l\ , s\rangle$

Otherwise, by using $\mathrm{zipop}'_{m+1}$, CnsCnsCase compares the top level of the two terms and if it succeeds, it extracts the subterm pairs and updates the list.

**Auxiliary functions.** In this paragraph we define the auxiliary functions used in the implementation of step.

$\mathsf{occCheck} : V \rightarrow \mathit{Term} \rightarrow \mathit{Bool}$

$\mathsf{occCheck}\ v\ t = \mathsf{occCLml}\ v\ (\mathrm{single}\ t)$

This is implemented in terms of occCLml that takes a list of terms instead of a term.

$\mathsf{occCLml} : V \rightarrow L(\mathit{Term}) \rightarrow \mathit{Bool}$

$\mathsf{occCLml}\ v\ \mathrm{Nil} = \bot$

$\mathsf{occCLml}\ v\ (\mathrm{Cons}\ (\mathrm{Var}\ v_0)\ ys) = (\mathrm{eq}_V\ v\ v_0) \text{ or } (\mathsf{occCLml}\ v\ ys)$

$\mathsf{occCLml}\ v\ (\mathrm{Cons}\ (\mathrm{Cns}\ w)\ ys) = \mathsf{occCLml}\ v\ (\mathrm{append}\ (\mathrm{extract}_{m+1,m}\ w)\ ys)$

This function checks if a variable occurs in a list of terms. If the list of terms is not empty, it takes the term in the head of the list and if this term is a variable compare the two variables and continue the check on the rest of the list, otherwise the list of variables occurring in that term is extracted and added to the list of terms.

$\mathsf{subUpdate} : V \rightarrow \mathit{Term} \rightarrow \mathit{Pairs} \rightarrow \mathit{Subst} \rightarrow \mathit{Pairs} \times \mathit{Subst}$

$\mathsf{subUpdate}\ v\ t\ l\ s \ = \langle \mathsf{subInst}\ v\ t\ l, \mathsf{subComp}\ v\ t\ s\rangle$

This function pairs subInst and subComp.

$\mathsf{subComp} : V \rightarrow \mathit{Term} \rightarrow \mathit{Subst} \rightarrow \mathit{Subst}$

$\mathsf{subComp}\ v\ t\ s\ w = \text{if } \mathrm{eq}_V\ w\ v \text{ then } t \text{ else } \mathsf{subst}\ v\ t\ (s\ w)$

This function updates a substitution, once it is given a pair variable-term (elementary substitution).

$\mathsf{subInst} : V \to \mathit{Term} \to \mathit{Pairs} \to \mathit{Pairs}$

$\mathsf{subInst}\ v\ t\ \mathrm{Nil} = \mathrm{Nil}$

$\mathsf{subInst}\ v\ t\ (\mathrm{Cons}\ y\ ys) = \mathrm{Cons}\ \langle \mathsf{subst}\ v\ t\ (\pi_0\ y), \mathsf{subst}\ v\ t\ (\pi_1\ y)\rangle\ (\mathsf{subInst}\ v\ t\ ys)$

This function applies an elementary substitution to a list of pairs of terms.

$\mathsf{subst} : V \to \mathit{Term} \to \mathit{Term} \to \mathit{Term}$

$\mathsf{subst}\ v\ t\ (\mathrm{Var}\ v) = \mathrm{if}\ \mathrm{eq}_V\ x\ v\ \mathrm{then}\ t\ \mathrm{else}\ \mathrm{Var}\ v$

$\mathsf{subst}\ v\ t\ (\mathrm{Cns}\ w) = \mathrm{Cns}\ (\mathrm{map}_{m+1}\mathrm{id}_{i:m}\ (\mathsf{subst}\ v\ t\ w))$

This applies an elementary substitution to a term.

#### 5.4.2.2 Unification — pure FML version

To define the FML-type corresponding to the ML-like datatypes defined in the previous section we need to define the following functors and types:

- $\mathit{Tu} = +\langle \Pi_0^{m+2}, F\langle \Pi_1^{m+2}, \ldots, \Pi_{m+1}^{m+2}\rangle^{m+2}\rangle^{m+2} : m+2$ is the functor that maps types $V, \overline{X}$ and $Y$ to $V + F(\overline{X}, Y)$.
- $T = \mu \mathit{Tu} : m+1$.
- $\mathit{Term} = T(V, \overline{X})$ is the type of terms with variables in $V$ and term constructors given by the functor $F$.
- $\mathit{Subst} = V \to \mathit{Term}$ is the type of substitutions, that is the functions from variables to terms.
- $\mathit{Pair} = \mathit{Term}^2$.
- $\mathit{Pairs} = L(\mathit{Pair})$.

For lists we use the list functor, $L$ in section 5.3.2.2. Recall that $L = \mu Lu$ where $Lu = +\langle 1\langle\rangle^2, \times\rangle^2$ and that the usual operations on lists are nil, cons, single, append, flatten.

The function $\mathsf{unify}$ is defined as in the previous ML-like version:

$\mathsf{unify} : \mathit{Pair} \to M(\mathit{Subst})$

$\mathsf{unify}\ \langle t_1, t_2\rangle = \mathsf{unifyl}\ (\mathrm{single}\langle t_1, t_2\rangle)\ \mathsf{id}_{subst}$

However, the function $\mathsf{unifyl}$ now needs to be redefined, since the ML-like implementation uses a general recursive constructor whereas in FML we have only primitive recursion.

$\mathsf{unifyl} : \mathit{Pairs} \to \mathit{Subst} \to M(\mathit{Subst})$

$\mathsf{unifyl}\ l\ s\ = \mathrm{let}\, p \Leftarrow \mathsf{outIter}\ (\mathsf{varNumL}\ l)\ \langle l, s\rangle\, \mathrm{in}\, \mathrm{ok}\ (\pi_1\ p)$

The implementation now requires two loops to be iterated. The outermost is bounded by the number of variables occurring in the list of pairs (for simplicity we take the number of occurrences), the innermost by the number of constructors occurring in the list of pairs. In fact, in order to guarantee that on each outermost iteration the number of variables decreases, we have only to iterate the basic step at most a number of times equal to the number of term constructors in the list. This occurs when the two terms have the same structure.

The function

$\mathsf{outIter} : L(1) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{outIter} = \mathrm{fold}_1(\lambda z.\ & \mathrm{case}\ \mathsf{flatunflist}\ z\ \mathrm{of}\\ & \mathrm{in}_0\ x \Rightarrow \mathrm{ok}\\ & \mathrm{in}_1\ y \Rightarrow \mathsf{innIter}(\mathsf{cnsNumL}\ l))\end{aligned}$$

implements the outer loop, whereas the function

$\mathsf{innIter} : L(1) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{innIter} = \mathrm{fold}_1(\lambda z.\ & \mathrm{case}\ \mathsf{flatunflist}\ z\ \mathrm{of}\\ & \mathrm{in}_0\ x \Rightarrow \mathrm{ok}\\ & \mathrm{in}_1\ y \Rightarrow \mathsf{step})\end{aligned}$$

implements the inner loop.

**Counting variables and constructors.** The natural number type is implemented as $L(1)$ so iteration can be mimicked by folding on a list.

$\mathsf{varNumL} : \mathit{Pairs} \to L(1)$

$\mathsf{varNumL} = (\mathrm{map}_1\ \mathrm{bang}) \circ \mathrm{extract}_{m+1,0} \circ \mathrm{pack}_{2,m+1} \circ \mathrm{pack}_{1,2}$

The number of variable occurrences in the list of pairs is computed by the function $\mathrm{extract}_{m+1,0}$ (defined in Section 5.3.3), which extracts the data of type $V$ from the data structure $L\langle\times\langle T, T\rangle\rangle(V, \overline{X})$. Note that $(\mathrm{pack}_{2,m+1} \circ \mathrm{pack}_{1,2})$ coerces $L(\mathit{Term}^2)$ to $L\langle\times\langle T, T\rangle\rangle(V, \overline{X})$.

$\mathsf{cnsNumL} : \mathit{Pairs} \to L(1)$

$$\begin{aligned}\mathsf{cnsNumL} = \mathrm{fold}_1(lz.\ & \text{case } \mathsf{flatunflist}\ z \text{ of}\\ & \mathrm{in}_0\ x \Rightarrow \mathrm{nil}\\ & \mathrm{in}_1\ \langle y, ys\rangle \Rightarrow\\ & \text{append (append } (\mathsf{cnsNum}(\pi_0 y))\ (\mathsf{cnsNum}(\pi_1 y)))\ ys)\end{aligned}$$

The number of constructors in the list of pairs is obtained by adding the number of constructors occurring in each term. This, in turn, is computed recursively adding the number of constructors of the subterms.

$\mathsf{cnsNum} : \forall G : m.\mu G(\overline{X}) \to L(1)$

$\mathsf{cnsNum} = \mathrm{fold}_{m+1}(lz.\text{append (single } un)(\text{flatten } (\mathrm{extract}_{m+1,m}\ z)))$

This function uses the $\mathrm{extract}_{m+1,m}$ function to locate the constructors inside a term.

**The step function.** The core of the algorithm is given by the function step which reduces to cases, as before.

$\mathsf{step} : \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{step}\ \langle l, s\rangle = \ & \text{case } \mathsf{unfoldlist}\ l \text{ of}\\ & \mathrm{in}_0\ x \Rightarrow \mathrm{ok}\ \langle l, s\rangle\\ & \mathrm{in}_1\ \langle y, ys\rangle \Rightarrow \ \text{case } \mathsf{unfoldterm}(\pi_0 y) \text{ of}\\ & \qquad \mathrm{in}_0\ v_0 \Rightarrow \ \text{case } \mathsf{unfoldterm}(\pi_1 y) \text{ of}\\ & \qquad\qquad \mathrm{in}_0\ v_1 \Rightarrow \mathsf{VarVarCase}\ v_0\ v_1\ \langle ys, s\rangle\\ & \qquad\qquad \mathrm{in}_1\ w_1 \Rightarrow \mathsf{VarCnsCase}\ v_0\ w_1\ \langle ys, s\rangle\\ & \qquad \mathrm{in}_1\ w_0 \Rightarrow \ \text{case } \mathsf{unfoldterm}(\pi_1 y) \text{ of}\\ & \qquad\qquad \mathrm{in}_0\ v_1 \Rightarrow \mathsf{VarCnsCase}\ v_1\ w_0\ \langle ys, s\rangle\\ & \qquad\qquad \mathrm{in}_1\ w_1 \Rightarrow \mathsf{CnsCnsCase}\ w_0\ w_1\ \langle ys, s\rangle\end{aligned}$$

The following functions deal with the three cases:

$\mathsf{VarVarCase} : V \to V \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{VarVarCase}\ v_0\ v_1\ \langle l, s\rangle \ = \ & \text{if } \mathrm{eq}_V\ v_0\ v_1\\ & \text{then ok } \langle l, s\rangle\\ & \text{else ok } (\mathsf{subUpdate}\ v_0\ (\mathsf{var2term}\ v_1)\ l\ s)\end{aligned}$$

$\mathsf{VarCnsCase} : V \to F(\overline{X}, \mathit{Term}) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{VarCnsCase}\ v\ w\ \langle l, s\rangle \ = \ & \text{if } \mathsf{occCheck}\ v\ (\mathsf{cns2term}\ w)\\ & \text{then fail}\\ & \text{else ok } (\mathsf{subUpdate}\ v\ (\mathsf{cns2term}\ w)\ l\ s)\end{aligned}$$

$\mathsf{CnsCnsCase} : F(\overline{X}, \mathit{Term}) \to F(\overline{X}, \mathit{Term}) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$\mathsf{CnsCnsCase}\ w_0\ w_1\ \langle l, s\rangle\ =$

$\mathrm{let}\, z \Leftarrow (\mathrm{zipop}_{m+1}\ \mathrm{eq}_{i:m}\ \mathrm{pair}\ w_0\ w_1)\, \mathrm{in}\, \mathrm{ok}\ \langle \mathrm{append}\ (\mathrm{extract}_{m+1,m}\ z)\ l\ , s\rangle$

**Auxiliary functions.** In this paragraph we define the auxiliary functions used in the implementation of $\mathsf{step}$.

$\mathsf{occCheck} : V \to \mathit{Term} \to \mathit{Bool}$

$\mathsf{occCheck}\ v\ t = \mathsf{occCheckL}\ v\ (\mathrm{extract}_{m+1,0}\ t)$

This function checks if a variable occurs in a term, by checking if the variable occurs in the list of the variables extracted from the term.

$\mathsf{occCheckL} : X \to L(X) \to \mathit{Bool}$

$$\begin{array}{ll}\mathsf{occCheckL}\ a = \mathrm{fold}_1(\lambda z. & \mathrm{case}\ \mathsf{flatunflist}\ z\ \mathrm{of} \\ & \mathrm{in}_0\ x \Rightarrow \bot \\ & \mathrm{in}_1\ \langle y, ys\rangle \Rightarrow\ \ \begin{array}[t]{l}\mathrm{if}\ \mathrm{eq}_X\ y\ x \\ \mathrm{then}\ \top \\ \mathrm{else}\ y)\end{array}\end{array}$$

This function checks if a value occurs in a list.

$\mathsf{subUpdate} : V \to \mathit{Term} \to \mathit{Pairs} \to \mathit{Subst} \to \mathit{Pairs} \times \mathit{Subst}$

$\mathsf{subUpdate}\ v\ t\ l\ s\ = \langle \mathsf{subInst}\ v\ t\ l, \mathsf{subComp}\ v\ t\ s\rangle$

This function pairs the functions $\mathsf{subInst}$ and $\mathsf{subComp}$.

$\mathsf{subComp} : V \to \mathit{Term} \to \mathit{Subst} \to \mathit{Subst}$

$$\begin{array}{ll}\mathsf{subComp}\ v\ t\ s\ = \lambda w. & \mathrm{if}\ \mathrm{eq}_V\ w\ v \\ & \mathrm{then}\ t \\ & \mathrm{else}\ \mathsf{subst}\ v\ t\ (s\ w)\end{array}$$

This function updates a substitution, once it is given a pair variable-term (elementary substitution).

$\mathsf{subInst} : V \to \mathit{Term} \to \mathit{Pairs} \to \mathit{Pairs}$

$$\begin{array}{ll}\mathsf{subInst}\ v\ t = \mathrm{fold}_1(\lambda z. & \mathrm{case}\ \mathsf{flatunflist}\ z\ \mathrm{of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{nil} \\ & \mathrm{in}_1\ \langle y, ys\rangle \Rightarrow \mathit{cons}\ \langle \mathsf{subst}\ v\ t\ (\pi_0\ y), \mathsf{subst}\ v\ t\ (\pi_1\ y)\rangle\ ys)\end{array}$$

This function applies an elementary substitution to a list of pairs of terms.

$\mathsf{subst} : V \to \mathit{Term} \to \mathit{Term} \to \mathit{Term}$

$$\begin{aligned}\mathsf{subst}\ v\ t = \mathrm{fold}_m(\lambda z.\ & \text{case } \mathsf{flatunfterm}\ z \text{ of} \\ & \mathrm{in}_0\ x \Rightarrow\ \ \text{if } \mathrm{eq}_V\ x\ v \\ & \qquad\qquad \text{then } t \\ & \qquad\qquad \text{else } \mathsf{var2term}\ x \\ & \mathrm{in}_1\ y \Rightarrow \mathsf{cns2term}\ y)\end{aligned}$$

This function applies an elementary substitution to a term.

**Coercion functions.** The following functions operate a coercion between types. They make large use of the canonical functor constructors and destructors.

$\mathsf{var2term} : V \to \mathit{Term}$

$\mathsf{var2term} = \mathrm{wrap}_{m+1} \circ \mathrm{pack}_{2,m+2} \circ \mathrm{in}_0 \circ \mathrm{tag}_{m+1,0}$

This function coerces a variable to a term.

$\mathsf{cns2term} : F(\overline{X}, \mathit{Term}) \to \mathit{Term}$

$\mathsf{cns2term} = \mathrm{wrap}_{m+1} \circ \mathrm{pack}_{2,m+2} \circ \mathrm{in}_1 \circ \mathrm{pack}_{m+1,m+2} \circ (\mathrm{map}_{m+1}(\mathrm{tag}_{m+1,i})_{i:[1..m+1]})$

This function coerces an applied constructor to a term.

$\mathsf{flatunfterm} : \mathit{Tu}(V, \overline{X}, Y) \to V + F(\overline{X}, Y)$

$$\begin{aligned}\mathsf{flatunfterm}\ z = \ & \text{case } (\mathrm{unpack}_{2,m+2}\ z) \text{ of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{in}_0\ (\mathrm{untag}_{m+2,0}\ x) \\ & \mathrm{in}_1\ y \Rightarrow \mathrm{unpack}_{m+1,m+2}\ (\mathrm{map}_{m+1}(\mathrm{untag}_{m+2,i})_{i:[1..m+1]}\ y)\end{aligned}$$

This function coerces the type $\mathit{Tu}(V, \overline{X}, Y)$ to the type $V + F(\overline{X}, Y)$.

$\mathsf{unfoldterm} : \mathit{Term} \to V + F(\overline{X}, \mathit{Term})$

$\mathsf{unfoldterm} = \mathsf{flatunfterm} \circ \mathrm{unwrap}_{m+1}$

This function unfolds a term and coerces it to the type $V + F(\overline{X}, \mathit{Term})$.

$\mathsf{flatunflist} : \mathit{Lu}(X, Y) \to 1 + X \times Y$

$$\begin{aligned}\mathsf{flatunflist}\ z = \ & \text{case } (\mathrm{unpack}_{2,2}\ z) \text{ of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{in}_0\ (\mathrm{unpack}_{0,2}\ x) \\ & \mathrm{in}_1\ y \Rightarrow \mathrm{in}_1\ y\end{aligned}$$

This function coerces the type $\mathit{Lu}(X, Y)$ to the type $1 + X \times Y$.

$\mathsf{unfoldlist} : L(X) \to 1 + X \times L(X)$

$\mathsf{unfoldlist} = \mathsf{flatunflist} \circ \mathrm{unwrap}_1$

This function unfolds a list and coerces it to the type $1 + X \times L(X)$.

## 5.5 Variations and extensions to FML

FML provides a general framework for studying a new form of polymorphism centered around the notion of *functor*. In fact, the key new feature of FML is the introduction of functor expressions, rather than the new form of polymorphism. In designing FML we have intentionally kept the interaction between functors and types to a minimum, in particular the language of functor expressions is given independently from that of types and terms. As shown in Section 5.1.5, functor expressions can be interpreted in many possible ways, and none of them is a priori preferable than others. In this section we propose possible variations and extensions to FML. They can be classified into three groups, according to the source of their motivation.

- type theory: for instance, choosing between terms à la Curry or à la Church, identifying types and type schema (as in system $F$), or introducing higher order kinds (as in system $F\omega$).
- semantics: for instance, $\mathrm{FML}_{shape}$ (see Section 5.3.3).
- programming languages: for instance, various forms of polymorphism, recursive definitions, or polytypic definitions.

First we give a brief overview of generic, type-theoretic variations and extensions, and then we discuss in greater details two programming language extensions: recursive definitions of polymorphic expressions, and polytypic definitions. These extensions will be embodied in $\mathrm{FML}_{poly}$ in Section 5.5.3.

Both extensions exploit in an essential way shape polymorphism, i.e. quantification over functors, and moreover they require another form of polymorphism, namely quantification over functor arities $m : N$. A formal account of quantification over functor arities involves some technical subtleties with dependent types (well-known to people working on logical frameworks based on type theory), which are beyond the scope of this paper. Instead we give an informal presentation, in which the finitary nature of syntax and inference rules is lost.

In both versions:

- unify : $Pair \to M(Subst)$
- unifyl : $Pairs \to Subst \to M(Subst)$
- step : $Pairs \times Subst \to M(Pairs \times Subst)$
- VarVarCase : $V \to V \to Pairs \times Subst \to M(Pairs \times Subst)$
- VarCnsCase : $V \to F(\overline{X}, Term) \to Pairs \times Subst \to M(Pairs \times Subst)$
- CnsCnsCase : $F(\overline{X}, Term) \to F(\overline{X}, Term) \to Pairs \times Subst \to M(Pairs \times Subst)$
- occCheck : $V \to Term \to Bool$
- occCheckL : $X \to L(X) \to Bool$
- subUpdate : $V \to Term \to Pairs \to Subst \to Pairs \times Subst$
- subComp : $V \to Term \to Subst \to Subst$
- subInst : $V \to Term \to Pairs \to Pairs$
- subst : $V \to Term \to Term \to Term$

Where:

- $Subst = V \to Term$
- $Pair = Term^2$
- $Pairs = L(Pair)$

Figure 5.7: Types of the functions used in the unification algorithm

In ML-like version:

- Cases : $Pair \to Pairs \to Subst \to M(Pairs \times Subst)$
- occCLml : $V \to L(Term) \to Bool$

In pure FML version:

- outIter : $L(1) \to Pairs \times Subst \to M(Pairs \times Subst)$
- innIter : $L(1) \to Pairs \times Subst \to M(Pairs \times Subst)$
- varNumL : $Pairs \to L(1)$
- cnsNumL : $Pairs \to L(1)$
- cnsNum : $\forall G : m + 1.\mu G(\overline{X}) \to L(1)$
- var2term : $V \to Term$
- cns2term : $F(\overline{X}, Term) \to Term$
- flatunfterm : $Tu(V, \overline{X}, Y) \to V + F(\overline{X}, Y)$
- unfoldterm : $Term \to V + F(\overline{X}, Term)$
- flatunflist : $Lu(X, Y) \to 1 + X \times Y$
- unfoldlist : $L(X) \to 1 + X \times L(X)$

Where:

- $Subst = V \to Term$
- $Pair = Term^2$
- $Pairs = L(Pair)$

Figure 5.8: Types of the functions used in the unification algorithm - cont.

### 5.5.1 Type-theoretic variations and extensions

We consider some simple variations on terms which make sense for other typed calculi. They are: terms à la Church; some extensions suggested by system $F$ and $F\omega$ as extensions of ML; and an extension of FML with the inductive kind of natural numbers.

#### 5.5.1.1 Terms à la Church

The syntax of terms à la Curry is parameterized with respect to constants $c$

$$t \quad ::= \quad x \mid c \mid \lambda x.t \mid t_1\ t_2 \mid \mathrm{let}\ x = t_1\ \mathrm{in}\ t_2$$

A more general approach is to parameterize terms with respect to binders $c$

$$\begin{array}{lcll} t & ::= & x \mid c(\overline{f}) & \text{terms} \\ f & ::= & [\overline{x}].t & \text{abstractions} \end{array}$$

Using abstraction and application one could replace binders with higher-order constants, but binders are preferable in the absence of functional types. For instance, application of the constant $\mathrm{map}_m$ for mapping applied to some $\lambda x.t_i$ and $t$ is replaced by the binder $\mathrm{map}_m(([x].t_i)_{i:m}, t)$.

Another variation is to use terms à la Church, i.e. with explicit type information

$$t \quad ::= \quad x \mid c \mid \lambda x : \tau.t \mid t_1\ t_2 \mid \Lambda X : \mathrm{T}.t \mid t\ \tau \mid \Lambda X : m.t \mid t\ F \mid \mathrm{let}\ x : \sigma = t_1\ \mathrm{in}\ t_2$$

Terms à la Church are rather cumbersome to write, but the extra information can be very useful, as advocated by [HM95, AFH94, Tol94]. For instance, terms à la Church are essential to describe the most general pattern for polytypic definitions, where functor information is used (as in ad hoc polymorphism) to choose among incompatible alternatives. Terms à la Church are also useful to ensure decidability of type-checking in some extensions of FML.

#### 5.5.1.2 $F$-like extensions

By analogy with system $F$ we could identify types and type schema i.e.

$$\tau \quad ::= \quad X \mid F(\overline{\tau}) \mid \tau_1 \rightarrow \tau_2 \mid \forall X : \mathrm{T}.\tau \mid \forall X : m.\tau$$

(One could also contemplate identifying types with functors of arity 0 but this would make the languages of functor expressions dependent on that of type expressions.) A more substantial extension, which could be called $FF\omega$, is suggested by system $F\omega$. In this extension one can make explicit the level of kinds $k$, which include T and functor arities $m$, and add higher order kinds

$$\begin{array}{rcl} k & ::= & \mathrm{T} \mid m \mid k_1 \to k_2 \\ u & ::= & X \mid C \mid \Lambda X : k.u \mid u_1\ u_2 \\ t & ::= & x \mid c \mid \lambda x : \tau.t \mid t_1\ t_2 \mid \Lambda X : k.t \mid t\ u \end{array}$$

Here $u$ represents constructors, and $t$ terms. In $FF\omega$ functor constructors and the type constructor $\to$ can be viewed as constants $C$ of higher order kind, and functors of kind $m$ coexist with $m$-ary type constructors of kind $\mathrm{T}^m \to \mathrm{T}$.

#### 5.5.1.3 Inductive kinds

A more novel extension is to add polymorphism over functor arities. In FML it would support a single combinator for mapping

$$\mathrm{map} \quad : \quad \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m} : \mathrm{T}.(X_i \to Y_i)_{i:m} \to F(\overline{X}) \to F(\overline{Y})$$

This requires quantification over arities in type schema

$$\sigma \quad ::= \quad \ldots \mid \forall m : N.\sigma_m$$

with the corresponding terms and reductions. A formal presentation of arity quantification would involve rules such as

$$\forall_N \ \frac{\Delta, m : N \vdash \sigma}{\Delta \vdash \forall m : N.\sigma} \ .$$

However, one would have to spell out what are the expressions of kind $N$, and how they could be used in other syntactic categories. This could be done in the setting of Martin-Löf type theory, by taking $N$ as the inductive kind of natural numbers, which could then be used to define kinds, constructors and terms by induction. For instance, in system $F\omega$ we could define the family of kinds $\mathrm{T}^m \to \mathrm{T}$ by induction on $m : N$.

Here we give an informal description based on the following rule:

$$\forall_N \ \frac{\Delta \vdash \sigma_m \quad m : N}{\Delta \vdash \forall m : N.\sigma_m} \ .$$

This rule says that given an infinite family of type schema, we can form a type schema (of infinite size!). Of course, we can arbitrarily restrict the arities to any particular limit, e.g. 3 to recover finitariness and formality.

The syntax of terms and the typing rules are extended as follows

$$t \quad ::= \quad \ldots \mid (\Lambda m.t_m) \mid t_e$$

where $m$ is a numeric variable, and $e$ is a numeric expression.

$$\text{App}_N \ \frac{\Delta; \Gamma \vdash t : \forall m : N.\sigma_m}{\Delta; \Gamma \vdash t_e : \sigma_e}$$

$$\Lambda_N \ \frac{\Delta; \Gamma \vdash t_m : \sigma_m \quad m : N}{\Delta; \Gamma \vdash (\Lambda m.t_m) : \forall m : N.\sigma_m}$$

The reduction rule for these term-constructors is

$$(\Lambda m.t_m)_e \quad > \quad t_e$$

#### 5.5.1.4 Recursion

There are two ways of adding recursive definitions to ML. The first option, taken in (lazy) ML, is to introduce a fix-point combinator

$$\text{fix} : \forall X : \text{T}.(X \to X) \to X$$

with the usual reduction

$$\text{fix } t > t \ (\text{fix } t).$$

The addition of such combinator does not affect the type-inference algorithm.

The second option, taken in $\text{ML}^+$ (see [KT92]), allows polymorphic recursion with the usual unfolding as reduction:

$$\text{rec} \ \frac{\Delta; \Gamma, x : \sigma \vdash t : \sigma}{\Delta; \Gamma \vdash (\mu x.t) : \sigma}$$

$$(\mu x.t) > [(\mu x.t)/x]t.$$

This second option is strictly more powerful (i.e. more terms are typable), but typability becomes undecidable. Of course, when types and type schema are identified (as in system $F$), the two extensions are equivalent.

In FML we adopt the second option. To achieve decidable type checking in a (formal presentation of) this extension, it is better to adopt a formulation of FML à la Church, i.e. with explicit type and functor information in terms.

### 5.5.2 Polytypic definitions

The combinators for mapping, folding, traversing, zipping, etc. capture fundamental properties common to large classes of functors. It is obviously desirable that programmers be able to construct these and other examples directly, using recursive definitions. In ML + fix one can define primitive recursion on inductive datatypes in terms of pattern matching and general recursion. By analogy, one might expect that in FML + fix one could define $\mathrm{fold}_m$ and $\mathrm{map}_m$ in terms of pattern matching for inductive types given by

$$\mathrm{match}^{\mu}_{m} \quad : \quad \forall F : m+1.\forall X_{i:m}, Y : \mathrm{T}.(F(\overline{X}, \mu F(\overline{X})) \to Y) \to \mu F(\overline{X}) \to Y$$

$$\mathrm{match}^{\mu}_{m}\ f\ (\mathrm{intro}^{\mu}_{m}\ t) \quad > \quad f\ t$$

Indeed, one can define $\mathrm{fold}_m$ and $\mathrm{map}_m$ for any closed functor expression $F$ (by induction on the structure of $F$), but there is no way to define them *uniformly* for every functor $F$. There are two reasons why fix does not suffice for defining a combinator like $\mathrm{map}_m$. First, we need mutually recursive definitions of families of combinators $c_{m:N}$ indexed by the natural numbers, since $\mathrm{map}_m$ is defined in terms of $\mathrm{map}_n$. Second, we need induction on the structure of inductive functors, i.e. polytypic definitions [She93, HS93, Jeu95, Mee96, JJ97].

For the first, we need inductive kinds, so that the type of a polytypic program can be presented as $\forall m : N.\forall F : m.\sigma_m(F)$. For the second, we require a new term formation rule poly $[-]$ to perform case analysis on functors. Finally, to unleash the power of this rule we require the stronger, polymorphic recursion.

The polytypic construction must cover the following four possibilities:

- constant functors $C$
- projection functors $\Pi^m_i$
- compositions $F\langle\overline{G}\rangle^m$ of functors
- initial algebra functors $\mu F$.

Here is the corresponding rule.

$$\text{poly}\ \frac{\begin{array}{lr} \Delta;\Gamma \vdash t_C : \sigma_{n_C}(C) & C \\ \Delta;\Gamma \vdash t^{\Pi}_{m,i} : \sigma_m(\Pi^m_i) & i : m : N \\ \Delta, F : n, G_{i:n} : m;\Gamma \vdash t^{\circ}_{n,m} : \sigma_m(F\langle\overline{G}\rangle^m) & m, n : N \\ \Delta, F : m+1;\Gamma \vdash t^{\mu}_{m} : \sigma_m(\mu F) & m : N \end{array}}{\Delta;\Gamma \vdash \text{poly}\ \left[t_C, t^{\Pi}_{m,i}, t^{\circ}_{n,m}, t^{\mu}_{m}\right] : \forall m : N.\forall F : m.\sigma_m(F)}$$

Note that when applying this rule there must be a separate constant $t_C$ for each constant functor $C$, and that the other premises represent infinite families.

In the formulation of FML à la Curry, i.e. without explicit type and functor information in terms, it is impossible to have reduction rules for poly $\left[t_C, t^{\Pi}_{m,i}, t^{\circ}_{n,m}, t^{\mu}_{m}\right]$, which satisfy $SR$

(Subject Reduction) and other syntactic properties. For example, it is easy to construct a combinator $is_C : \forall m : N.\forall F : n.Bool$, which tests whether $F$ is the constant functor $C$. However, the most natural reductions ($is_C > \top$ and $is_C > \bot$) violate $CR$ (Church-Rosser).

Therefore, we restrict the typing rule (poly) by requiring $\sigma_m(F)$ to be of the form $\forall\Delta_m.F(\overline{\tau}) \to \ldots$, where $\Delta$ is a sequence of type and functor variables. The restriction to (poly) means that a polytypic definition must define a function, and its first argument, whose type is $F(\overline{\tau})$, has enough information for selecting the appropriate branch of the polytypic definition.

With this restriction the following reduction rules are compatible with the typing (we assume that the constant functors are only $1 : 0$, $\times : 2$ and $+ : 2$):

$$\begin{array}{rcl}
(\text{poly } \left[t_1, t_\times, t_+, t^\Pi, t^\circ, t^\mu\right])_0\ (\text{intro}^1) & > & t_1\ (\text{intro}^1) \\
(\text{poly } \left[t_1, t_\times, t_+, t^\Pi, t^\circ, t^\mu\right])_2\ (\text{intro}^x\ t_0\ t_1) & > & t_\times\ (\text{intro}^x\ t_0\ t_1) \\
(\text{poly } \left[t_1, t_\times, t_+, t^\Pi, t^\circ, t^\mu\right])_2\ (\text{intro}^+_j\ t) & > & t_+\ (\text{intro}^+_j\ t) \\
(\text{poly } \left[t_1, t_\times, t_+, t^\Pi, t^\circ, t^\mu\right])_m\ (\text{intro}^\Pi_{m,i}\ t) & > & t^\Pi_{m,i}\ (\text{intro}^\Pi_{m,i}\ t) \\
(\text{poly } \left[t_1, t_\times, t_+, t^\Pi, t^\circ, t^\mu\right])_m\ (\text{intro}^\circ_{n,m}\ t) & > & t^\circ_{n,m}\ (\text{intro}^\circ_{n,m}\ t) \\
(\text{poly } \left[t_1, t_\times, t_+, t^\Pi, t^\circ, t^\mu\right])_m\ (\text{intro}^\mu_m\ t) & > & t^\mu_m\ (\text{intro}^\mu_m\ t)
\end{array}$$

**Remark 5.16** The type-theoretic intuition behind the formation rule (poly) is that functors form an inductive family of types indexed by natural numbers. Therefore, one should consider similar formation rules for defining types, functors and type schema by induction on the structure of functors. For instance, definitions of types by induction on the structure of functors are needed to define the type constructor $C_F$ corresponding to the functor expression $F$ by unfolding composition, i.e. $C_{F\langle G_{i:m}\rangle^n}(\overline{X}) = C_F(C_{G_i}(\overline{X})_{i:m})$.

### 5.5.3 Polytypic definitions in FML$_{poly}$

FML$_{poly}$ is the extension of FML with the inductive, recursive and polytypic constructions of Sections 5.5.1.3, 5.5.1.4 and 5.5.2. In this section we demonstrate its expressive power by defining the polytypic combinators of FML$_{shape}$. For notational convenience, in the examples we write

$$\begin{array}{lll}
\text{let poly} & & \\
f_0\ \text{un} & = t_1 & | \\
f_2\ \langle x_0, x_1\rangle & = t_\times & | \\
f_2\ (\text{in}_0\ x) & = t_{+,0} & | \\
f_2\ (\text{in}_1\ x) & = t_{+,1} & | \\
f_m\ (\text{intro}^{\Pi}_{m,i}\ x) & = t^{\Pi}_{m,i} & | \\
f_m\ (\text{intro}^{\circ}_{n,m}\ x) & = t^{\circ}_{n,m} & | \\
f_m\ (\text{intro}^{\mu}_{m}\ x) & = t^{\mu}_{m} & | \\
\text{in}\ t & &
\end{array}$$

for

$$\text{let}\ f = \text{poly}\left[\begin{array}{ll}
\text{elim}^1\ t_1, & \\
\text{elim}^x\ (\lambda x_0, x_1.t_\times), & \\
\text{elim}^+\ (\lambda x_0.t_{+,0})\ (\lambda x_1.t_{+,1}), & \\
\text{elim}^{\Pi}_{m,i}(\lambda x.t^{\Pi}_{m,i}) & i : m : N, \\
\text{elim}^{\circ}_{n,m}(\lambda x.t^{\circ}_{n,m}) & m, n : N, \\
\text{match}^{\mu}_{m}(\lambda x.t^{\mu}_{m}) & m : N
\end{array}\right]\ \text{in}\ t.$$

**Remark 5.17** The above notational convention exactly matches the restriction imposed on (poly). There is a minor disuniformity in the definition of the polytypic let, since in all cases we use a destructor, except for $\mu F$, where we use the combinator $\text{match}^{\mu}_{m}$ for pattern matching. The reason is that in all other cases destructors are only doing pattern matching.

We show that the polytypic combinators $\text{map}_m$, $\text{ltraverse}_m$ and $\text{zipop}_m$ are definable in $\text{FML}_{poly}$. More precisely, for each polytypic combinator we give a closed term of $\text{FML}_{poly}$ such that: it has the same type schema as the combinator, and; the reduction rules for the combinator are derivable.

#### 5.5.3.1 Definability of $\text{map}_m$

The $\text{map}_m$ combinator is defined using an auxiliary combinator, whose only purpose is to rearrange the arguments to fit the restriction imposed by the typing rule (poly).

$$pm \quad : \quad \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m} : \text{T}.\ F(\overline{X}) \to (X_i \to Y_i)_{i:m} \to F(\overline{Y})$$

$$
\begin{array}{l}
\mu\ map.\ \text{let poly} \\
\quad \begin{array}{lll}
pm_0\ \text{un} & = \text{un} & | \\
pm_2\ \langle x_0, x_1\rangle & = \lambda f_0, f_1.\ \langle f_0\ x_0, f_1\ x_1\rangle & | \\
pm_2\ (\text{in}_0\ x) & = \lambda f_0, f_1.\ \text{in}_0\ (f_0\ x) & | \\
pm_2\ (\text{in}_1\ x) & = \lambda f_0, f_1.\ \text{in}_1\ (f_1\ x) & | \\
pm_m\ (\text{intro}^{\Pi}_{m,i}\ x) & = \lambda f_{i:m}.\ \text{intro}^{\Pi}_{m,i}\ (f_i\ x) & | \\
pm_m\ (\text{intro}^{\circ}_{n,m}\ x) & = \lambda f_{i:m}.\ \text{intro}^{\circ}_{n,m}\ (map_n\ (map_m\ \overline{f})_{j:n}\ x) & | \\
pm_m\ (\text{intro}^{\mu}_{m}\ x) & = \lambda f_{i:m}.\ \text{intro}^{\mu}_{m}\ (map_{m+1}\ \overline{f}\ (map_m\ \overline{f})\ x) & | \\
\text{in}\ (\Lambda m.\lambda f_{i:m}.\lambda x.\ pm_m\ x\ \overline{f}) & &
\end{array}
\end{array}
$$

As an example, here is the derivation of the mapping reduction

$$\text{map}_m\ f_{i:m}\ (\text{intro}^{\mu}_{m}\ t)\ \ >\ \ \text{intro}^{\mu}_{m}\ (\text{map}_{m+1}\ \overline{f}\ (\text{map}_m\ \overline{f})\ t)$$

using the reduction rules for recursive and polytypic definitions. In what follows, we identify $map$ with the term $(\mu\ map.\ldots)$, and $pm$ with its polytypic definition poly $pm_0$ un $=$ un $\ldots$, in which $map$ has been replaced by its recursive definition.

$$
\begin{array}{ll}
map_m\ f_{i:m}\ (\text{intro}^{\mu}_{m}\ t) > & (\mu \text{ and let}) \\
(\Lambda m.\lambda f_{i:m}.\lambda x.pm_m\ x\ \overline{f})_m\ \overline{f}\ (\text{intro}^{\mu}_{m}\ t) > & (\beta) \\
pm_m\ (\text{intro}^{\mu}_{m}\ t)\ \overline{f} > & (\text{poly and match}^{\mu}_{m}) \\
(\lambda x.\lambda f_{i:m}.\text{intro}^{\mu}_{m}\ (map_{m+1}\ \overline{f}\ (map_m\ \overline{f})\ x))\ t\ \overline{f} > & (\beta) \\
\text{intro}^{\mu}_{m}\ (map_{m+1}\ \overline{f}\ (map_m\ \overline{f})\ t). &
\end{array}
$$

#### 5.5.3.2 Definability of $\text{ltraverse}_m$

The $\text{ltraverse}_m$ combinator is defined using an auxiliary combinator

$$
\begin{array}{lll}
plt & : & \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m}, S : \mathrm{T}. \\
& & F(\overline{X}) \to (X_i \to P_S(Y_i))_{i:m} \to P_S(F(\overline{Y}))
\end{array}
$$

where $P_S$ is the parsing monad defined in Section 5.3.2.1.

$$
\begin{array}{l}
\mu\ lt.\ \text{let poly} \\
\quad \begin{array}{lll}
plt_0\ \text{un} & = [\text{un}] & | \\
plt_2\ \langle x_0, x_1\rangle & = \lambda f_0, f_1.\text{let}\, y_0 \Leftarrow f_0\ x_0 \,\text{in let}\, y_1 \Leftarrow f_1\ x_1 \,\text{in}\, [\langle y_0, y_1\rangle] & | \\
plt_2\ (\text{in}_0\ x) & = \lambda f_0, f_1.\text{let}\, y \Leftarrow f_0\ x \,\text{in}\, [\text{in}_0\ y] & | \\
plt_2\ (\text{in}_1\ x) & = \lambda f_0, f_1.\text{let}\, y \Leftarrow f_1\ x \,\text{in}\, [\text{in}_1\ y] & | \\
plt_m\ (\text{intro}^{\Pi}_{m,i}\ x) & = \lambda f_{i:m}.\text{let}\, y \Leftarrow f_i\ x \,\text{in}\, [\text{intro}^{\Pi}_{m,i}\ y] & | \\
plt_m\ (\text{intro}^{\circ}_{n,m}\ x) & = \lambda f_{i:m}.\text{let}\, y \Leftarrow lt_n (lt_m\ \overline{f})_{j:n}\ x \,\text{in}\, [\text{intro}^{\circ}_{n,m}\ y] & | \\
plt_m\ (\text{intro}^{\mu}_{m}\ x) & = \lambda f_{i:m}.\text{let}\, y \Leftarrow lt_{m+1}\ \overline{f}\ (lt_m\ \overline{f})\ x \,\text{in}\, [\text{intro}^{\mu}_{m}\ y] & | \\
\text{in}\ (\Lambda m.\lambda f_{i:m}.\lambda x.plt_m\ x\ \overline{f}) & &
\end{array}
\end{array}
$$

#### 5.5.3.3 Definability of $\mathrm{zipop}_m$

The $\mathrm{zipop}_m$ combinator is defined using an auxiliary combinator

$$\begin{aligned} pz \;:\; & \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m}, Z_{i:m}, S : \mathrm{T}. \\ & F(\overline{X}) \to (X_i \to Y_i \to P_S(Z_i))_{i:m} \to F(\overline{Y}) \to P_S(F(\overline{Z})) \end{aligned}$$

where $P_S$ is the parsing monad defined in Section 5.3.2.1.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mu\ zop.\ \text{let poly}} \\
\quad pz_0\ \text{un} & = \text{elim}^1\ ([\text{un}]) & | \\
\quad pz_2\ \langle x_0, x_1\rangle & = \begin{array}{l}\lambda f_0, f_1.\text{elim}^x(\lambda y_0, y_1. \\ \text{let } z_0 \Leftarrow f_0\ x_0\ y_0 \text{ in let } z_1 \Leftarrow f_1\ x_1\ y_1 \text{ in } [\langle z_0, z_1\rangle])\end{array} & | \\
\quad pz_2\ (\text{in}_0\ x) & = \lambda f_0, f_1.\text{elim}^+\ (\lambda y.\text{let } z \Leftarrow f_0\ x\ y \text{ in } [\text{in}_0\ z])\ (\lambda y.\text{fail}) & | \\
\quad pz_2\ (\text{in}_1\ x) & = \lambda f_0, f_1.\text{elim}^+\ (\lambda y.\text{fail})\ (\lambda y.\text{let } z \Leftarrow f_1\ x\ y \text{ in } [\text{in}_1\ z]) & | \\
\quad pz_m\ (\text{intro}^{\Pi}_{m,i}\ x) & = \lambda f_{i:m}.\text{elim}^{\Pi}_{m,i}\ (\lambda y.\text{let } z \Leftarrow f_i\ x\ y \text{ in } [\text{intro}^{\Pi}_{m,i}\ z]) & | \\
\quad pz_m\ (\text{intro}^{\circ}_{n,m}\ x) & = \begin{array}{l}\lambda f_{i:m}.\text{elim}^{\circ}_{n,m}(\lambda y. \\ \text{let } z \Leftarrow zop_n(zop_m\ \overline{f})_{j:n}\ x\ y \text{ in } [\text{intro}^{\circ}_{n,m}\ z])\end{array} & | \\
\quad pz_m\ (\text{intro}^{\mu}_{m}\ x) & = \begin{array}{l}\lambda f_{i:m}.\text{elim}^{\mu}_{m}(\lambda y. \\ \text{let } z \Leftarrow zop_{m+1}\ \overline{f}\ (zop_m\ \overline{f})\ x\ y \text{ in } [\text{intro}^{\mu}_{m}\ z])\end{array} & | \\
\quad \text{in } (\Lambda m.\lambda f_{i:m}.\lambda x, y.pz_m\ x\ \overline{f}\ y) & &
\end{array}$$

#### 5.5.3.4 Limitations of $\mathrm{FML}_{poly}$ à la Curry

We give some examples of polytypic functions that are definable in $\mathrm{FML}_{poly}$ with the unrestricted (poly) rule, but cannot be defined using the restricted (poly) rule, where $\sigma_m(F) \equiv \forall\Delta.F(\overline{\tau}) \to \ldots$ and $\Delta$ is a sequence of type and functor variables. These examples identify clearly the limitations at the level of polytypic definitions, when terms have no explicit type and functor information.

**Example 5.18** Consider the polytypic function $is_C : \forall m : N.\forall F : m.Bool$, which tests whether $F$ is equal to $C$. Clearly, we cannot define it using the restricted (poly) rule. However, we could add an extra argument of type $F(\overline{1})$, so that the restricted (poly) rule become applicable and we have enough information about $F$.

This trick does not get very far, e.g. consider $occur_C : \forall m : N.\forall F : m.Bool$, which tests whether $C$ occurs in $F$. In this case we cannot get the necessary information about $F$ by inspecting a term of type $F(\overline{1})$, e.g. when $C = 1$ and $F$ is the functor $F(X) = X + 1$, more formally $F = +\langle \Pi^1_1, 1\langle\rangle^1\rangle^1$.

**Example 5.19** A more interesting example is the problem of constructing polytypic parser, which takes a string and produces an element of type $F(\overline{X})$. This parser is used in the polytypic data compression algorithm of [JJ97, Jan97]. More precisely, we would like a combinator for generating a parser for the type $F(\overline{X})$ given parsers for the $X_i$

$$parse \quad : \quad \forall m : N.\forall F : m.\forall X_{i:m} : \mathrm{T}.(P_S X_i)_{i:m} \to P_S(F(\overline{X}))$$

where $P_S$ is the parsing monad $P_S X = S \to (X \times S) + 1$ defined in Section 5.3.2.1, and usually the type parameter $S$ consists of either lists of characters or lists of tokens. [Bjö97] define a more complex $parse$ combinator

```
parse :: (d a -> Struct a) -> Parser String a -> Parser String (d a)
```

parameterized over a regular datatype $d$ rather than a functor.

It is natural to decompose $parse$ in two parts: $parseshape$ for parsing the shape information, and $parsedata$ for filling the shape with data.

$$\begin{aligned} parseshape &: \forall m : N.\forall F : m.P_S(F(\overline{1})) \\ parsedata &: \forall m : N.\forall F : m.\forall X_{i:m} : \mathrm{T}.(P_S X_i)_{i:m} \to F(\overline{1}) \to P_S(F(\overline{X})) \end{aligned}$$

$parsedata$ is unproblematic and could be defined as an instance of $\mathrm{ltraverse}_m$. Defining $parseshape$ requires either explicit knowledge of the functor $F$ or a single type to describe all possible shapes.

## 5.6 Comparison of FML with PolyP

In this section we make a comparison between **PolyP** (as described in [JJ97]) and FML. At present FML is only a formal calculus, where choices and issues are spelled out and addressed precisely, but the language has not been implemented. On the other hand, **PolyP** is being implemented as an extension of *Haskell*, but some issues are resolved in a crude way in order to get an implementation up and running, and less attention has been give to a rigorous documentation.

**Functors and datatype constructors.** FML starts from the observation that the categorical concept of a functor is an abstraction with which it is both useful and feasible to write programs. Indeed several researchers have advocated *categorical programming* and stressed the importance of functors (e.g. [Hag87a, Hag87b, MFP91, CF92, CS95]).

In FML, functors are considered as fundamental as types (and type schema); there is no attempt to explain them in terms of type constructors with additional structure and properties. Its functor application is the canonical way of getting a type constructor from a functor, but there is no way to go in the opposite direction.

On the contrary, in **PolyP** functors (of arity 2) seem auxiliary to *regular* datatypes (of arity 1). The **PolyP** operations

$$\begin{aligned} FunctorOf &: Regular \to Bifunctor \\ Mu &: (Bifunctor, Regular) \to Regular \end{aligned}$$

do not have a clear category-theoretic reading (unlike the FML functor constructors). They appear to be motivated mainly by the desire to keep information about names of datatype constructors in the qualified kind *Regular*, but ignore it in *Bifunctor*. In particular, *FunctorOf* is defined by induction on the syntax of datatype definitions. For example, given the datatypes

$$\begin{aligned} Empty_1\ X &= \text{empty of } Empty_1\ X \\ Empty_2\ X &= \text{empty of } Empty_1\ X \end{aligned}$$

$FunctorOf(Empty_1) = Rec$ and $FunctorOf(Empty_2) = Empty_1@Par$ are different, although the types $Empty_1\ X$ and $Empty_2\ X$ are isomorphic to the empty type. *FunctorOf* does not make much sense semantically because it recovers a functor from its initial algebra.

**Functor arities.** At present **PolyP** handles only bifunctors, i.e. functors of arity 2. Moreover, the syntactic category of bifunctors $F$ makes reference to two other syntactic categories: datatypes $D$ and types $\tau$. These are not given explicitly, thus it is unclear whether they correspond to the FML arities 1 and 0.

Fixing an upper bound to functor arities is a simple way to avoid both the infinitary rules (that we adopted in the informal account of $\text{FML}_{poly}$) and the dependent types (that are needed in a formal account). Such upper bounds are inherently ad hoc, but once the upper bound is chosen an implementation of $\text{FML}_{poly}$ becomes feasible.

**Separation between functors and types.** FML makes a clear decision: the language for functors comes before and is independent from the language for types. This is very important for having a considerable degree of freedom in interpreting functors (see Section 5.1.5). On the other hand, **PolyP** has a construct $Con : * \to Bifunctor$, which converts a type into a constant bifunctor. Because of this the bifunctors of **PolyP** may fail to preserve equality types. Moreover, it is unclear how the $Con\ \tau$ case of a polytypic definition should be handled, i.e. whether it may treat different $\tau$s differently or it must treat them uniformly.

**Type inference.** Type inference for FML is a fairly straightforward extension of type inference for ML. In **PolyP** the situation is more delicate. In the definition of a polytypic function the type schema should be given explicitly. However, Section 2.2 of [JJ97] claims that one can infer the functor argument of a polytypic function. This seems doubtful without restrictions on the type schema allowed in polytypic definitions, like those introduced in the typing rule (poly) of $\mathrm{FML}_{poly}$. For instance, we expect type inference to be problematic, when one allows polytypic functions like *parse* (see Example 5.19).

**Inductive functors or beyond.** The polytypic construct of **PolyP** is very natural and convenient for dealing with inductive functors. Indeed in $\mathrm{FML}_{poly}$ one may easily define polytypic combinators which have to be taken as primitive in $\mathrm{FML}_{basic}$ and $\mathrm{FML}_{shape}$. On the other hand, $\mathrm{FML}_{basic}$ and $\mathrm{FML}_{shape}$ take an open-ended view of functors, which is incompatible with mechanisms allowing the programmer to define their own polytypic operations by induction on the structure of functors. In these calculi the only way to build polytypic functions is by combining existing ones according to fixed patterns. Nevertheless the combinators of $\mathrm{FML}_{shape}$ are powerful enough to express algorithms regarded as motivating examples for the polytypic construct. The intended semantic of $\mathrm{FML}_{shape}$ tell us that its polytypic combinators are applicable to a larger class of functors than the inductive ones. So there is not a clear best choice between $\mathrm{FML}_{shape}$ and $\mathrm{FML}_{poly}$.

## 5.7 Conclusions

FML is an extension of the Hindley-Milner type system that supports parametric functorial polymorphism. That is, one can write algorithms (e.g. mapping and folding) which work uniformly for a large class of functorial type constructors, unlike previous, ad hoc algorithms. The Hindley-Milner type inference algorithm extends smoothly to FML and reduction on well-formed terms is confluent and strongly normalizing.

The functor syntax admits functors of many variables, and functor composition. Canonical isomorphisms are used to distinguish different orders of composition, which allow terms to express the shape-data, or functor-argument decomposition necessary to locate their data. This feature is essential for parametric algorithms.

Much remains to be done. We expect that the usual denotational models of system F can be extended to handle explicit functors. Also, the exact relationship between FML and $F\omega$ is not yet clear. Many of the subscripts on the combinators seem to be redundant. By introducing *form variables* [Jay95a] to represent sequences of types we may be able to infer many of them, just as we infer types. Another, basic shape polymorphic operation is that of extracting the data from the shape. This is fundamental to search operations, pattern-matching etc. and should be comfortably supported within the current system, as a new combinator.

FML should be considered as an intermediate language. Indeed, the examples show that FML is rather awkward in comparison with ML. FML provides a fine analysis of access to data via the canonical isomorphisms, and should be compared with other intermediate languages, such as those proposed in [PJ91, Ler92] to distinguish between boxed and unboxed values and providing explicit coercions between them. One can envisage an intensional semantics where $\mathrm{intro}^{\Pi}_{m,i}(t) \in \Pi^m_i(\overline{X})$ is like a boxed value, since $t$ is wrapped with additional information about $m$ and $i$, while $\mathrm{intro}^{\circ}_{m,n}$ acts like data redistribution. Here, distinguishing between types and functors is crucial.

Another possibility is to add additional base functors that are not inductive, e.g. arrays. This would allow for types such as trees whose nodes support arrays, as appeared in [WFK97], and connect to research on shape analysis [JS97, JCSS97, JS98].

Finally, it remains to implement FML as an extension of an existing programming language, so that its merits can be tested by the community of programmers.

# Part III

# Conclusion

# Chapter 6

# Future work

This thesis has suggested and provided a semantic framework for developing efficient implementations of programming languages, possibly on parallel machines. The development of linguistic constructs and corresponding analysis and compilation techniques for improving software productivity have not been closely investigated in this thesis, but we believe that a *clear understanding of the underlying abstractions and their properties are critical*, as Harper and Mitchell has stressed in [HM96].

In the first part we have considered toy languages that take into account aspects related to shapes. Now we have to integrate these aspects with features of more realistic languages, such as non-termination. Also polymorphism, and in particular shape polymorphism, should be integrated, so as to join the aspects we have discussed in the first and in the second part of the dissertation, respectively. Other run-time features, such as exceptions and side-effects, can be incorporated in the computational monad used to give the interpretation of the toy languages.

Shape analysis supports program specialization. Metalanguages are right for semantics, but simpler languages, for example languages that support arrays of fixed length, are more appropriate for studying analysis techniques. Adapting traditional techniques to deal with information related to shape, we can develop forms of partial evaluation for such languages. In these cases partial evaluation should consist in: detection of shape errors, shape constants propagation, specialization of expressions.

Regarding metalanguages, **S2** should be provided with an operational semantics. Probably this can be done by borrowing ideas from [Gog94], where a typed operational semantics for type theory is introduced, and from [Dav96], where a logical construction of a binding-time type system based on temporal logic is given. It is then important to extend **S2** to take into account concurrency and data distribution as well as complexity aspects. Preliminary results have been already achieved ([Bel98]) by building on the notion of action model and its relationship to linear logic. Functional programming has made extensive use of intuitionistic logic for motivating language designs and analyzing programs and specifications. Formulas of linear logic allow for more expressive types. In fact from a programming point of view, we can say that linear logic is about process states, events, or resources, which must be carefully accounted for. Action calculi, introduced by Milner [Mil96] is a framework for investigating models of concurrent and interactive behaviour. A connection between action calculi and linear logic is given in [BGHP98]. They give a translation from a type-theoretic account of action calculi in intuitionistic linear logic, corresponding to the fact that Benton's models of linear logic [Ben95] yield models of action calculi.

A refined semantics, which may take into account complexity as well as data distribution and concurrent aspects, allows one to single out optimizing transformations. More simply, we can use these refined models to validate transformations appearing in the literature. For example, program transformations in [To95, Ski94] could be a good starting point. Semantic models then, can suggest which, among the many operations and constructs for parallel programming proposed in literature, have to be considered basic. Of course, given the variety of parallel programming models, it is necessary to restrict to particular paradigms. For example we could take inspiration from [SHM97, DGTY95, BDO+95].

We have already outlined desirable evolvements about FML in Section 5.7. Here we would like to stress two tasks, which we can identify as *efficiency* task and *aesthetic* task. The aim of the first one is to make FML more efficient by a translation of FML into ML. ML does not support the class of functors. Instead of implementing this class, we propose to specialize the combinators (which have polymorphic types with respect to functors). So the translation occurs in two steps: the first one corresponds to the inference of information needed for the second step, which is the proper translation. FML is rather awkward and requires to specify the arities of the functors involved in the combinators, and to explicitly write the canonical isomorphisms. Therefore, the second task concerns the study of the possibility of inferring both arities and isomorphisms. Moreover we could always identify particular useful constructors and add constants for them.

# Appendix A

# Glossary: mathematical preliminaries

In this chapter we recall some categorical notions which appear in the literature. The chapter is organized by themes.

## A.1 Category theory

### A.1.1 Strength, monads and Kleisli categories

**Definition A.1 (Strength)** *A* strength *for a functor $F : \mathbf{C} \to \mathbf{C}$ over a category $\mathbf{C}$ with (explicitly given) finite products, is a natural transformation $t_{A,B}$ from $A \times FB$ to $F(A \times B)$, such as*

$$\begin{tikzcd}
FA & \\
1 \times FA \arrow[u, "r_{FA}"] \arrow[r, "t_{1,A}"'] & F(1 \times A) \arrow[ul, "Fr_A"']
\end{tikzcd}$$

$$\begin{tikzcd}
(A \times B) \times FC \arrow[r, "t_{A\times B,C}"] \arrow[d, "\alpha_{A,B,FC}"'] & F((A \times B) \times C) \arrow[dr, "F\alpha_{A,B,C}"] & \\
A \times (B \times FC) \arrow[r, "\mathrm{id}_A \times t_{B,C}"] & A \times F(B \times C) \arrow[r, "t_{A,B\times C}"] & F(A \times (B \times C))
\end{tikzcd}$$

*where $r$ and $\alpha$ are the natural isomorphisms*

$r_A : (1 \times A) \to A$, $\alpha_{A,B,C} : (A \times B) \times C \to A \times (B \times C)$

**Definition A.2 (Kleisli triple)** *A* Kleisli triple *over a category* $\mathbf{C}$ *is a triple* $(T, \eta, -^*)$, *where* $T : Obj(\mathbf{C}) \to Obj(\mathbf{C})$, $\eta_A : A \to TA$ *for* $A \in Obj(\mathbf{C})$, $f^* : TA \to TB$ *for* $f : A \to TB$ *and the following equations hold:*

- $\eta_A^* = id_{TA}$
- $\eta_A; f^* = f$ *for* $f : A \to TB$
- $f^*; g^* = (f; g^*)^*$ *for* $f : A \to TB$ *and* $g : B \to TC$

Kleisli triples are just an alternative description for monads.

**Definition A.3 (Monad)** *A* monad *over a category* $\mathbf{C}$ *is a triple* $(T, \eta, \mu)$, *where* $T : \mathbf{C} \to \mathbf{C}$ *is a functor,* $\eta : Id_{\mathbf{C}} \to T$ *and* $\mu : T^2 \to T$ *are natural transformations and the following diagrams commute:*

$$\begin{tikzcd}
T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T_{\mu_A}"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[dr, "id_{TA}"'] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "id_{TA}"] \\
& TA &
\end{tikzcd}$$

The following proposition which relates Kliesli triples and monads is due to Manes

**Proposition A.4** *There is a one-one correspondence between Kleisli triples and monads.*

**Definition A.5 (Strong Monad)** *A* strong monad $(T, t)$ *over a category* $\mathbf{C}$ *with (explicitly given) finite products is a monad* $(T, \eta, \mu)$ *together with a strenght* $t$ *s.t.*

$$
\begin{tikzcd}
A \times B \arrow[d, "id_{A \times \eta_B}"'] \arrow[dr, "\eta_{A \times B}"] & & \\
A \times TB \arrow[r, "t_{A,B}"] & T(A \times B) & \\
A \times T^2 B \arrow[u, "id_{A \times \mu_B}"] \arrow[r, "t_{A,TB}"] & T(A \times TB) \arrow[r, "Tt_{A,B}"] & T^2(A \times B)) \arrow[ul, "\mu_{A \times B}"']
\end{tikzcd}
$$

It is often convenient to deal with a metalanguage for a category rather then directly with the category itself. In this view (see [Mog91b]), monads are unary type-constructors $T$, with two term-constructors

$$
\begin{array}{rcl}
val_T & : & \forall X : \mathrm{T}.X \to TX \\
let_T & : & \forall X, Y : \mathrm{T}.(X \to TY), TX \to TY
\end{array}
$$

We adopt the syntax of [Mog91b], namely $[x] \triangleq val_T\ x$ and let $x \Leftarrow e_1$ in $e_2 \triangleq let_T\ (\lambda x.e_2)\ e_1$.

**Definition A.6 (Kleisli category of a monad)** *Given a monad $(T, \eta, \mu)$ over $\mathbf{C}$, the* Kleisli category $\mathbf{C}_T$ *is defined as follows:*

- *the objects of $\mathbf{C}_T$ are those of $\mathbf{C}$*
- *the set $\mathbf{C}_T(A, B)$ of morphisms from $A$ to $B$ in $\mathbf{C}_T$ is $\mathbf{C}(A, TB)$*
- *the identity on $A$ in $\mathbf{C}_T$ is $\eta_A : A \to TA$*
- *$f \in \mathbf{C}_T(A, B)$ followed by $g \in \mathbf{C}_T(B, C)$ in $\mathbf{C}_T$ is $\mu_B \circ Tg \circ f$.*

### A.1.2 List object and natural number object

**Definition A.7 (List object - strong)** *Let $\mathbf{C}$ be a category with finite products. A* list object *$LA$ for an object $A$ in $\mathbf{C}$ is given by a pair of morphisms*

$$
\begin{tikzcd}
1 \arrow[r, "nil"] & LA & A \times LA \arrow[l, "cons"']
\end{tikzcd}
$$

*with the following property. For any choice of objects $B$ and $C$, and morphisms $x : B \to C$ and $h : A \times C \to C$ there is a unique morphism $foldr(x, h) : LA \times B \to C$ (called* foldright *of $x$ and $h$) that makes the following diagram commute:*

$$\begin{tikzcd}
B \arrow[r, "\langle nil, id_B \rangle"] \arrow[dr, "x"'] & LA \times B \arrow[d, dashed, "foldr(x,h)"] & A \times LA \times B \arrow[l, "\langle cons, id_B \rangle"'] \arrow[d, dashed, "\langle id_A, foldr(x,h) \rangle"] \\
& C & A \times C \arrow[l, "h"']
\end{tikzcd}$$

The list object $L$ extends to a functor. If $f : A \to B$ is a morphism then $Lf = foldr(nil, cons \circ (f \times id_{LB}))$. $nil$ and $cons$ are natural transformations, $nil : Id \to L$ and $cons : Id \times L \to L$.

Also $foldr(x, h)$ extends to a natural transformation, in this way. If $\alpha : F \to G$ and $\beta : Id \times G \to G$ are natural transformation then $\gamma_A = foldr(\alpha_A, \beta_A)$ is a natural transformation $L \times F \to G$.

Further, $(L, \eta, \mu, \mathrm{t})$ is a strong monad, where

$$\begin{aligned}
\eta_A &= cons \circ \langle id_A, nil \rangle \\
\mu_A &= foldr(nil, foldr(id_{LA}, cons)) \\
\mathrm{t}_{A,B} &= L(c_{B,A}) \circ (\pi_1 \circ foldr(\langle id_A, nil_{B \times A} \rangle, \langle \pi_1, cons_{B \times A} \rangle)) \circ c_{A,LB}
\end{aligned}$$

and $\eta$ and $\mu$ are cartesian natural transformations.

Moreover we can define the natural transformation $@_A : LA \times LA \to LA$, namely

$$@_A = foldr(id_{LA}, cons)$$

.

**Definition A.8 (Natural Number Object - strong)** *Let $\mathbf{C}$ be a category with finite products. A* natural number object *$\mathbf{N}$ is given by a pair of morphisms*

$$1 \xrightarrow{zero} \mathbf{N} \xleftarrow{succ} N$$

*with the following property. For any choice of objects $B$ and $C$ and morphisms $x : B \to C$ and $h : C \to C$ there is a unique morphism $It(x, h) : N \times B \to C$ (called the* iterator *of $x$ and $h$) that makes the following diagram commute:*

$$\begin{tikzcd}
B \arrow[r, "\langle zero, id_B \rangle"] \arrow[dr, "x"'] & \mathbf{N} \times B \arrow[d, dashed, "It(x,h)"] & N \times B \arrow[l, "\langle succ, id_B \rangle"'] \arrow[d, dashed, "It(x,h)"] \\
& C & C \arrow[l, "h"']
\end{tikzcd}$$

We denote by $eq : \mathbf{N} \times \mathbf{N} \to 1 + 1$ the equality for the natural number object.

A list object for 1 is a natural number object $\mathbf{N}$.

### A.1.3 Extensivity

For an introduction to extensive categories we refer the reader to [CLW93].

**Definition A.9 (Extensive category)** *A category* $\mathbf{A}$ *with finite sums is called* extensive *if for each pair* $X_0, X_1$ *of objects in* $\mathbf{A}$*, the canonical functor*

$$+ : \mathbf{A}/X_0 \times \mathbf{A}/X_1 \to \mathbf{A}/(X_0 + X_1)$$

*is an equivalence.*

Being extensive is equivalent to requiring that all coproduct diagrams have disjoint (monomorphic) inclusions, and are stable under pulling back.

Using a slogan in [CLW93]: an extensive category is one in which sums exist and are well-behaved.

**Definition A.10 (Lextensive category)** *An extensive category with all finite limits is called* lextensive.

**Definition A.11 (Locos)** *A* locos *is a lextensive category which has all list objects.*

### A.1.4 Some categorical constructions

**Definition A.12 (Slice category)** *Let* $\mathbf{C}$ *be an arbitrary category and let* $X \in \mathbf{C}$*. Then the* slice category $\mathbf{C}/X$ *has as objects the morphisms with codomain* $X$ *and as morphisms from* $f_1 : Y_1 \to X$ *to* $f_2 : Y_2 \to X$ *the* $g : Y_1 \to Y_2$ *s.t.*

$$\begin{tikzcd} Y_1 \arrow[r, "g"] \arrow[dr, "f_1"'] & Y_2 \arrow[d, "f_2"] \\ & X \end{tikzcd}$$

**Definition A.13 (Arrow category)** *Let* $\mathbf{C}$ *be an arbitrary category. The category* $\mathbf{C}^{\to}$ *has objects given by arrows* $\alpha : A \to I$ *in* $\mathbf{C}$ *and morphisms from* $\alpha$ *to* $\beta : B \to J$ *are given by pairs or arrows* $f : A \to B$ *and* $u : I \to J$ *in* $\mathbf{C}$*, such the the following square*

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "\alpha"'] & B \arrow[d, "\beta"] \\ I \arrow[r, "u"] & J \end{tikzcd}$$

*commutes.*

**Definition A.14 (Set-indexed families)** *Let* $\mathbf{C}$ *be an arbitrary category. The category* $\mathbf{Fam}(\mathbf{C})$ *of set-indexed families of objects and morphisms of* $\mathbf{C}$ *is defined as follows:*

**objects** *families* $(X_i)_{i \in I}$ *where for each element* $i$ *of the index set* $I$*,* $X_i$ *is an object of* $\mathbf{C}$*; these may be described by a pair* $(I, X)$ *with* $X : I \to obj(\mathbf{C})$*.*

**morphisms** $f : (X_i)_{i \in I} \to (Y_j)_{j \in J}$ *is given by a function* $u : I \to J$ *such that for every* $i \in I$ *one has* $f_i : X_i \to Y_{u(i)}$ *in* $\mathbf{C}$*.*

$\mathbf{Fam}(\mathbf{C})$ is the free category with sums on the category $\mathbf{C}$.

**Notation A.15**

1. A family $(A_k)_{k \in I+J}$ can be denoted as $[(A_{\langle 0,i \rangle})_{i \in I}, (A_{\langle 1,j \rangle})_{j \in J}]$ or equivalently $[(A_{\langle 0,i \rangle})_{i \in I} | (A_{\langle 1,j \rangle})_{j \in J}]$ .

2. A morphism $\langle u, (f_k)_{k\in I+J}\rangle : (A_k)_{k\in I+J} \to (B_l)_{l\in L}$ can be denoted as $\langle u, [(f_{\langle 0,i\rangle})_{i\in I}|(f_{\langle 1,j\rangle})_{j\in J}]\rangle$.

3. Families $(A)_{*\in 1}$ indexed over the singleton set can be denoted simply as $A$.

The following proposition, taken from [CLW93, Prop. 2.4], say that the set-indexed families are extensive.

**Proposition A.16** *The free category with sums on a category* $\mathbf{C}$ *is extensive.*

The following proposition shows the categorical structure in **Fam**.

**Proposition A.17** *Let* $\mathbf{B}$ *be a cartesian closed category with small products. Then the category* $\mathbf{Fam}(\mathbf{B})$ *is cartesian closed with small products and small sums, extensive and with a natural number object.*

**Proof.** The relevant categorical structure in $\mathbf{Fam}(\mathbf{B})$ is defined as follows: $1 = (1)_{*\in 1}$, $(A_i)_{i\in I} \times (B_j)_{j\in J} = (C\langle i,j\rangle)_{\langle i,j\rangle\in I\times J}$ with $C_{i,j} = A_i \times B_j$, $(B_j)_{j\in J}^{(A_i)_{i\in I}} = (C_f)_{f\in J^I}$ with $C_f = \prod_{i\in I} B_{fi}^{A_i}$, $\coprod_{i\in I}(B_{x_i})_{x_i\in J_i} = \langle \coprod_{i\in I} J_i, C\rangle$ with $C_{i,x_i} = B_{x_i}$.

Moreover set-indexed families are extensive, see Prop. A.16. The natural number object in $\mathbf{Fam}(\mathbf{B})$ is defined as the family $(1_n)_{n\in N}$.

∎

**Definition A.18** ($\wedge$) *The category* $\wedge$ *is given by the following graph*

$$\begin{tikzcd} & \bullet \arrow[dl] \arrow[dr] & \\ \bullet & & \bullet \end{tikzcd}$$

### A.1.5 Miscellany

**Definition A.19 (section)** *For a morphism* $f : A \to B$*, a* section *is a morphism* $g : B \to A$ *with* $f \circ g = id_B$*. A section (which is usually not unique) is also called a* right inverse *of* $f$*.*

## A.2 Fibred category theory

The definitions in this section have been taken from [Jac91].

**Definition A.20** *Let $p : \mathbf{E} \to \mathbf{B}$ be a functor. For an object $I \in \mathbf{B}$, the* fibre $\mathbf{E}_I$ *over $I$ is the category with*

**objects** *$X \in \mathbf{E}$ with $pX = I$*

**morphisms** *$X \to Y$ in $\mathbf{E}_I$ are morphisms $f : X \to Y$ in $\mathbf{E}$ for which $pf$ is the identity on $I$ in $\mathbf{B}$.*

*An object $X \in \mathbf{E}$ such that $pX = I$ is said to be* above *$I$; similarly a morphism $f$ in $\mathbf{E}$ with $pf = u$ is said to be* above *$u$. A morphism in $\mathbf{E}$ will be called* vertical *if it is above some identity morphism in $\mathbf{B}$, that is, when it is in a fibre category. We call $\mathbf{B}$ the base category and $\mathbf{E}$ the total category.*

**Definition A.21 (Fibration)** *Let $p : \mathbf{E} \to \mathbf{B}$ be a functor.*

1. *A morphism $f : X \to Y$ in $\mathbf{E}$ is* cartesian over *$u : I \to J$ in $\mathbf{B}$ if $pf = u$ and every $g : Z \to Y$ in $\mathbf{E}$ for which one has $pg = u \circ w$ for some $w : pZ \to I$, uniquely determines an $h : Z \to X$ above $w$ with $f \circ h = g$. We call $f : X \to Y$ in the total category $\mathbf{E}$ cartesian if it is cartesian over its underlying map $pf$ in $\mathbf{B}$.*

2. *The functor $p : \mathbf{E} \to \mathbf{B}$ is a* fibration *if for every $Y \in \mathbf{E}$ and $u : I \to pY$ in $\mathbf{B}$, there is a cartesian morphism $f : X \to Y$ in $\mathbf{E}$ above $u$. Sometimes a fibration will be called a* fibred category *or a* category (fibred) over *$\mathbf{B}$.*

**Reindexing.** If $p : \mathbf{E} \to \mathbf{B}$ is a fibration and $f : X \to Y$ and $f' : X' \to Y'$ are both cartesian morphisms over $u$ then $f \cong f'$ in $\mathbf{E}_Y$ by a vertical isomorphism. Hence given $u : I \to J$ in $\mathbf{B}$ and $Y$ above $J$, it makes sense to choose a cartesian lifting of $u$ with codomain $Y$; we often write $\bar{u}(Y) : u^*(Y) \to Y$ for such a choice. A collection of choices — for every appropriate $u$ and $Y$ — is called a *cleavage*. It induces for every $u : I \to J$ a functor $u^* : \mathbf{E}_J \to \mathbf{E}_I$, called *inverse image, reindexing, relabelling* or *substitution* functor. Different cleavages give rise to different, but naturally isomorphic, reindexing functors. In general, one obtains vertical natural isomorphisms $(u \circ v)^* \cong v^* \circ u^*$ and $id^* \cong Id$. If one happens to have identities here (for a certain cleavage), one says that the fibration can be *split*.

**Definition A.22 (1-cell)** *A morphism* $(K, H)$ *between fibrations* $p : \mathbf{E} \to \mathbf{B}$ *and* $q : \mathbf{D} \to \mathbf{A}$ *is given by a commuting square as below, in which the functor* $H$ *preserves cartesian morphisms.*

$$
\begin{tikzcd}
\mathbf{E} \arrow[r, "H"] \arrow[d, "p"'] & \mathbf{D} \arrow[d, "q"] \\
\mathbf{B} \arrow[r, "K"] & \mathbf{A}
\end{tikzcd}
$$

**Definition A.23** *Assume* $(K, H)$ *and* $(L, G)$ *are morphisms between fibrations* $p : \mathbf{E} \to \mathbf{B}$ *and* $q : \mathbf{D} \to \mathbf{A}$ *(i.e. 1-cells). A* 2-cell $(K, H) \stackrel{\cdot}{\to} (L, G)$ *in* **Fib** *consists of a pair of natural transformations* $\sigma : K \stackrel{\cdot}{\to} L$ *and* $\tau : H \stackrel{\cdot}{\to} G$ *in*

$$
\begin{tikzcd}
\mathbf{E} \arrow[r, shift left=2, "H"] \arrow[r, shift right=2, "G"'] \arrow[r, phantom, "\Downarrow\tau"] \arrow[d, "p"'] & \mathbf{D} \arrow[d, "q"] \\
\mathbf{B} \arrow[r, shift left=2, "K"] \arrow[r, shift right=2, "L"'] \arrow[r, phantom, "\Downarrow\sigma"] & \mathbf{A}
\end{tikzcd}
$$

*such that* $\tau$ *is above* $\sigma$*; that is, for* $X \in E$*, the component* $\tau_X$ *is above the component* $\sigma_{pX}$*.*

Fibrations, 1-cells and 2-cells yield a 2-category (see e.g. [Mac71]), called **Fib**.

**Definition A.24 (Fibred notions)** *Let* $\diamond \in$*{terminal (initial) object, binary (co-) product, (co-) equalizer, exponent }. We say that a fibration* $p$ *has* fibred $\diamond$*'s if every fibre category has* $\diamond$*'s and all reindexing functors preserve the* $\diamond$*'s.*

**Definition A.25 (Products)** *Let* $\mathbf{B}$ *be a category with pullbacks and* $p : \mathbf{E} \to \mathbf{B}$ *be a fibration on* $\mathbf{B}$*. One says that* $p$ *has* products *(resp.* coproducts*) if both*

- *for every morphism* $u : I \to J$ *in* $\mathbf{B}$*, every substitution functor* $u^* : \mathbf{E}_J \to \mathbf{E}_I$ *has a right adjoint* $\prod_u$ *(resp. a left adjoint* $\coprod_u$*);*

- *the Beck-Chevalley condition holds: for every pullback in* $\mathbf{B}$ *of the form*

$$\begin{tikzcd} K \arrow[r, "v"] \arrow[d, "r"'] \arrow[dr, phantom, very near start, "\lrcorner"] & L \arrow[d, "s"] \\ I \arrow[r, "u"] & J \end{tikzcd}$$

*one has that the canonical natural transformation* $s^* \prod_u \to \prod_v r^*$ *(resp.* $\coprod_v r^* \to s^* \coprod_u$*) is an isomorphism.*

**Definition A.26 (Indexed category)** *An* indexed category *is a functor of the form* $\mathbf{D} : \mathbf{B}^{op} \to \mathbf{Cat}$*.*

**Remark A.27** We understand an indexed category as a *functor* and not as a *pseudo-functor*. The latter would mean that one allows isomorphisms $\mathbf{D}(id) \cong id$ and $\mathbf{D}(u \circ v) \cong \mathbf{D}(v) \circ \mathbf{D}(u)$, in a coherent way.

**Definition A.28 (Grothendieck construction)** *Let* $\mathbf{D} : \mathbf{C}^{op} \to \mathbf{Cat}$ *be an indexed category. The* Grothendieck construction $\mathcal{G}\mathbf{D}$ *of* $\mathbf{D}$ *is the category with*

**objects** $\langle I, X \rangle$ *where* $I \in \mathbf{C}$ *and* $X \in \mathbf{D}[I]$

**morphisms** $\langle I, X \rangle \to \langle J, Y \rangle$ *are pairs* $\langle u, f \rangle$ *with* $u : I \to J$ *in* $\mathbf{C}$ *and* $f : X \to u^*(Y) = \mathbf{D}[u](Y)$ *in* $\mathbf{D}[I]$.

# Appendix B

# Strong normalization for FML

We prove SN for *functorial* $\mathrm{F}_{shape}$ (briefly $\mathrm{FF}_{shape}$), a system more powerful than $\mathrm{FML}_{shape}$ which can type every term typable in $\mathrm{FML}_{shape}$ (see Section 5.3). Therefore SN for $\mathrm{FF}_{shape}$ trivially implies SN for $\mathrm{FML}_{basic/shape}$. The proof follows [Men91], which proves SN for system F extended with inductive and coinductive types using Girard's *reducibility candidates* method.

First, the system $\mathrm{FF}_{shape}$ is defined. The definition of system $\mathrm{FF}_{shape}$ involves, as in the case of $\mathrm{FML}_{shape}$, a family of parsing monads $P_S$ (see section 5.3.2.1). The rest of the section consists in the strong normalization proof. The proof is divided into two parts. The first part is given independently from the particular choice of the family of monads. It requires only: a family of monads $P_S$ with terms *val*, *let* and *fail*; a family of monotonic functions $M_W$ on Sat, which satisfies some properties connected to the terms *val*, *let* and *fail*. We now outline this first part.

1. We fix a generic collection of monad $P_S$ and its associated terms *val*, *let* and *fail* (Axiom B.3).
2. Let SN be the set of strongly normalizing terms (it involves the terms *val*, *let* and *fail*). Sat is a complete lattice of subsets of SN (Definition B.4). Types will be modeled by elements in Sat.
3. We fix a family $\{M_W\}_{W \in \mathrm{Sat}}$ of monotonic functions on Sat, such as it satisfies some properties connected to the terms *val*, *let* and *fail* (Axiom B.8).
4. We define what is a shape-based functorial operator (this definition involves the family $\{M_W\}_{W \in \mathrm{Sat}}$). $\mathrm{Sat}_m$ will be the set of shape-based functorial operators (Definition B.9). Functors will be modeled by elements in $\mathrm{Sat}_m$.
5. We give an interpretation $[\![-]\!]$ of functors and types.

6. Proposition B.13 relates typable terms in $\mathrm{FF}_{shape}$ with the interpretations of their types under hypothesis which connect the family of monads (point 1.) to the family of monotonic functions (point 3.). In particular it is required that the interpretation of types $P_S(\tau)$ corresponds to the set $M_{[\![S]\!]}([\![\tau]\!])$.

The second and last part of the proof consists on the Theorem B.14. Only now do we need to provide the family of monads, the family of parsing monads in this case, and a suitable family of monotonic functions on Sat. The monads and the functions are required to satisfy the hypothesis of Proposition B.13. Thus the strong normalization result for $\mathrm{FF}_{shape}$ is a consequence of Proposition B.13.

### B.0.1 System $\mathrm{FF}_{shape}$

System FF is obtained from FML by identifying type and type schemas, and it extends system F in the same way as FML extends ML. More precisely:

- functors are as in FML, i.e.

$$F, G ::= \; X \mid C \mid \Pi_i^m \mid F\langle \overline{G} \rangle^n \mid \mu F$$

- types and type schemas are identified (and called types), i.e.

$$\tau, \sigma ::= \; X \mid F(\overline{\tau}) \mid \tau \to \sigma \mid \forall X : \mathrm{T}.\tau \mid \forall X : m.\tau$$

- formation rules for type contexts $\Delta \vdash$ and functors $\Delta \vdash F : n$ are as in FML, while formation rules for types $\Delta \vdash \tau$ and contexts $\Delta; \Gamma \vdash$ are as in FML modulo the identification of types and type schemas.

- raw terms are as in FML but the term let $x = t_1$ in $t_2$ will be identified with the term $(\lambda x.t_2)t_1$ (this cannot be done in FML),i.e.

$$t ::= \; x \mid c \mid \lambda x.t \mid t_1 \; t_2$$

- the type assignment rules for deriving $\Delta; \Gamma \vdash t : \tau$ in FF are

$$x \; \frac{\Delta; \Gamma \vdash}{\Delta; \Gamma \vdash x : \sigma} \; \sigma = \Gamma(x) \qquad c \; \frac{\Delta; \Gamma \vdash}{\Delta; \Gamma \vdash c : \sigma_c}$$

$$\mathrm{app} \; \frac{\Delta; \Gamma \vdash t : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash t_1 : \tau_1}{\Delta; \Gamma \vdash (t \; t_1) : \tau_2}$$

$$\lambda \; \frac{\Delta;\Gamma, y:\tau_1 \vdash [y/x]t:\tau_2}{\Delta;\Gamma \vdash (\lambda x.t):\tau_1 \to \tau_2} \; y \notin \mathrm{DV}(\Gamma)$$

$$\mathrm{App} \; \frac{\Delta;\Gamma \vdash t : \forall X:\mathrm{T}.\sigma \quad \Gamma \vdash \tau}{\Delta;\Gamma \vdash t : [\tau/X]\sigma}$$

$$\Lambda \; \frac{\Delta, Y:\mathrm{T};\Gamma \vdash t : [Y/X]\sigma}{\Delta;\Gamma \vdash t : \forall X:\mathrm{T}.\sigma} \; Y \notin \mathrm{DV}(\Delta)$$

$$\mathrm{App}_n \; \frac{\Delta;\Gamma \vdash t : \forall X:n.\sigma \quad \Delta \vdash F : n}{\Delta;\Gamma \vdash t : [F/X]\sigma}$$

$$\Lambda_n \; \frac{\Delta, Y:n;\Gamma \vdash t : [Y/X]\sigma}{\Delta;\Gamma \vdash t : \forall X:n.\sigma} \; Y \notin \mathrm{DV}(\Delta)$$

**Lemma B.1** *If $\Delta;\Gamma \vdash t:\tau$ is derivable in* FML*, then it is derivable in* FF*.*

**Proof.** The type assignment rules of FML not already in FF are $(x)$, $(c)$ and (let). The rule $(x)$ of FML is derivable from $(x)$ of FF by repeatedly applying (App) and ($\mathrm{App}_n$). Similarly for $(c)$. (let) of FML is derivable from $(\lambda)$ and (app) of FF and by repeatedly applying $(\Lambda)$ and $(\Lambda_n)$ to the first premise. ∎

For the sake of simplicity, we consider the following variation on the set of terms: combinators will be required to be *fully applied* (see section 5.5.1, this is not essential for SN but it make the definition of value much shorter). As a consequence, the type assignment rules for combinators need to be changed, e.g. the new rule for $\mathrm{intro}^{\mu}_{m}$ becomes

$$\mathrm{intro}^{\mu}_{m} \; \frac{\begin{array}{c}\Delta \vdash F : m+1 \quad \Delta \vdash \tau_{i\in m} \\ \Delta;\Gamma \vdash t : F(\overline{\tau}, \mu F)\end{array}}{\Delta;\Gamma \vdash \mathrm{intro}^{\mu}_{m}(t) : \mu F(\overline{\tau})}$$

**Terms.** We consider as set of combinators the ones in $\mathrm{FML}_{shape}$. We call the resulting calculus $\mathrm{FF}_{shape}$.

We introduce two syntactic categories: terms $t \in \Lambda$ and values $v \in \mathrm{V}$.

$$\begin{array}{rl} f,t \in \Lambda ::= & x \mid v \mid t_1\ t_2 \mid \mathrm{elim}^1(t_1,t_2) \mid \mathrm{elim}^x(f,t) \mid \mathrm{elim}^+(f_0,f_1,t) \mid \\ & \mathrm{elim}^{\mu}_{m}(f,t) \mid \mathrm{elim}^{\Pi}_{m,i}(f,t) \mid \mathrm{elim}^{\circ}_{m,n}(f,t) \mid \mathrm{map}_m(f_{i\in m},t) \mid \\ & \mathrm{ltraverse}_m(f_{i\in m},t) \mid \mathrm{zipop}_m(f_{i\in m},t_1,t_2) \\ v \in \mathrm{V} ::= & \lambda x.t \mid \mathrm{intro}^1 \mid \mathrm{intro}^x(t_0,t_1) \mid \mathrm{intro}^+_i(t) \mid \mathrm{intro}^{\mu}_{m}(t) \mid \\ & \mathrm{intro}^{\Pi}_{m,i}(t) \mid \mathrm{intro}^{\circ}_{m,n}(t) \end{array}$$

**Lemma B.2 (Closure under substitution)** *If $S$ is a term substitution (i.e. a function from term variables to terms) and $v \in \mathrm{V}$, then $S(v) \in \mathrm{V}$.*

**Reduction** Recall that the definitions of $\mathrm{ltraverse}_m$ and $\mathrm{zipop}_m$ is parameterized with respect to a family of parsing monads $P_S$, and that *val*, *let* and *fail* are the terms on $\Lambda$ associated to it (see section 5.3.2.1).

The notion of reduction $\mathrm{F}\beta$ on $\Lambda$ is given by

- $(\lambda x.t_2)\ t_1 > [t_1/x]t_2$
- $\mathrm{elim}^1\ (t, \mathrm{intro}^1) > t$
- $\mathrm{elim}^{\times}(f, \mathrm{intro}^{\times}(t_0, t_1)) > f\ t_i$
- $\mathrm{elim}^{+}(f_0, f_1, \mathrm{intro}^{+}_i(t)) > f_i\ t$
- $\mathrm{elim}^{\Pi}_{m,i}(f, \mathrm{intro}^{\Pi}_{m,i}(t)) > f\ t$
- $\mathrm{elim}^{\circ}_{m,n}(f, \mathrm{intro}^{\circ}_{m,n}(t)) > f\ t$
- $\mathrm{elim}^{\mu}_m(f, \mathrm{intro}^{\mu}_m(t)) > f\ \mathrm{map}_{m+1}((\lambda x.x)_{i\in m}, \lambda x.\mathrm{elim}^{\mu}_m(f,x), t)$
- $\mathrm{map}_0(\mathrm{intro}^1) > \mathrm{intro}^1$
  $\mathrm{map}_2(f_0, f_1, \mathrm{intro}^{\times}(t_0, t_1)) > \mathrm{intro}^{\times}(f_0\ t_0, f_1\ t_1)$
  $\mathrm{map}_2(f_0, f_1, \mathrm{intro}^{+}_i(t)) > \mathrm{intro}^{+}_i(f_i\ t)$
  $\mathrm{map}_m(f_{j\in m}, \mathrm{intro}^{\Pi}_{m,i}(t)) > \mathrm{intro}^{\Pi}_{m,i}(f_i\ t)$
  $\mathrm{map}_n(f_{j\in n}, \mathrm{intro}^{\circ}_{m,n}(t)) > \mathrm{intro}^{\circ}_{m,n}(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i\in m}, t))$
  $\mathrm{map}_m(f_{i\in m}, \mathrm{intro}^{\mu}_m(t)) > \mathrm{intro}^{\mu}_m(\mathrm{map}_{m+1}(\overline{f}, \lambda x.\mathrm{map}_m(\overline{f}, x), t))$
- $\mathrm{ltraverse}_0\ \mathrm{intro}^1 > [\mathrm{intro}^1]$
  $\mathrm{ltraverse}_2(f_0, f_1, \mathrm{intro}^{\times}(t_0, t_1)) > \mathrm{let}\ y_0 \Leftarrow f_0\ t_0\ \ \mathrm{in}\ \mathrm{let}\ y_1 \Leftarrow f_1\ t_1\ \ \mathrm{in}\ [\mathrm{intro}^{\times}(y_0, y_1)]$
  $\mathrm{ltraverse}_2(f_0, f_1, \mathrm{intro}^{+}_j(t)) > \mathrm{let}\ y \Leftarrow f_j\ t\ \ \mathrm{in}\ [\mathrm{intro}^{+}_j(y)]$
  $\mathrm{ltraverse}_m(f_{i:m}, \mathrm{intro}^{\Pi}_{m,i}(t)) > \mathrm{let}\ y \Leftarrow f_i\ t\ \ \mathrm{in}\ [\mathrm{intro}^{\Pi}_{m,i}(y)]$
  $\mathrm{ltraverse}_n(f_{i:n}, \mathrm{intro}^{\circ}_{m,n}(t))) > \mathrm{let}\ y \Leftarrow \mathrm{ltraverse}_m\ ((\lambda x.\mathrm{ltraverse}_n(\overline{f}, x))_{i:m}, t)\ \mathrm{in}\ [\mathrm{intro}^{\circ}_{m,n}(y)]$
  $\mathrm{ltraverse}_m(f_{i:m}, \mathrm{intro}^{\mu}_m(t)) > \mathrm{let}\ y \Leftarrow \mathrm{ltraverse}_{m+1}(\overline{f}, \lambda x.\mathrm{ltraverse}_m(\overline{f}, x), t)\ \mathrm{in}\ [\mathrm{intro}^{\mu}_m(y)]$
- $\mathrm{zipop}_0\ \mathrm{intro}^1\ \mathrm{intro}^1 > [\mathrm{intro}^1]$
  $\mathrm{zipop}_2(f_0, f_1, \mathrm{intro}^{\times}\ (s_0, s_1), \mathrm{intro}^{\times}\ (t_0, t_1)) >$
  $\mathrm{let}\ z_0 \Leftarrow f_0\ s_0\ t_0\ \ \mathrm{in}\ \mathrm{let}\ z_1 \Leftarrow f_1\ s_1\ t_1\ \ \mathrm{in}\ [\mathrm{intro}^{\times}(z_0, z_1)]$
  $\mathrm{zipop}_2(f_0, f_1, \mathrm{intro}^{+}_j(s), \mathrm{intro}^{+}_j(t)) > \mathrm{let}\ z \Leftarrow f_j\ s\ t\ \mathrm{in}\ [\mathrm{intro}^{+}_j(z)]$
  $\mathrm{zipop}_2(f_0, f_1, \mathrm{intro}^{+}_0(s), \mathrm{intro}^{+}_1\ (t)) > \mathit{fail}$

$$\mathrm{zipop}_2(f_0, f_1, \mathrm{intro}_1^+(s), \mathrm{intro}_0^+\ (t)) > \mathit{fail}$$

$$\mathrm{zipop}_m(f_{i:m}, \mathrm{intro}_{m,i}^\Pi(s), \mathrm{intro}_{m,i}^\Pi(t)) > \mathrm{let}\ z \Leftarrow f_i\ s\ t\ \ \mathrm{in}\ [\mathrm{intro}_{m,i}^\Pi(z)]$$

$$\mathrm{zipop}_n(f_{i:n}, \mathrm{intro}_{m,n}^\circ(s), \mathrm{intro}_{m,n}^\circ(t)) >$$
$$\mathrm{let}\ z \Leftarrow \mathrm{zipop}_m((\lambda x.\lambda y.\mathrm{zipop}_n(\overline{f}, x, y))_{i:m}, s, t)\ \mathrm{in}\ [\mathrm{intro}_{m,n}^\circ(z)]$$

$$\mathrm{zipop}_m(f_{i:m}, \mathrm{intro}_m^\mu(s), \mathrm{intro}_m^\mu(t)) >$$
$$\mathrm{let}\ z \Leftarrow \mathrm{zipop}_{m+1}(\overline{f}, \lambda x.\lambda y.\mathrm{zipop}_m(\overline{f}, x, y), s, t)\ \mathrm{in}\ [\mathrm{intro}_m^\mu(z)]$$

We write $>$ also for the corresponding reduction, i.e. the *compatible*, reflexive and transitive closure of $>$.

## B.0.2 Semantic domains

This section defines the sets Sat and $\mathrm{Sat}_m$: Sat consists of sets for interpreting types, whereas $\mathrm{Sat}_m$ consists of functions for interpreting functors of arity $m$.

**Axiom B.3** *Fix a family of monads* $S, X : \mathrm{T} \vdash P_S(X) : \mathrm{T}$ *and three terms* $S : \mathrm{T} \vdash \mathit{val} : \forall X : \mathrm{T}.X \to P_S(X)$, $S : \mathrm{T} \vdash \mathit{let} : \forall X, Y : \mathrm{T}.(X \to P_S(Y)) \to P_S(X) \to P_S(Y)$ *and* $S : \mathrm{T} \vdash \mathit{fail} : \forall X : \mathrm{T}.P_S(X)$. *We will write* $[x]$ *for* $\mathrm{val}\ x$ *and* $\mathrm{let}\ x \Leftarrow t\ \mathrm{in}\ b$ *for* $\mathit{let}(\lambda x.b)\ t$.

**Definition B.4 (Saturated sets)**

- SN *is the set of* $t \in \Lambda$ *which are strongly normalizable.*
- $\mathrm{V}(t)$ *is the set of values of* $t$, *i.e. those* $v \in \mathrm{V}$ *such that* $t >^* v$.
- Sat *is the set of saturated sets, i.e. those* $X \subseteq \mathrm{SN}$ *such that*
  - *closure under reduction:* $t \in X$ *and* $t > t'$ *imply* $t' \in X$;
  - *saturation:* $t \in \mathrm{SN}$ *and* $\mathrm{V}(t) \subseteq X$ *imply* $t \in X$.

**Lemma B.5** Sat *ordered by inclusion is a complete lattice, with the least element* $\perp$ *being the set of strongly normalizable terms which cannot reduce to a value, and with meet given by intersection.*

In particular there is a least fixed-point operator $\mathrm{lfp} : (\mathrm{Sat} \to_{mon} \mathrm{Sat}) \to_{mon} \mathrm{Sat}$, where $\mathrm{Sat} \to_{mon} \mathrm{Sat}$ is the family of monotonic functions on Sat.

**Definition B.6** *Given* $X_1, X_2 \in \mathrm{Sat}$*, the set* $(X_1 \to X_2) \in \mathrm{Sat}$ *is given by*

$$X_1 \to X_2 = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t). \exists t_2.(v \equiv \lambda x.t_2) \wedge (\forall t_1 \in X_1.[t_1/x]t_2 \in X_2)\}.$$

It is straightforward to prove that $(X_1 \to X_2) \in \mathrm{Sat}$.

**Definition B.7** *A function* $F : \mathrm{Sat}^m \to \mathrm{Sat}$ *is a* **functorial operator** *iff for any* $X_{i\in m}, Y_{i\in m} \in \mathrm{Sat}$ *the following properties hold*

- *monotonicity:* $(\forall i \in m.X_i \subseteq Y_i)$ *implies* $F(\overline{X}) \subseteq F(\overline{Y})$*;*
- *functoriality:* $(\forall i \in m.f_i \in (X_i \to Y_i))$ *and* $t \in F(\overline{X})$ *imply* $\mathrm{map}_m(\overline{f}, t) \in F(\overline{Y})$*.*

**Axiom B.8** *Fix a family of monotonic functions* $M_W : \mathrm{Sat} \to \mathrm{Sat}$*, indexed on sets* $W \in \mathrm{Sat}$*, such that for all* $W, X, Y \in \mathrm{Sat}$*, val* $\in X \to M_W(X)$*, let* $\in (X \to M_W(Y)) \to M_W(X) \to M_W(Y)$ *and fail* $\in M_W(X)$*.*

**Definition B.9** *A function* $F : \mathrm{Sat}^m \to \mathrm{Sat}$ *is a* **shape-based functorial operator***, iff it is a functorial operator and for any* $X_{i\in m}, Y_{i\in m}, Z_{i\in m} \in \mathrm{Sat}$ *and for any* $W \in \mathrm{Sat}$ *the following properties hold*

- $(\forall i \in m.f_i \in (X_i \to M_W(Y_i)))$ *and* $t \in F(\overline{X})$ *imply* $\mathrm{ltraverse}_m(\overline{f}, t) \in M_W(F(\overline{Y}))$*;*
- $(\forall i \in m.f_i \in (X_i \to Y_i \to M_W(Z_i)))$*,* $s \in F(\overline{X})$ *and* $t \in F(\overline{Y})$ *imply* $\mathrm{zipop}_m(\overline{f}, s, t) \in M_W(F(\overline{Z}))$*.*

$\mathrm{Sat}_m$ *is the set of shape-based functorial operators of arity* $m$*.*

### B.0.3 Semantic functions

This section introduce the semantic counterpart of functor constructors, i.e.

- $[\![1]\!] \in \mathrm{Sat}_0$ and $[\![\times]\!], [\![+]\!] \in \mathrm{Sat}_2$ interpret the functor constants;
- $\mathbf{P}_i^m \in \mathrm{Sat}_m$ interprets the projection $\Pi_i^m$;
- $\mathbf{C}_{m,n} : \mathrm{Sat}_m \to \mathrm{Sat}_n^m \to \mathrm{Sat}_n$ interprets functor composition $F\langle G_{i\in m}\rangle^n$;
- $\mathbf{m}_m : \mathrm{Sat}_{m+1} \to \mathrm{Sat}_m$ interprets the inductive functor construction $\mu F$.

The semantic counterpart of type constructors is *as expected* (see [Men91]), i.e.

- functional types are interpreted by saturated sets of the form $X_1 \to X_2$;
- universal quantification is interpreted by intersection (on Sat or $\mathrm{Sat}_m$);
- functor application is interpreted by function application.

After the definition of the semantics function, we sketch a proof that they have the expected codomain (this is tricky only for $\mathbf{m}_m$, where the proof mimics that for inductive types given in [Men91]).

**Definition B.10**

1. $[\![1]\!]() = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).v \equiv \mathrm{un}\}$.
2. $[\![\times]\!](X_0, X_1) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists t_0 \in X_0.\exists t_1 \in X_1.v \equiv \langle t_0, t_1\rangle\}$.
3. $[\![+]\!](X_0, X_1) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists i \in 2.\exists t' \in X_i.v \equiv \mathrm{in}_i(t')\}$.
4. $\mathbf{P}_i^m(\overline{X}) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists t' \in X_i.v \equiv \mathrm{tag}_{m,i}(t')\}$.
5. $\mathbf{C}_{m,n}(F, \overline{G})(\overline{X}) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists t' \in F(G_i(\overline{X})_{i \in m}).v \equiv \mathrm{pack}_{m,n}(t')\}$.
6. $\mathbf{m}_m(F)(\overline{X}) = \mathrm{lfp}\, H_{F,\overline{X}}$, *where* $H_{F,\overline{X}} : \mathrm{Sat} \to_{mon} \mathrm{Sat}$ *is*
   $H_{F,\overline{X}}(X) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists t' \in F(\overline{X}, X).v \equiv \mathrm{wrap}_m(t')\}$.

**Lemma B.11** *The semantics functions in Definition B.10 are well defined, i.e.*

1. $[\![1]\!] \in \mathrm{Sat}_0$.
2. $[\![\times]\!] \in \mathrm{Sat}_2$.
3. $[\![+]\!] \in \mathrm{Sat}_2$.
4. $\mathbf{P}_i^m \in \mathrm{Sat}_m$.
5. *If* $F \in \mathrm{Sat}_m$ *and* $G_{i \in m} \in \mathrm{Sat}_n$, *then* $\mathbf{C}_{m,n}(F, \overline{G}) \in \mathrm{Sat}_n$.
6. *If* $F \in \mathrm{Sat}_{m+1}$, *then* $\mathbf{m}_m(F) \in \mathrm{Sat}_m$.

**Proof.**

1. We have to prove that $[\![1]\!] \in \mathrm{Sat}_0$.

   - $[\![1]\!]() \in \mathrm{Sat}$. Closure under reduction: if $t \in [\![1]\!]()$, $t > t'$, and $v \in \mathrm{V}(t')$, then $v \in \mathrm{V}(t)$, and so $v \equiv \mathrm{un}$. Saturation: let $t \in \mathrm{SN}$ and $\mathrm{V}(t) \subseteq [\![1]\!]()$, then for all $v \in \mathrm{V}(t)$ we have $v \equiv \mathrm{un}$, hence $t \in [\![1]\!]()$.
   - Monotonicity is trivial since the arity of the functor is 0.
   - Functoriality:
     let $t \in [\![1]\!]()$. First we have to show that $\mathrm{map}_0(t) \in \mathrm{SN}$. Suppose it has an infinite reduction path. Then because $t$ is strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to un, and then the reduction yields un; contradiction. This shows also that all values of $\mathrm{map}_0(t)$ are of the form un and hence that $\mathrm{map}_0(t) \in [\![1]\!]()$.
   - Shapeliness:
     - Let $t \in [\![1]\!]()$. We have to show that $\mathrm{ltraverse}_0(t) \in M_W([\![1]\!]())$. Since $M_W([\![1]\!]()) \in \mathrm{Sat}$, by the saturation property, it is sufficient to show that $\mathrm{ltraverse}_0(t) \in \mathrm{SN}$ and that $\mathrm{V}(\mathrm{ltraverse}_0(t)) \subseteq M_W([\![1]\!]())$. So first we have to show that $\mathrm{ltraverse}_0(t) \in \mathrm{SN}$. Suppose it has an infinite reduction path. Then because $t$ is strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to un, and then the reduction yields [un]; since $\mathrm{un} \in [\![1]\!]()$, by Axiom B.8, $[\mathrm{un}] \in M_W([\![1]\!]())$ and thus is strongly normalizable, contradiction. This shows also that all values of $\mathrm{ltraverse}_0(t)$ are reducts of [un] and hence, by closure under reduction that they belong to $M_W([\![1]\!]())$.
     - Let $s \in [\![1]\!]()$ and $t \in [\![1]\!]()$. We have to show that $\mathrm{zipop}_0(s,t) \in M_W([\![1]\!]())$. Since $M_W([\![1]\!]()) \in \mathrm{Sat}$, by saturation property, it is sufficient to show that $\mathrm{zipop}_0(s,t) \in \mathrm{SN}$ and that $\mathrm{V}(\mathrm{zipop}_0(s,t)) \subseteq M_W([\![1]\!]())$. So first we have to show that $\mathrm{zipop}_0(s,t) \in \mathrm{SN}$. Suppose it has an infinite reduction path. Then because $s$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $s$ and $t$ have been both reduced to un, and then the reduction yields [un]; since $\mathrm{un} \in [\![1]\!]()$, by Axiom B.8, $[\mathrm{un}] \in M_W([\![1]\!]())$ and thus is strongly normalizable, contradiction. This shows also that all values of $\mathrm{zipop}_0(s,t)$ are reducts of [un] and hence, by closure under reduction that they belong to $M_W([\![1]\!]())$.

2. We have to prove that $[\![\times]\!] \in \mathrm{Sat}_2$.

   - $[\![\times]\!](X_0, X_1) \in \mathrm{Sat}$. Closure under reduction: it follows from the fact that $\mathrm{V}(t') \subseteq \mathrm{V}(t)$ whenever $t > t'$: let $t \in [\![\times]\!](X_0, X_1)$, if $t > t'$ then $t'$ is strongly normalizable and for all $v \in \mathrm{V}(t')$, there exist $t_0 \in X_0, t_1 \in X_1$, such that $v \equiv \langle t_0, t_1 \rangle$. Saturation: let $t \in \mathrm{SN}$, if $\mathrm{V}(t) \subseteq [\![\times]\!](X_0, X_1)$ then for all $v \in \mathrm{V}(t)$, there exist $t_0 \in X_0, t_1 \in X_1$, such that $v \equiv \langle t_0, t_1 \rangle$ and this is the required.

- Monotonicity: straightforward.
- Functoriality:
let $f_i \in (X_i \to Y_i)$, for $i \in 2$ and let $t \in [\![\times]\!](X_0, X_1)$. We are to show that $\mathrm{map}_2(f_0, f_1, t) \in [\![\times]\!](Y_0, Y_1)$. First we will show that $\mathrm{map}_2(f_0, f_1, t) \in \mathrm{SN}$. Suppose by absurd that there exists an infinite reduction path. Then because $f_0, f_1$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to a value. Since $t \in [\![\times]\!](X_0, X_1)$ then the value is of the form $\langle t_0, t_1 \rangle$. Therefore the outermost step yields $\langle f_0' t_0, f_1' t_1 \rangle$, where $f_0', f_1'$ are reducts respectively of $f_0, f_1$, which are strongly normalizable since $f_i'$ is in $(X_i \to Y_i)$. Now we show that if $v \in \mathrm{V}(\mathrm{map}_2(f_0, f_1, t))$ then there are terms $t_0 \in Y_0, t_1 \in Y_1$ such that $v \equiv \langle t_0, t_1 \rangle$. Consider such a $v$. It has been obtained from $\mathrm{map}_2(f_0, f_1, t)$ by a sequence of non-outermost reduction steps and, once $t$ has been reduced to a value, by an outermost reduction step. But $t$ can only reduce to a value of the form $\langle t_0, t_1 \rangle$ since $t$ is in $[\![\times]\!](X_0, X_1)$. So $v$ has the form $\langle t_0', t_1' \rangle$ for some $t_i'$ that are reducts of $f_i' t_i$. Since $f_i \in (X_i \to Y_i)$ it follows that $f_i' t_i \in Y_i$ and, by closure under reduction, we have that $t_i' \in Y_i$ as required.
- Shapeliness:
  - Let $f_i \in (X_i \to M_W(Y_i))$, for $i \in 2$ and let $t \in [\![\times]\!](X_0, X_1)$. We are to show that $\mathrm{ltraverse}_2(f_0, f_1, t) \in M_W([\![\times]\!](Y_0, Y_1))$. First we will show that $\mathrm{ltraverse}_2(f_0, f_1, t) \in \mathrm{SN}$. Suppose by absurd that there exists an infinite reduction path. Then because $f_0, f_1$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to a value. Since $t \in [\![\times]\!](X_0, X_1)$ then the value is of the form $\langle t_0, t_1 \rangle$. Therefore the outermost step yields $\mathrm{let}\, y_0 \Leftarrow f_0' \;\; t_0 \,\mathrm{in}\, \mathrm{let}\, y_1 \Leftarrow f_1' \;\; t_1 \,\mathrm{in}\, [\langle y_0, y_1 \rangle]$, where $f_0', f_1'$ are reducts respectively of $f_0, f_1$. Now $f_i' \;\; t_i \in M(Y_i)$ and for all $u_i \in Y_i$, $[\langle u_0, u_1 \rangle] \in M_W([\![\times]\!](Y_0, Y_1))$, by hypothesis and $[\![\times]\!](Y_0, Y_1)$ definition. By Axiom B.8 we can conclude that $\mathrm{let}\, y_0 \Leftarrow f_0 \; t_0 \,\mathrm{in}\, \mathrm{let}\, y_1 \Leftarrow f_1 \; t_1 \,\mathrm{in}\, [\langle y_0, y_1 \rangle] \in M_W([\![\times]\!](Y_0, Y_1))$ and thus that it is strongly normalizable. This show also that $\mathrm{let}\, y_0 \Leftarrow f_0 \; t_0 \,\mathrm{in}\, \mathrm{let}\, y_1 \Leftarrow f_1 \; t_1 \,\mathrm{in}\, [\langle y_0, y_1 \rangle] \in M([\![\times]\!](Y_0, Y_1))$. Since the values of $\mathrm{ltraverse}_2(f_0, f_1, t)$ are the same of $\mathrm{let}\, y_0 \Leftarrow f_0 \;\; t_0 \,\mathrm{in}\, \mathrm{let}\, y_1 \Leftarrow f_1 \;\; t_1 \,\mathrm{in}\, [\langle y_0, y_1 \rangle]$ we can conclude also that $\mathrm{ltraverse}_2(f_0, f_1, t)) \in M_W([\![\times]\!](Y_0, Y_1)))$.

- Let $f_i \in (X_i \to Y_i \to M_W(Z_i))$, for $i \in 2$ and let $s \in [\![\times]\!](X_0, X_1)$ and $t \in [\![\times]\!](Y_0, Y_1)$. We are to show that $\mathrm{zipop}_2(f_0, f_1, s, t) \in M_W([\![\times]\!](Z_0, Z_1))$. First we will show that $\mathrm{zipop}_2(f_0, f_1, s, t) \in \mathrm{SN}$. Suppose by absurd that there exists an infinite reduction path. Then because $f_0, f_1, s$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $s$ and $t$ have been reduced to a value. Since $s, t \in [\![\times]\!](X_0, X_1)$ then the values are of the form $\langle s_0, s_1\rangle$ and $\langle t_0, t_1\rangle$. Therefore the outermost step yields $\mathrm{let}\, z_0 \Leftarrow (f'_0\ s_0\ t_0)\, \mathrm{in}\, \mathrm{let}\, z_1 \Leftarrow (f'_1\ s_1\ t_1)\, \mathrm{in}\, [\langle z_0, z_1\rangle]$, where $f'_0, f'_1$ are reducts respectively of $f_0, f_1$. Now $f'_i s_i t_i \in M(Z_i)$ and $[\langle u_0, u_1\rangle] \in M_W([\![\times]\!](Z_0, Z_1))$, for all $u_i \in Z_i$ by hypothesis and $[\![\times]\!]$ definitions. By Axiom B.8 we can conclude that $\mathrm{let}\, z_0 \Leftarrow (f_0\ s_0\ t_0)\, \mathrm{in}\, \mathrm{let}\, z_1 \Leftarrow (f_1\ s_1\ t_1)\, \mathrm{in}\, [\langle z_0, z_1\rangle] \in M([\![\times]\!](Z_0, Z_1))$ and thus is strongly normalizable. This show also that $\mathrm{let}\, z_0 \Leftarrow (f_0\ s_0\ t_0)\, \mathrm{in}\, \mathrm{let}\, z_1 \Leftarrow (f_1\ s_1\ t_1)\, \mathrm{in}\, [\langle z_0, z_1\rangle] \in M_W([\![\times]\!](Z_0, Z_1))$. Since the values of $\mathrm{zipop}_2(f_0, f_1, s, t)$ are the same of $\mathrm{let}\, z_0 \Leftarrow (f_0\ s_0\ t_0)\, \mathrm{in}\, \mathrm{let}\, z_1 \Leftarrow (f_1\ s_1\ t_1)\, \mathrm{in}\, [\langle z_0, z_1\rangle]$ we can conclude also that $\mathrm{zipop}_2(f_0, f_1, s, t) \in M_W([\![\times]\!](Z_0, Z_1))$.

3. Proceed as in the case for $[\![\times]\!]$.

4. Proceed as in the case for $[\![\times]\!]$.

5. We have to prove that $\mathbf{C}_{m,n}(F, \overline{G}) \in \mathrm{Sat}_n$.

   - $\mathbf{C}_{m,n}(F, \overline{G})(\overline{X}) \in \mathrm{Sat}$. Closure under reduction and saturation: proceed as in the case for $[\![\times]\!]$.
   - Monotonicity: straightforward.
   - Functoriality:
     let $Y_{j \in n}$, $f_j \in (X_j \to Y_j)$, for $j \in n$ and let $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$. We are to show that $\mathrm{map}_n(\overline{f}, t) \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{Y})$. First we show that it is strongly normalizable. Suppose by absurd that there exists an infinite reduction path. Then because $f_{j \in n}$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to value. Since $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$ then $t$ has been reduced to a value of the form $\mathrm{pack}_{m,n}(t_1)$. Therefore the outermost step yields $(\mathrm{pack}_{m,n}(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i \in m}, t_1))$.
     Now suppose that $\lambda x.\mathrm{map}_n(\overline{f}, x) \in G_i(\overline{X}) \to G_i(\overline{Y})$ for all $i \in m$. Clearly $t_1 \in F(G_i(\overline{X})_{i \in m})$. Then we have that $\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i \in m}, t_1) \in F(G_i(\overline{Y})_{i \in m})$ by functoriality of $F$, which gives the desired contradiction.

We are left with showing that $(\lambda x.\mathrm{map}_n(\overline{f}, x))$ is in $G_i(\overline{X}) \to G_i(\overline{Y})$ for all $i \in m$. Let $i$ be arbitrary. Then $\lambda x.\mathrm{map}_n(\overline{f}, x)$ is in SN since reduction can only occur inside the subterm $f$ which by hypothesis is strongly normalizable. Consider a value $v$ of $\lambda x.\mathrm{map}_n(\overline{f}, x)$. This $v$ is of the form $\lambda x.\mathrm{map}_n(\overline{f'}, x)$, where each $f'_j$ is a reduct of the corresponding $f_j$. By the closure condition we have that $f'_j \in X_j \to Y_j$. By functoriality of $G_i$, for all $t \in G_i(\overline{X})$ we have that $\mathrm{map}_n(\overline{f'}, t) \in G_i(\overline{Y})$. Now we show that if $v \in \mathrm{V}(\mathrm{map}_n(\overline{f}, t))$ then there is a term $t_1 \in F(G_i(\overline{X})_{i \in m})$ such that $v \equiv \mathrm{pack}_{m,n}(t_1)$. Consider such a $v$. It has been obtained from $\mathrm{map}_n(\overline{f}, t)$ by a sequence of non-outermost reduction steps and, once that $t$ has been reduced to a value, by an outermost reduction step. But $t$ can only reduce to a value of the form $\mathrm{pack}_{m,n}(t_1)$ since $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$. So $v$ has the form $\mathrm{pack}_{m,n}(t_2)$ for some $t_2$ that are reducts of $(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i \in m}, t_1)$. Since $\lambda x.\mathrm{map}_n(\overline{f}, x) \in G_i(\overline{X}) \to G_i(\overline{Y})$ for all $i \in m$ and $t_1 \in F(G_i(\overline{X})_{i \in m})$, by functoriality of $F$ we can conclude that $(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i \in m}, t_1) \in F(G_i(\overline{Y})_{i \in m})$. By closure under reduction we have $t_2 \in F(G_i(\overline{Y})_{i \in m})$.

- Shapeliness:
  - Let $Y_{j \in n}$, $f_j \in (X_j \to M_W(Y_j))$, for $j \in n$ and let $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$. We are to show that $\mathrm{ltraverse}_n(\overline{f}, t) \in M_W(\mathbf{C}_{m,n}(F, \overline{G})(\overline{Y}))$. First we show that it is strongly normalizable. Suppose by absurd that there exists an infinite reduction path. Then because $f_{j \in n}$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to value. Since $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$ then $t$ has been reduced to a value of the form $\mathrm{pack}_{m,n}(t_1)$. Therefore the outermost step yields $\mathrm{let}\, y \Leftarrow \mathrm{ltraverse}_m\ (\lambda x.\mathrm{ltraverse}_n\ \overline{f'}, x)_{i:m}\ t_1 \,\mathrm{in}\, [\mathrm{pack}_{m,n}\ y]$, for some $\overline{f}'$ reducts of $\overline{f}$.

    Now suppose that $\lambda x.\mathrm{ltraverse}_n(\overline{f'}, x) \in G_i(\overline{X}) \to M_W(G_i(\overline{Y}))$ for all $i \in m$. Clearly $t_1 \in F(G_i(\overline{X})_{i \in m})$. Then we have that $\mathrm{ltraverse}_m((\lambda x.\mathrm{ltraverse}_n(\overline{f'}, x))_{i \in m}, t_1) \in M_W(F(G_i(\overline{Y})_{i \in m}))$ by shapeliness of $F$, which gives that $\mathrm{let}\, y \Leftarrow \mathrm{ltraverse}_m\ (\lambda x.\mathrm{ltraverse}_n\ \overline{f'}, x)_{i:m}\ t_1 \,\mathrm{in}\, [\mathrm{pack}_{m,n}\ y] \in M_W(\mathbf{C}_{m,n}(F, \overline{G})(\overline{Y}))$ and thus the desired contradiction.

    We are left with showing that $(\lambda x.\mathrm{ltraverse}_n(\overline{f'}, x))$ is in $G_i(\overline{X}) \to M_W(G_i(\overline{Y}))$ for all $i \in m$. Let $i$ be arbitrary. Then $\lambda x.\mathrm{ltraverse}_n(\overline{f'}, x)$ is in SN since reduction can only occur inside the subterm $f$ which by hypothesis is strongly normalizable. Consider a value $v$ of $\lambda x.\mathrm{ltraverse}_n(\overline{f'}, x)$. This $v$ is of the form $\lambda x.\mathrm{ltraverse}_n(\overline{f''}, x)$, where each $f''_j$ is a reduct of the corresponding $f'_j$. By the closure condition we have that

$f''_j \in X_j \to M_W(Y_j)$. By shapeliness of $G_i$, for all $t \in G_i(\overline{X})$ we have that $\text{ltraverse}_n(\overline{f'}, t) \in M_W(G_i(\overline{Y}))$. This show also that $\text{let}\, y \Leftarrow \text{ltraverse}_m \quad (\lambda x.\text{ltraverse}_n \quad \overline{f}, x)_{i:m} \quad t_1 \,\text{in}\, [\text{pack}_{m,n} \quad y] \in M_W(\mathbf{C}_{m,n}(F, \overline{G})(\overline{Y}))$. And since values of this term are the same to the values of $M_W(\mathbf{C}_{m,n}(F, \overline{G})(\overline{Y}))$ we have the desired result.

- Similarly for $\text{zipop}_m$.

6. We have to prove that $\mathbf{m}_m(F) \in \text{Sat}_m$. We show that:

   (a) $H_{F,\overline{X}}$ is well-defined, i.e. $\forall X \in \text{Sat}.H_{F,\overline{X}}(X) \in \text{Sat}$.

   (b) $H_{F,\overline{X}} : \text{Sat} \to_{mon} \text{Sat}$ is monotone; hence there exists the least fixed-point.

   (c) $\mathbf{m}_m(F)$ satisfies monotonicity, functoriality and shapeliness.

   (a) We proceed as in the case for $[\![\times]\!]$.

   (b) Straightforward.

   (c) • Monotonicity:
   follows from monotonicity of the least fixed-point operator.

   • Functoriality:
   Let $\overline{X}$ and $\overline{Y}$ be fixed. We have to prove that for all $i \in m$, $f_i \in (X_i \to Y_i)$ and that $t \in \mathbf{m}_m(F)(\overline{X})$ imply $\text{map}_m(\overline{f}, t) \in \mathbf{m}_m(F)(\overline{Y})$.
   The chain for the least fixed-point $\mathbf{m}_m(F)(\overline{X})$ is

$$\begin{array}{lcl} \xi_0 & = & \bot \\ \xi_{\alpha+1} & = & \{t \in \text{SN} | \forall v \in \text{V}(t).\exists t' \in F(\overline{X}, \xi_\alpha).v \equiv \text{wrap}_m(t')\} \\ \xi_\lambda & = & \bigcup_{\alpha<\lambda} \xi_\alpha \quad \text{for limit } \lambda. \end{array}$$

   The chain for the least fixed-point $\mathbf{m}_m(F)(\overline{Y})$ is

$$\begin{array}{lcl} \eta_0 & = & \bot \\ \eta_{\alpha+1} & = & \{t \in \text{SN} | \forall v \in \text{V}(t).\exists t' \in F(\overline{Y}, \eta_\alpha).v \equiv \text{wrap}_m(t')\} \\ \eta_\lambda & = & \bigcup_{\alpha<\lambda} \eta_\alpha \quad \text{for limit } \lambda. \end{array}$$

   By ordinal induction we prove that fixed $X_i$ and $Y_i$, for all $\alpha$ we have that $\forall (f_i \in X_i \to Y_i)_{i \in m}.\forall t \in \xi_\alpha.\text{map}_m(\overline{f}, t) \in \eta_\alpha$.
   *Basis:* Let $t \in \xi_0$. Since $t$ cannot reduce to any value, neither can the term $\text{map}_m(\overline{f}, t)$. Hence $\text{map}_m(\overline{f}, t) \in \bot = \eta_0$.
   *Inductive Case:* Let $t \in \xi_{\alpha+1}$. We have to show that $\text{map}_m(\overline{f}, t) \in \eta_{\alpha+1}$:

   - $\text{map}_m(\overline{f}, t) \in \text{SN}$. Suppose by absurd that there exists an infinite reduction path. Then because $f_{i \in m}$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to a value. Since $t \in \xi_{\alpha+1}$ then the value is of the form $\text{wrap}_m(t_1)$. Therefore the outermost step yields $\text{wrap}_m(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1))$, where each $f'_i$ is a reduct of the corresponding $f_i$.

We can conclude provided we show that $(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1)) \in F(\overline{Y}, \eta_\alpha)$.

Consider $\lambda x.\text{map}_m(\overline{f'}, x)$; this term belongs to $\xi_\alpha \to \eta_\alpha$:

* $\lambda x.\text{map}_m(\overline{f'}, x) \in \text{SN}$: follows from the fact that a reduction can only occur inside the subterms $f'_i$ which by hypothesis are strongly normalizable.
* Consider a value $v$ of $\lambda x.\text{map}_m(\overline{f'}, x)$. This $v$ is of the form $\lambda x.\text{map}_m(\overline{f''}, x)$, where each $f''_i$ is a reduct of the corresponding $f'_i$. By the closure condition we have that $f''_i \in X_i \to Y_i$. By induction we get that $\forall t \in \xi_\alpha.\text{map}_m(\overline{f''}, t) \in \eta_\alpha$.

Since $f'_i \in (X_i \to Y_i)$, for all $i \in m$, $\lambda x.\text{map}_m(\overline{f'}, x) \in \xi_\alpha \to \eta_\alpha$ and $t_1 \in F(\overline{X}, \xi_\alpha)$, we have
$(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1)) \in F(\overline{Y}, \eta_\alpha)$ by functoriality of $F$

– Now we show that if $v \in \text{V}(\text{map}_m(\overline{f}, t))$ then there is a term $t' \in F(\overline{X}, \eta_\alpha)$ such that $v \equiv \text{wrap}_m(t')$.
Consider such a $v$. It has been obtained from $\text{map}_m(\overline{f}, t)$ by a sequence of non-outermost reduction steps and once $t$ has been reduced to a value, by an outermost reduction step. But $t$ can only reduce to a value of the form $\text{wrap}_m(t_1)$. So $v$ has the form $\text{wrap}_m(t_2)$ where $t_2$ is a reduct of $(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1))$.
Since $(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1)) \in F(\overline{Y}, \eta_\alpha)$, by closure under reduction we can conclude.

*Limit Case:* Let $t \in \xi_\lambda$. Then $t \in \xi_\alpha$ for some $\alpha \in \lambda$. By inductive hypothesis $(\text{map}_m(\overline{f}, t)) \in \eta_\alpha$. Hence $(\text{map}_m(\overline{f}, t)) \in \eta_\alpha \subseteq \eta_\lambda$.

- Shapeliness:
  – Let $\overline{X}$ and $\overline{Y}$ be fixed. We have to prove that for all $i \in m$, $f_i \in (X_i \to M_W(Y_i))$ and that $t \in \mathbf{m}_m(F)(\overline{X})$ imply $\text{ltraverse}_m(\overline{f}, t) \in M_W(\mathbf{m}_m(F)(\overline{Y}))$.
  The chains for the least fixed-points $\mathbf{m}_m(F)(\overline{X})$ and $\mathbf{m}_m(F)(\overline{Y})$ are defined as above.
  By ordinal induction we prove that fixed $X_i$ and $Y_i$, for all $\alpha$ we have that $\forall (f_i \in X_i \to M_W(Y_i))_{i \in m}.\forall t \in \xi_\alpha.\text{ltraverse}_m(\overline{f}, t) \in M_W(\eta_\alpha)$.
  *Basis:* Let $t \in \xi_0$. Since $t$ cannot reduce to any value, neither can the term $\text{ltraverse}_m(\overline{f}, t)$. Hence $\text{ltraverse}_m(\overline{f}, t) \in \perp \subseteq M_W(\eta_0)$, since $\perp$ is the bottom.

*Inductive Case:* Let $t \in \xi_{\alpha+1}$. We have to show that $\text{ltraverse}_m(\overline{f}, t) \in M_W(\eta_{\alpha+1})$:

* $\text{ltraverse}_m(\overline{f}, t) \in \text{SN}$. Suppose by absurd that there exists an infinite reduction path. Then because $f_{i \in m}$ and $t$ are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when $t$ has been reduced to a value. Since $t \in \xi_{\alpha+1}$ then the value is of the form $\text{wrap}_m(t_1)$. Therefore the outermost step yields $\text{let}\, y \Leftarrow \text{ltraverse}_{m+1}\ \overline{f'}\ (\lambda x.\text{ltraverse}_m\ \overline{f'} x)\ t_1 \,\text{in}\, [\text{wrap}_m\ y]$, where each $f'_i$ is a reduct of the corresponding $f_i$. We can conclude provided we show that $(\text{ltraverse}_{m+1}(\overline{f'}, \lambda x.\text{ltraverse}_m(\overline{f'}, x), t_1)) \in M_W(F(\overline{Y}, \eta_\alpha))$. Consider $\lambda x.\text{ltraverse}_m(\overline{f'}, x)$. This term belongs to $\xi_\alpha \to M_W(\eta_\alpha)$:
  · $\lambda x.\text{ltraverse}_m(\overline{f'}, x) \in \text{SN}$: follows from the fact that a reduction can only occur inside the subterms $f'_i$ which by hypothesis are strongly normalizable.
  · Consider a value $v$ of $\lambda x.\text{ltraverse}_m(\overline{f'}, x)$. This $v$ is of the form $\lambda x.\text{ltraverse}_m(\overline{f''}, x)$, where each $f''_i$ is a reduct of the corresponding $f'_i$. By the closure condition we have that $f''_i \in X_i \to M_W(Y_i)$. By induction we get that $\forall t \in \xi_\alpha.\text{ltraverse}_m(\overline{f''}, t) \in M_W(\eta_\alpha)$.

  Since $f'_i \in (X_i \to Y_i)$, for all $i \in m$, $\lambda x.\text{ltraverse}_m(\overline{f'}, x) \in \xi_\alpha \to M_W(\eta_\alpha)$ and $t_1 \in F(\overline{X}, \xi_\alpha)$, we have $(\text{ltraverse}_{m+1}(\overline{f'}, \lambda x.\text{ltraverse}_m(\overline{f'}, x), t_1)) \in M_W(F(\overline{Y}, \eta_\alpha))$ by shapeliness of $F$
* The previous point shows that $\text{let}\, y \Leftarrow \text{ltraverse}_{m+1}\ \overline{f'}\ (\lambda x.\text{ltraverse}_m\ \overline{f'} x)\ t_1 \,\text{in}\, [\text{wrap}_m\ y] \in M_W(F(\overline{Y}, \eta_\alpha))$. Since the values of $\text{ltraverse}_m(\overline{f}, t)$ are the same of $\text{let}\, y \Leftarrow \text{ltraverse}_{m+1}\ \overline{f'}\ (\lambda x.\text{ltraverse}_m\ \overline{f'} x)\ t_1 \,\text{in}\, [\text{wrap}_m\ y]$ it follows that
  $\text{ltraverse}_m(\overline{f}, t) \in M_W(F(\overline{Y}, \eta_\alpha))$.

*Limit Case:* Let $t \in \xi_\lambda$. Then $t \in \xi_\alpha$ for some $\alpha \in \lambda$. By inductive hypothesis $(\text{ltraverse}_m(\overline{f}, t)) \in M_W(\eta_\alpha)$. Hence $\text{ltraverse}_m(\overline{f}, t) \in M_W(\eta_\alpha) \subseteq M_W(\eta_\lambda)$, because $M_W$ is monotone.

– Similarly for $\text{zipop}_m$

∎

### B.0.4 Interpretation

Using the semantic functions we define an interpretation of well-formed type contexts, functors and types by induction on the derivation of $\Delta \vdash J$.

- If $\Delta \vdash$, then $[\![\Delta]\!]$ is a set

$$\begin{array}{rcl} [\![\emptyset]\!] & = & 1 \\ [\![\Delta, X : n]\!] & = & [\![\Delta]\!] \times \mathrm{Sat}_n \\ [\![\Delta, X : \mathrm{T}]\!] & = & [\![\Delta]\!] \times \mathrm{Sat} \end{array}$$

- If $\Delta \vdash F : n$, then $[\![\Delta.F]\!] : [\![\Delta]\!] \to \mathrm{Sat}_n$

$$\begin{array}{rcl} [\![\Delta.X]\!]\rho & = & \rho_X \\ [\![\Delta.C]\!]\rho & = & [\![C]\!] \\ [\![\Delta.\Pi_i^m]\!]\rho & = & \mathbf{P}_i^m \\ [\![\Delta.F\langle\overline{G}\rangle^n]\!]\rho & = & \mathbf{C}_{m,n}(H, \overline{K}) \\ [\![\Delta.\mu F']\!]\rho & = & \mathbf{m}_m(H') \end{array}$$

  where $H = [\![\Delta.F]\!]\rho \in \mathrm{Sat}_m$, $H' = [\![\Delta.F']\!]\rho \in \mathrm{Sat}_{m+1}$ and $K_i = [\![\Delta.G_i]\!]\rho \in \mathrm{Sat}_n$.

- If $\Delta \vdash \tau$, then $[\![\Delta.\tau]\!] : [\![\Delta]\!] \to \mathrm{Sat}$

$$\begin{array}{rcl} [\![\Delta.X]\!]\rho & = & \rho_X \\ [\![\Delta.F(\overline{\tau})]\!]\rho & = & H(\overline{Y}) \\ [\![\Delta.\tau_1 \to \tau_2]\!]\rho & = & Y_1 \to Y_2 \\ [\![\Delta.(\forall X : m.\tau)]\!]\rho & = & \displaystyle\bigcap_{H \in \mathrm{Sat}_m} [\![\Delta\ X : m.\tau]\!]\langle\rho, H\rangle \\ [\![\Delta.(\forall X : \mathrm{T}.\tau)]\!]\rho & = & \displaystyle\bigcap_{Z \in \mathrm{Sat}} [\![\Delta\ X : \mathrm{T}.\tau]\!]\langle\rho, Z\rangle \end{array}$$

  where $H = [\![\Delta.F]\!]\rho \in \mathrm{Sat}_m$, $Y_i = [\![\Delta.\tau_i]\!]\rho \in \mathrm{Sat}$.

- If $\Delta; \Gamma \vdash$, then $[\![\Delta.\Gamma]\!] : [\![\Delta]\!] \to \mathrm{DV}(\Gamma) \to \mathrm{Sat}$

$$[\![\Delta.\Gamma]\!]\rho\ x = [\![\Delta.\tau]\!]\rho \quad \text{where } \tau = \Gamma(x).$$

### B.0.5 Strong normalization proof

The following lemma shows that the terms are well behaved with respect to the semantic functions. We will need it to prove the Proposition B.13.

**Lemma B.12** *Let* $W, X, Y, X_{i\in m}, Y_{i\in m}, Z_{j\in n} \in \mathrm{Sat}$, $F \in \mathrm{Sat}_m$, $G_{i\in m} \in \mathrm{Sat}_n$ *and* $K \in \mathrm{Sat}_{m+1}$. *The following hold:*

1. $\mathrm{intro}^1 \in [\![1]\!]()$;
2. *if* $\forall t_1 \in X_1.[t_1/x]t \in X_2$ *then* $\lambda x.t \in (X_1 \to X_2)$;
3. *if* $t \in X_1 \to X_2$ *and* $t_1 \in X_1$ *then* $tt_1 \in X_2$;
4. *if for all* $i \in 2$, $t_i \in X_i$, *then* $\mathrm{intro}^x(t_0, t_1) \in [\![\times]\!](X_0, X_1)$; *]*
5. *if* $t \in [\![\times]\!](X_0, X_1)$ *and* $f \in X_0 \to X_1 \to Y$ *then* $\mathrm{elim}^x(f, t) \in Y$;
6. *for all* $i \in 2$, *if* $t \in X_i$ *then* $\mathrm{intro}_i^+(t) \in [\![+]\!](X_0, X_1)$;
7. *if for all* $i \in 2$, $t \in [\![+]\!](X_0, X_1)$ *and* $f_i \in X_i \to Y$ *then* $\mathrm{elim}^+(f_0, f_1, t) \in Y$;
8. *for all* $i \in m$, *if* $t \in X_i$ *then* $\mathrm{intro}_{m,i}^{\Pi}(t) \in \mathbf{P}_i^m(\overline{X})$;
9. *for all* $i \in m$, *if* $t \in \mathbf{P}_i^m(\overline{X})$ *and* $f \in X_i \to Y$ *then* $\mathrm{elim}_{m,i}^{\Pi}(f, t) \in Y$;
10. *if* $t \in F(G_i(\overline{Z})_{i\in m})$ *then* $\mathrm{intro}_{m,n}^{\circ}(t) \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{Z})$;
11. *if* $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{Z})$ *and* $f \in F(G_i(\overline{Z})_{i\in m} \to Y$ *then* $\mathrm{elim}_{m,n}^{\circ}(t) \in Y$;
12. *if* $t \in K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$ *then* $\mathrm{intro}_m^{\mu}(t) \in \mathbf{m}_m(K)(\overline{X})$;
13. *if* $f \in (K(\overline{X}, Y) \to Y)$ *and* $t \in \mathbf{m}_m(K)(\overline{X})$ *then* $\mathrm{elim}_m^{\mu}(f, t) \in Y$;
14. *if for all* $i \in m, f_i \in (X_i \to Y_i)$ *and* $t \in F(\overline{X})$ *then* $\mathrm{map}_m(\overline{f}, t) \in F(\overline{Y})$;
15. *if for all* $i \in m, f_i \in (X_i \to M_W(Y_i))$ *and* $t \in F(\overline{X})$ *then* $\mathrm{ltraverse}_m(\overline{f}, t) \in M_W(F(\overline{Y}))$;
16. *if for all* $i \in m, f_i \in (X_i \to Y_i \to M_W(Z_i))$, $s \in F(\overline{X})$ *and* $t \in F(\overline{Y})$ *then* $\mathrm{zipop}_m(\overline{f}, s, t) \in M_W(F(\overline{Z}))$.

**Proof.** The proof is given only for the last five cases (for the others is similar).

- Given $K \in \mathrm{Sat}_{m+1}$, $X_{i\in m} \in \mathrm{Sat}$ and $Y \in \mathrm{Sat}$, if $t \in K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$ then $\mathrm{wrap}_m(t) \in \mathbf{m}_m(K)(\overline{X})$:

  Since $\mathbf{m}_m(K)(\overline{X}) = H_{K,\overline{X}}(\mathbf{m}_m(K)(\overline{X}))$, where $H_{K,\overline{X}}$ is the monotonic map in the definition of $\mathbf{m}_m(K)$, we prove that $\mathrm{wrap}_m(t) \in H_{K,\overline{X}}(\mathbf{m}_m(K)(\overline{X}))$. We have that $\mathrm{wrap}_m(t) \in \mathrm{SN}$ since $t \in \mathrm{SN}$. Now, a value of $\mathrm{wrap}_m(t)$ is of the form $\mathrm{wrap}_m(t')$, where $t'$ is a reduct of $t$. Since $t' \in K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$ we have that $\mathrm{wrap}_m(t) \in H_{K,\overline{X}}(\mathbf{m}_m(K)(\overline{X})) = \mathbf{m}_m(K)(\overline{X})$.

- Given $K \in \mathrm{Sat}_{m+1}$, $X_{i\in m} \in \mathrm{Sat}$ and $Y \in \mathrm{Sat}$, if $f \in (K(\overline{X}, Y) \to Y)$ and $t \in \mathbf{m}_m(K)(\overline{X})$, then $\mathrm{fold}_m(f, t) \in Y$:

  Let $(\xi_\alpha)_{\alpha \in Ord}$ be the chain for the least fixed-point $\mathbf{m}_m(K)(\overline{X})$, we prove by induction on $\alpha$ that
  $$\forall f \in (K(\overline{X}, Y) \to Y).\forall t \in \xi_\alpha.\mathrm{fold}_m(f, t) \in Y$$

  *Basis:* Let $f$ be a term in $K(\overline{X}, Y) \to Y$. If $t \in \xi_0$ then $\mathrm{fold}_m(f, t)$ cannot reduce to a value so it belongs to $\perp \subseteq Y$.

  *Inductive Case:* Suppose that $t \in \xi_{\alpha+1}$. Let $f \in (K(\overline{X}, Y) \to Y)$. If $t$ does not reduce to a value then neither $\mathrm{fold}_m(f, t)$ cannot reduce to a value, so $\mathrm{fold}_m(f, t)$ belongs to $\perp \subseteq Y$. Otherwise, $t$ reduces to a value of the form $\mathrm{wrap}_m(t_1)$, where $t_1 \in K(\overline{X}, \xi_\alpha)$. Hence $\mathrm{fold}_m(f, t)$ reduces to $f'(\mathrm{map}_{m+1}((\lambda x.x)_{i\in m}, \lambda x.\mathrm{fold}_m(f', x), t_1))$ where $f'$ is a reduct of $f$. Using inductive hypothesis we get $\lambda x.\mathrm{fold}_m(f', x) \in (\xi_\alpha \to Y)$. Functoriality of $K$ allows to conclude that $(\mathrm{map}_{m+1}((\lambda x.x)_{i\in m}, \lambda x.\mathrm{fold}_m(f', x), t_1)) \in K(\overline{X}, Y)$. Since $f \in (K(\overline{X}, Y) \to Y)$ by hypothesis, we have $\mathrm{fold}_m(f, t) \in Y$.

  *Limit Case:* Suppose that $t \in \xi_\lambda$. Then $t \in \xi_\alpha$ for some $\alpha \in \lambda$. By inductive hypothesis, we have $\mathrm{fold}_m(f, t) \in Y$, since $f \in (K(\overline{X}, Y) \to Y)$.

- Given $F \in \mathrm{Sat}_m$ and $X_{i\in m}, Y_{i\in m} \in \mathrm{Sat}$, if $f_i \in (X_i \to Y_i)$ and $t \in F(\overline{X})$, then $\mathrm{map}_m(\overline{f}, t) \in F(\overline{Y})$:

  Immediate from the definition of $F \in \mathrm{Sat}_m$.

- Given $F \in \mathrm{Sat}_m$ and $W, X_{i\in m}, Y_{i\in m} \in \mathrm{Sat}$, if $f_i \in (X_i \to M_W(Y_i))$ and $t \in F(\overline{X})$, then $\mathrm{ltraverse}_m(\overline{f}, t) \in M_W(F(\overline{Y}))$:

  Immediate from the definition of $F \in \mathrm{Sat}_m$.

- Given $F \in \mathrm{Sat}_m$ and $W, X_{i\in m}, Y_{i\in m}, Z_{i\in m} \in \mathrm{Sat}$, if $f_i \in (X_i \to Y_i \to M_W(Z_i))$, $s \in F(\overline{X})$ and $t \in F(\overline{Y})$, then $\mathrm{zipop}_m(\overline{f}, s, t) \in M_W(F(\overline{Z}))$:

  Immediate from the definition of $F \in \mathrm{Sat}_m$.

∎

**Proposition B.13** *Suppose that if $\Delta \vdash P_\upsilon(\tau)$ then for any $\rho \in [\![\Delta]\!]$,*

$$[\![\Delta.P_\upsilon(\tau)]\!]\rho = M_{[\![\Delta.\upsilon]\!]\rho}([\![\Delta.\tau]\!]\rho).$$

*If $\Delta; \Gamma \vdash t : \tau$ is derivable in $\mathrm{FF}_{shape}$, then $S(t) \in [\![\Delta.\tau]\!]\rho$ for any $\rho \in [\![\Delta]\!]$ and term substitution $S$, such that $S\ x \ \in [\![\Delta.\Gamma]\!]\rho\ x$, with $x \in \mathrm{DV}(\Gamma)$.*

**Proof.** The proof is by induction on the derivation of the judgment $\Delta; \Gamma \vdash t : \tau$. Here are the justifications for the rules $\mathrm{intro}^\mu_m$, $\mathrm{elim}^\mu_m$, $\mathrm{map}_m$, $\mathrm{ltraverse}_m$ and $\mathrm{zipop}_m$.

- $\mathrm{intro}^\mu_m$ $\dfrac{\begin{array}{c}\Delta \vdash F : m+1 \quad \Delta \vdash \tau_{i \in m} \\ \Delta; \Gamma \vdash t : F(\overline{\tau}, \mu F(\overline{\tau}))\end{array}}{\Delta; \Gamma \vdash \mathrm{intro}^\mu_m(t) : \mu F(\overline{\tau})}$

  By inductive hypothesis

  $$S(t) \in [\![\Delta.F(\overline{\tau}, \mu F(\overline{\tau}))]\!]\rho = K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$$

  where $K = [\![\Delta.F]\!]\rho$ and $X_i = [\![\Delta.\tau_i]\!]\rho$. By Lemma B.12 we can conclude

  $$S(\mathrm{intro}^\mu_m(t)) \equiv \mathrm{intro}^\mu_m(S(t)) \in \mathbf{m}_m(K)(\overline{X}) = [\![\Delta.\mu F(\overline{\tau})]\!]\rho$$

- $\mathrm{elim}^\mu_m$ $\dfrac{\begin{array}{c}\Delta \vdash F : m+1 \quad \Delta \vdash \tau_{i \in m}, \sigma \\ \Delta; \Gamma \vdash f : F(\overline{\tau}, \sigma) \to \sigma \quad \Delta; \Gamma \vdash t : \mu F(\overline{\tau})\end{array}}{\Delta; \Gamma \vdash \mathrm{elim}^\mu_m(f, t) : \sigma}$

  By inductive hypothesis $S(f) \in K(\overline{X}, Y) \to Y$ and $S(t) \in \mathbf{m}_m(K)(\overline{X})$, where $K = [\![\Delta.F]\!]\rho$, $X_i = [\![\Delta.\tau_i]\!]\rho$ and $Y = [\![\Delta.\sigma]\!]\rho$.

  By Lemma B.12 we can conclude

  $$S(\mathrm{elim}^\mu_m(f, t)) \equiv \mathrm{elim}^\mu_m(S(f), S(t)) \in Y = [\![\Delta.\sigma]\!]\rho$$

- $\mathrm{map}_m$ $\dfrac{\begin{array}{c}\Delta \vdash F : m \quad \Delta \vdash \tau_{i \in m}, \sigma_{i \in m} \\ \Delta; \Gamma \vdash f_i : \tau_i \to \sigma_i \quad \Delta; \Gamma \vdash t : F(\overline{\tau})\end{array}}{\Delta; \Gamma \vdash \mathrm{map}_m(\overline{f}, t) : F(\overline{\sigma})}$

  By inductive hypothesis $S(f_i) \in X_i \to Y_i$ (for $i \in m$) and $S(t) \in H(\overline{X})$, where $H = [\![\Delta.F]\!]\rho$, $X_i = [\![\Delta.\tau_i]\!]\rho$ and $Y_i = [\![\Delta.\sigma_i]\!]\rho$.

  by Lemma B.12 we can conclude

  $$S(\mathrm{map}_m(\overline{f}, t)) \equiv \mathrm{map}_m((S(f_i))_{i \in m}, S(t)) \in H(\overline{Y}) = [\![\Delta.F(\overline{\sigma})]\!]$$

- $\text{ltraverse}_m$ $\dfrac{\begin{array}{c}\Delta \vdash F : m \quad \Delta \vdash \tau_{i \in m}, \sigma_{i \in m}, \upsilon \\ \Delta; \Gamma \vdash f_i : \tau_i \to P_\upsilon(\sigma_i) \quad \Delta; \Gamma \vdash t : F(\overline{\tau})\end{array}}{\Delta; \Gamma \vdash \text{ltraverse}_m(\overline{f}, t) : P_\upsilon(F(\overline{\sigma}))}$

  By inductive hypothesis $S(t) \in H(\overline{X})$ and $S(f_i) \in X_i \to M_W(Y_i)$ (for $i \in m$), where $H = [\![\Delta.F]\!]\rho$, $W = [\![\Delta.\upsilon]\!]\rho$, $X_i = [\![\Delta.\tau_i]\!]\rho$ and $Y_i = [\![\Delta.\sigma_i]\!]\rho$.

  By Lemma B.12
  $$\text{ltraverse}_m((S(f_i))_{i \in m}, S(t)) \in M_W(H(\overline{Y}))$$
  thus
  $$S(\text{ltraverse}_m(\overline{f}, t)) \in [\![\Delta.P_\upsilon(F(\overline{\sigma}))]\!].$$

- $\text{zipop}_m$ $\dfrac{\begin{array}{c}\Delta \vdash F : m \quad \Delta \vdash \sigma_{i \in m}, \tau_{i \in m}, \theta_{i \in m}, \upsilon \\ \Delta; \Gamma \vdash f_i : \sigma_i \to \tau_i \to P_\upsilon(\theta_i) \quad \Delta; \Gamma \vdash s : F(\overline{\sigma}) \quad \Delta; \Gamma \vdash t : F(\overline{\tau})\end{array}}{\Delta; \Gamma \vdash \text{zipop}_m(\overline{f}, s, t) : P_\upsilon(F(\overline{\theta}))}$

  By inductive hypothesis $S(s) \in H(\overline{X})$, $S(t) \in H(\overline{Y})$ and $S(f_i) \in X_i \to Y_i \to M_W(Z_i)$ (for $i \in m$), where $H = [\![\Delta.F]\!]\rho$, $W = [\![\Delta.\upsilon]\!]\rho$, $X_i = [\![\Delta.\sigma_i]\!]\rho$, $Y_i = [\![\Delta.\tau_i]\!]\rho$ and $Z_i = [\![\Delta.\theta_i]\!]\rho$.

  By Lemma B.12
  $$\text{zipop}_m((S(f_i))_{i \in m}, S(s), S(t)) \in M_W(H(\overline{Z}))$$
  thus
  $$S(\text{zipop}_m(\overline{f}, s, t)) \in [\![\Delta.P_\upsilon(F(\overline{\theta}))]\!].$$

∎

**Theorem B.14 (SN)** *If $\Delta; \Gamma \vdash t : \tau$ is derivable in $\text{FF}_{shape}$, then $t \in \text{SN}$.*

**Proof.** We consider the family of parsing monads $P_S$ (see section 5.3.2.1). First we have to provide a family of functions $M_W : \text{Sat} \to \text{Sat}$ which satisfies Axiom B.8.

We define $M_W : \text{Sat} \to \text{Sat}$ as $M_W X = W \to [\![+]\!]([\![x]\!](X, W), [\![1]\!]())$. The monotonicity of $M_W$ derives from the monotonicity of the functions $[\![+]\!]$, $[\![x]\!]$, $[\![1]\!]$ and from the monotonicity of $\to$ with respect the covariant argument. Now we prove that for any $W \in \text{Sat}$ and for any $X, Y \in \text{Sat}$:

- *val* $= \lambda x.\lambda s.\mathrm{in}_0\langle x, s\rangle \in X \to M_W(X)$: by lemma B.12 it is sufficient to prove that for all $t_1 \in X$ one has $[t_1/x]\lambda s.\mathrm{in}_0\langle x, s\rangle \equiv \lambda s.\mathrm{in}_0\langle t_1, s\rangle \in M_W(X)$, that is for all $t_2 \in W$, $\mathrm{in}_0\langle t_1, t_2\rangle \in [\![+]\!]([\![x]\!](W, X), [\![1]\!]())$. Trivial.

- *let* $= \lambda f.\lambda t.\lambda s.\mathrm{case}\ (t\ \ s)\ \mathrm{of}\ \mathrm{in}_0\ \langle x, u\rangle \Rightarrow (f\ \ x\ \ u)\ \mathrm{in}_1\ y \Rightarrow \mathrm{fail}\ \ \in (X \to M_W(Y)) \to M_W(X) \to M_W(Y)$: by lemma B.12 it is sufficient to prove that for all $t_1 \in X \to M_W(Y)$, for all $t_2 \in M_W(X)$, for all $t_3 \in W$ one has case $(t_2\ t_3)$ of $\mathrm{in}_0\ u \Rightarrow (t_1\ (\pi_0\ u)\ (\pi_1\ u))\ \mathrm{in}_1\ y \Rightarrow \mathrm{fail}\ \ \in [\![+]\!]([\![x]\!](W, Y), [\![1]\!]())$. This term can reduce either to $t_1\ (\pi_0\ u)\ (\pi_1\ u)$ or to fail. Both terms belong to $[\![+]\!]([\![x]\!](W, Y), [\![1]\!]())$.

- *fail* $= \lambda s.\mathrm{fail} \in M_W(X)$: it is sufficient to show that fail $\in [\![+]\!]([\![x]\!](W, Y), [\![1]\!]())$. Trivial.

It is straightforward to prove that that for any $\rho \in [\![\Delta]\!]$ $[\![\Delta.P_\upsilon(\tau)]\!]\rho = M_{[\![\Delta.\upsilon]\!]\rho}([\![\Delta.\tau]\!]\rho)$.

So now we can apply Proposition B.13 and from $\Delta; \Gamma \vdash t : \tau$ we have $S(t) \in [\![\Delta.\tau]\!]\rho$, whenever $\rho \in [\![\Delta]\!]$ and $S \in [\![\Delta.\Gamma]\!]\rho$. Every $X \in \mathrm{Sat}$ contains the term variables because of saturation: in fact $x \in \mathrm{SN}$ and $\mathrm{V}(x)$ is empty. Therefore, by taking $S$ to be the identity substitution, we have that $t \in [\![\Delta.\tau]\!]\rho$ and so $t \in \mathrm{SN}$. ∎

# Bibliography


[Acz78] P. Aczel. A general Church-Rosser theorem. Technical report, Univ. of Manchester, 1978.

[AFH94] S. Aditya, C. Flood, and J. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *ACM Conference on Lisp and Functional Programming*, pages 12–23, June 1994.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[Bac88] R. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS 8810, Groningen University, 1988.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984. revised edition.

[Bar92] H.P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science.* Oxford Univ. Press, 1992.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L:structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 3(7):225–255, May 1995.

[Bel98] G. Bellè. A metalanguage for concurrency and data distribution. Draft, August 1998.

[Ben67] J. Benabou. *Introduction to bicategories*, volume 47 of *Lecture Notes in Mathematics.* Springer, 1967.

[Ben95] N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Springer-Verlag, editor, *Proceedings of Computer Science Logic '94*, volume 933 of *LNCS*, 1995.

[BFSS90] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[BGHP98] A. Barber, P. Gardner, M. Hasegawa, and G. Plotkin. From action calculi to linear logic. In *CSL'97*, volume 1414 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[Bir88] R.S. Bird. Lectures on constructive functional programming. Technical Monograph PRG 69, Oxford University, 1988.

[BJM96] G. Bellè, B. Jay, and E. Moggi. Functorial ML. In *PLILP'96*, volume 1140 of *LNCS*. Springer Verlag, 1996.

[Bjö97] S.L. Björk. Parsers, Pretty Printers and PolyP. Master's thesis, Göteborg University, 1997.

[Ble95] G.E. Blelloch. NESL: a Nested Data-Parallel Language (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, September 1995.

[BM97] G. Bellè and E. Moggi. Typed Intermediate Languages for Shape Analysis. In Philippe de Groote and J.Roger Hindley, editors, *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 11–29. Springer-Verlag, 1997.

[BNTW95] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149:3–48, 1995.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1990.

[Car78] J. Cartmell. *Generalized algebraic theories and contextual categories*. PhD thesis, University of Oxford, 1978.

[CCL91] M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers*, chapter Crystal: Theory and Pragmatics of Generating Efficient Parallel Code, pages 255–308. ACM Press Frontier Series, 1991.

[CF92] J.R.B. Cockett and T. Fukushima. About **charity**. Technical Report 92/480/18, University of Calgary, 1992.

[CLW93] A. Carboni, S. Lack, and R.F.C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84:145–158, 1993.

[Coc90] J. Robin Cockett. List-arithmetic distributive categories: Locoi. *Journal of Pure and Applied Algebra*, 66:1–29, 1990.

[CPM89] T. Coquand and C. Paulin-Mohring. Inductively defined types. volume 389 of *LNCS*. Springer-Verlag, 1989.

[CS95] J.R.B. Cockett and D. Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theoretical Computer Science*, 139(1–2):69–113, 1995.

[Dav96] R. Davies. A temporal-logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Elenventh Annual Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *Parallel Architectures & Languages Europe*. Springer-Verlag, June 93.

[DGTY95] J. Darlington, Y. Guo, H.W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In Seif Haridi, Khayri Ali, and Peter Magnussin, editors, *Euro-Par'95 Parallel Processing*, pages 55–69. Springer-Verlag, August 1995.

[Fok94] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in TCS*. CUP, 1989.

[Gog94] H. Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[Hag87a] T. Hagino. *A categorical programming language*. PhD thesis, University of Edinburgh, 1987.

[Hag87b] T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A Poigné, and D. E. Rydeheard, editors, *Category and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer, September 1987.

[HM95] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[HM96] R. Harper and J. Mitchell. ML and beyond. ACM Workshop on Strategic Directions in Computing Research, 1996.

[Hof95] M. Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, LFCS Edinburgh, 1995.

[Hof97] M. Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation.* Cambrigde University Press, 1997.

[HS93] J. Hook and T. Sheard. A semantics of compile-time reflection. Technical Report 93-019, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1993.

[Jac91] B. Jacobs. *Categorical Type Theory.* PhD thesis, University of Nijmegen, 1991.

[Jac93] B. Jacobs. Comprehension categories and the semantics of type theory. *Theoretical Computer Science*, 107:169–207, 1993.

[Jan97] P. Jansson. Functional polytypic programming — use and implementation. Technical report, Chalmers University of Technology, Gothenburg, Sweden, 1997. Licentiate thesis. Available from `http://www.cs.chalmers.se/~patrikj/lic/`.

[Jay93] C.B. Jay. Matrices, Monads and the Fast Fourier Transform. Technical Report UTS-SOCS-93.13, University of Technology, Sydney, 1993.

[Jay95a] C.B. Jay. Polynomial polymorphism. In R. Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference: Glenelg, South Australia 1–3 February, 1995*, volume 17, pages 237–243. A.C.S. Communications, 1995.

[Jay95b] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[Jay95c] C.B. Jay. Shape analysis for parallel computing. In J. Darlington, editor, *Parallel Computing Workshop '95 at Fujitsu Parallel Computing Centre, Imperial College*, pages 287–298. Imperial College / Fujitsu Parallel Computing Research Centre, 1995.

[Jay96] C.B. Jay. Shape in computing. *ACM Computing Surveys*, 28(2):355–357, 1996.

[Jay97] C.B. Jay. Covariant types. *Theoretical Computer Science*, 185:237–258, 1997.

[JBM98] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6), 1998.

[JC94] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, Lecture Notes in Computer Science, pages 302–316. Springer Verlag, 1994.

[JCSS97] C.B. Jay, M.I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 650–661. Springer, August 1997.

[Jeu95] J. Jeuring. Polytypic pattern matching. In *Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, pages 68–114. Springer-Verlag, 1996. LNCS 1129.

[JJ97] P. Jansson and J. Jeuring. PolyP – a polytypic programming language extension. In *Conference Record of POPL'97: the 24th SCM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, 1997.

[Jon95] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *J. of Functional Programming*, 5(1), 1995.

[Joy81] A. Joyal. Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42:1–82, 1981.

[JS97] C.B. Jay and M. Sekanina. Shape checking of array programs. In *Computing: the Australasian Theory Seminar, Proceedings, 1997*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.

[JS98] C.B. Jay and P.A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98*, volume 1381 of *LNCS*, pages 139–53. Springer Verlag, 1998.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980. Tracts 129.

[KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order$\lambda$-calculus. *Information and Computation*, 98(2):228–257, jun 1992.

[Lau91] J. Launchbury. *Projection factorisations in partial evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.

[Ler92] X. Leroy. Unboxed Objects and Polymorphic Typing. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, New-York, 1992.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[LT97] K. Lellahi and V. Tannen. A Calculus for Collections and Aggregates. In Moggi and Rosolini, editors, *Category Theory and Computer Science'97*, volume 1290 of *LNCS*, pages 261–280. Springer-Verlag, 1997.

[Mac71] S. MacLane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[MBJ98] E. Moggi, G. Bellè, and C.B. Jay. Monads, Shapely Functors and Traversals. Technical Report DISI-TR-98-06, DISI - University of Genova, 1998. http://www.disi.unige.it/person/BelleG/publications.html.

[Mee89] L. Meertens. Constructing a calculus of programs. In J. L. A. van de Snepscheut, editor, *Proceedings 1st Intl. Conf. on Maths. of Program Construction, MPC'89, Groningen, The Netherlands, 26–30 June 1989*, volume 375 of *Lecture Notes in Computer Science*, pages 66–90. Springer-Verlag, Berlin, 1989.

[Mee96] L. Meertens. Calculate Polytypically! In Kuchen and Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1996. Invited talk.

[Men91] N.P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51, 1991.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Procceding of the*

*5th ACM Conference on Functional Programming and Compter Architecture*, volume 523 of *LNCS*, pages 124–44. Springer Verlag, 1991.

[MH95] E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In *Procedings 7th International Conference on Functional Programming and Computer Architecture, San Diego, California, June 1995.* ACM Press, 1995.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978.

[Mil96] R. Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[MJ95] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Proceedings of the First International Springschool on Advanced Functional Programming Techniques*, Lecture Notes in Computer Science 925, pages 228–266. Springer-Verlag, 1995.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[Mog91a] E. Moggi. A category-theoretic account of program modules. *Math. Struct. in Computer Science*, 1:103–139, 1991.

[Mog91b] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[Mog95] E. Moggi. A semantics for evaluation logic. *Fundamenta Informaticae*, 22:117–152, 1995.

[Mog97] E. Moggi. A categorical account of two-level languages. In *13-th Conference on Mathematical Foundations of Programming Semantics*, volume 6 of *Electronic Notes in Theoretical Computer Science*, 1997.

[NN92] F. Nielson and H.R. Nielson. *Two-Level Functional Languages.* Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[NN96] F. Nielson and H. R. Nielson. Multi-level lambda-calculi: an algebraic description. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 338–354. Springer-Verlag, 1996.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's type theory:an introduction.* Number 7 in International series of monographs on computer science. Oxford University Press, New York, 1990.

[Pit95] A.M. Pitts. Categorical logic. Technical Report 367, University of Cambridge Computer Laboratory, May 1995.

[Pit96] A.M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

[PJ91] S.L. Peyton Jones. Unboxed values as first-class citizens. In *Functional Programming and Computer Architecture*, volume 523 of *LNCS*. Springer Verlag, 1991.

[RP90] J.C. Reynolds and G.D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 1990.

[Sek98] Milan Sekanina. *Shape Analysis*. PhD thesis, University of Technology, Sydney, 1998.

[She93] T. Sheard. Type parametric programming. Technical Report 93-018, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1993.

[SHM97] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about bsp. *Scientific Programming*, 6:249–274, 1997.

[Ski94] D. Skillicorn. *Foundations of Parallel Programming*. Number 6 in Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.

[ST94] D. Suciu and V. Tannen. Efficient compilation of high-level data parallel algorithms. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, June 1994.

[Suc95] D. Suciu. *Parallel Programming Languages for Collections*. PhD thesis, University of Pennsylvania, 1995.

[Tay86] P. Taylor. *Recursive Domains, Indexed Category Theory and Polymorphism*. PhD thesis, University of Cambridge, 1986.

[Tay99] P. Taylor. *Practical Foundations of Mathematics*. Number 59. Cambridge University Press, 1999.

[TJ92] J.P. Talpin and P. Jouvelot. The type and effect discipline. In *Seventh Annual IEEE symposium on Logic in Compute Science*. IEEE Computer Society Press, June 22-25 1992.

[To95] H.W. To. *Optimising the Parallel Behaviour of Combinations of Program Components*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1995.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988. Available as CST-52-88.

[Tol94] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on Lisp and Functional Programming*, pages 1–11, June 1994.

[TT94] M. Tofte and J.P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings from the 21st annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[Wad93] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 1993. Special issue of selected papers from 6th Conference on Lisp and Functional Programming, 1992.

[WFK97] Q. Wu, A.J. Field, and P.H.J. Kelly. M-tree: a parallel abstract data type for block-irregular adaptive applications. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 638–649. Springer, August 1997.

[WT91] D. A. Watt and P. Trinder. Towards a theory of bulk types. Technical Report FIDE TR, June 1991.

# Index


Array for families of types, 85

Categorical logical relations, 75
Category
- arrow , 158
- display map, 39
- extensive, 157
- indexed, 162
- Kleisli, 155
- lextensive, 157
- relative slice, 50
- slice, 157
- wedge, 159
- with attribute, 37

Class of display maps, 29, 39

Display map category, 39
- cartesian product, 52
- strong disjoint union, 52
- exists quantification, 52
- forall quantification, 52
- finite cardinals, 54
- product, 51
- strong natural number object, 52

Fibration, 160
- reindexing, 160

Fibre, 160
Functor
- relative shapely, 29
- shapely, 20

Grothendieck construction, 162

Independence, 50

Kleisli triple, 154

List object, 155
- for families of types, 84

Locos, 157

Monad, 154
- error- for families of types, 84
- error- for simple transl., 80
- error- plus for families of types, 84
- error- plus for simple transl., 81
- fibred computational, 62
- fibred shapely, 63
- mono preserving, 72
- relative shapely, 30
- shapely, 21
- strong, 154

Natural number object, 156

Section, 159
Set indexed families, 158
Simple, 50
Strength, 153

Traversal, 24
Transformation
- cartesian, 21
- cartesian w.r.t. class of monos, 72
- relative cartesian, 30
- shapely, 21
- strong, 20