The Productivity of Polymorphic Stream Equations *and* *The Composition of Circular Traversals*

Florent Balestrieri

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

October 2015

Abstract

This thesis has two independent parts. The first is a theoretical study of productivity for very restricted stream programs. In the second part we define a programming abstraction over a recursive pattern for defining circular traversals modularly.

Productivity is in general undecidable. By restricting ourselves to mutually recursive polymorphic stream equations having only three basic operations, namely `head`, `tail`, and `cons`, we aim to prove interesting properties about productivity. Still undecidable for this restricted class of programs, productivity of polymorphic stream functions is equivalent to the totality of their indexing function, which characterise their behaviour in terms of operations on indices. We prove that our equations generate all possible polymorphic stream functions, and therefore their indexing functions are all the computable functions, whose totality problem is indeed undecidable. We then further restrict our language by reducing the numbers of equations and parameters, but despite those constraints the equations retain their expressiveness. In the end we establish that even two non-mutually recursive equations on unary stream functions are undecidable with complexity Π^0_2. However, the productivity of a single unary equation is decidable.

Circular traversals have been used in the eighties as an optimisation to combine multiple traversals in a single traversal. In particular they provide more opportunities for applying deforestation techniques since it is the case that an intermediate datastructure can only be eliminated if it is consumed only once. Another use of circular programs is in the implementation of attribute grammars in lazy functional languages. There is a systematic transformation to define a circular traversal equivalent to multiple traversals. Programming with this technique is not modular since the individual traversals are merged together. Some tools exist to transform programs automatically and attribute grammars have been suggested as a way to describe the circular traversals modularly. Going to the root of the problem, we identify a recursive pattern that allows us to define circular programs modularly in a functional style. We give two successive implementations, the first one is based on algebras and has limited scope: not all circular traversals can be defined this way. We show that the recursive scheme underlying attribute grammars computation rules is essential to combine circular programs. We implement a generic recursive operation on a novel attribute grammar abstraction, using containers as a parametric generic representation of recursive datatypes. The abstraction makes attribute grammars first-class objects. Such a strongly typed implementation is novel and make it possible to implement a high level embedded language for defining attribute grammars, with many interesting new features promoting modularity.

Acknowledgements

My thanks to Venanzio Capretta who supervised my work, thank you for your guidance, your patience and support. Thank you Graham Hutton for your encouragements. My thanks to the Functional Programming group at the University of Nottingham who offered a nurturing environment and encouraged the exchange of ideas. Professors Thorsten Altenkirch, Venanzio Capretta, Graham Hutton, Henrik Nilsson. Doctors or PhD students Matteo Acerbi, Guillaume Allais, Patrick Bahr, Joey Capper, Paolo Capriotti, Nils Anders Danielsson, Laurence Day, Gabe Dijkstra, Bas van Gijzel, George Giorgidze, Alexander Green, Jennifer Hackett, Mauro Jaskelioff, Ambrus Kaposi, Nicolai Kraus, Iain Lane, Peter Morris, Darin Morrison, Li Nuo, Ivan Perez Dominguez, Juan Carlos Saenz Carrasco, Christian Sattler, Neil Sculthorpe.

Table of Contents

1 Introduction **1**
1.1 Coinductive Types and Functional Programming 1
1.1.1 Coinductive types in Type theory 2
1.1.2 Coinduction and Infinity 3
1.1.3 Productivity 3
1.1.4 Approach in the Thesis 4
1.2 Circular Traversals 5
1.3 Structure of the Thesis 5
1.4 Main Contributions of the Thesis 7
1.5 Related Work 8
1.5.1 On The Productivity of Stream Equations 8
1.5.2 On Circular Traversals and Attribute Grammars 9

I Productivity of Pure Stream Equations **11**

2 Coinduction and Productivity **13**
2.1 Coinductive types and Coalgebras 13
2.1.1 Terminal Coalgebra 13
2.1.2 Fixed-Points 14
2.1.3 Bisimulation and Bisimilarity 15
2.2 Productivity 18
2.2.1 Productivity and Denotational Semantics 18
2.2.2 Productivity of Polymorphic Definitions 21
2.2.3 Productivity and Strict Languages 21
2.2.4 Strong Languages, Inductive Types 21
2.2.5 Syntactic Criteria 22
2.2.6 Non Productive Definitions in Type Theory 22

3 Pure Stream Equations **25**
3.1 Streams and Polymorphic Functions 25
3.1.1 Polymorphism 26
3.1.2 Polymorphic Streams Functions 26
3.1.3 Indexing Functions 27
3.1.4 Coalgebraic Stream Equations 30
3.1.5 The Indexing Functions of Coalgebraic Equations 31
3.2 Pure Stream Equations 31
3.2.1 Introduction 32
3.2.2 The Syntax of Pure Stream Equations 33
3.2.3 Semantics of Pure Stream Equations 34
3.2.4 Interleaving and Projection 36

3.2.5 Zip-Proj Equations 38

4 PSES Productivity is Π_2^0-Complete 41
4.1 Terminology: Problems, Computability and Reductions 41
4.2 A Generalisation of the Collatz Problem 44
4.2.1 The Collatz Problem 44
4.2.2 A Generalisation 45
4.2.3 Properties of Collatz Functions 45
4.3 Generalised Collatz is Reduced to PSES Productivity 46
4.3.1 Comparison with Other Proofs of Undecidability 48
4.4 The Generalised Collatz Problem is Π_2^0-Complete 49
4.4.1 Kurtz and Simon's Generalisation 50
4.4.2 Reduction from KSP to GCP 51
4.4.3 Discussion 53

5 Expressivity of PSES 55
5.1 PSE Definability 55
5.1.1 Counter Machines 56
5.1.2 Construction of PSE Definability 57
5.1.3 Proof 57
5.2 Unary Definability 59
5.2.1 Properties of some Indexing Functions 60
5.2.2 Collatz Functions and If-Programs 61
5.2.3 Iteration-Programs and Their Encoding 64
5.2.4 Proof of Unary Decidability 69
5.3 Further Results 70

II Circular Traversals Compositionally 71

6 Circular Traversals Using Algebras 73
6.1 Introduction 73
6.1.1 Circular Traversals in General 74
6.1.2 Issues With the Transformation 75
6.2 Abstract Programming Interface for Computations over a Data Structure 77
6.2.1 The Environment Arrow 79
6.2.2 A primitive to define traversals 80
6.3 Circular Implementation 82
6.3.1 Recursive Pattern for Circular Traversals 82
6.3.2 ArrowCata Instance 84
6.3.3 Proving the Homomorphism Properties 86
6.3.4 Kleisli Arrow for an Indexed Monad 87
6.4 Example: List of Deviations 88

7 Circular Traversals Using Attribute Grammars 91
7.1 Palindrome 91
7.1.1 The Problem 91

7.1.2 Bird's Solution 92
7.1.3 Fixing Bird's Solution 93
7.1.4 Palindrome with ArrowCata 95
7.1.5 Pairing Higher-Order Algebras 96
7.1.6 A New Traversal Primitive 96
7.2 Attribute Grammars 99
7.2.1 Attribute Grammar Implementations 99
7.2.2 Generic AG Rules 100
7.2.3 Attribute Grammar Systems 102
7.3 Containers and W Types 102
7.3.1 Type-Theoretical Implementation 103
7.3.2 Haskell Implementation 103
7.4 Circular Programs as Compositions of AGs 105
7.4.1 Generic View as a W Type 106
7.4.2 Generic Attribute Grammar Traversals 107
7.4.3 ArrowAG 110
7.4.4 Multiple Traversals 110
7.4.5 Circular Implementation 110
7.5 Examples 112
7.5.1 Palindrome 112
7.5.2 Removing the Redundant Tests 113

8 Perspectives 115
8.1 Performance 115
8.1.1 Experimental Results 116
8.2 Optimisation and Recursion Schemes 117
8.3 Generic Programming with Containers 118
8.3.1 Related Works 118
8.3.2 Generic Functions and Generic Types 119
8.3.3 Design Choices for the Payload Functor 119
8.3.4 Mutually Recursive Data Structures 120
8.3.5 Modular Datatypes and Paths 121
8.4 Embedding a Strongly Typed AG System 121
8.4.1 Example 123
8.4.2 Full Listing 130
8.5 Related Works 132
8.5.1 Circular Traversals without Attribute Grammars 132
8.5.2 Circular Traversals with Attribute Grammars 132
8.5.3 Optimisation of Circular Programs 134

9 Conclusion 135
9.1 Productivity of Pure Stream Equations 135
9.2 Circular Traversals Compositionally 136

References 138

Chapter 1.
Introduction

Coinductive types are essential to model infinite computations. Such computations must be productive, which means they always produce more data if we need them to. In the first part of the thesis, we study the productivity of polymorphic stream equations: a simple case of coinductive programs. In the second part, we study the composition of functions that traverse the same datastructures. Such traversals can be in fact combined in a circular definition, provided the evaluation is lazy.

Overview In this chapter, we introduce the notions of coinduction in §1.1 and productivity in §1.1.3, providing a motivation for the first part of the thesis. In §1.2 we motive the second part on circular traversals. §1.3 gives a road-map of the thesis, summarising each chapters. §1.4 states the main contributions of the thesis. §1.5 is a review of related work in the field.

1.1. Coinductive Types and Functional Programming

In an influential article underlining the merits of functional programming [Hug89], John Hughes identified many types of *glue* that one can use to combine simple programs into complex ones. One of these glues is *laziness*. Lazy datastructures are constructed on demand and allow us to structure our programs into producers and consumers, thus giving more opportunities of code reuse.

Lazy datastructures are ubiquitous in lazy languages such as Haskell (the language of all our programming examples); but they are also common in strict languages (with eager evaluation strategies), like ML. The usefulness of lazy datastructures and lazy lists in particular was wildly recognised and adapted to other programming paradigms [Sch00, NW00, Gru06, SBMG07]. Existing standards of popular programming languages have been extended to cater for lazy lists, among others Perl, Ruby, Python, SWI-Prolog.

One interesting consequence of laziness is that it is possible to define infinite datastructures, meaning that a producer can produce an unbounded amount of data, yet only what is needed by its consumers would ever be computed.

The most common lazy datastructure is the lazy list. For instance, in Haskell the list of all integers is defined with:

```
ints :: [Integer]
ints = from 0
from x = x : from (x+1)
```

from is an example of a *corecursive* definition. The corecursive calls take an ever-growing integer argument, and the list computed by from x is infinite. It looks as if the program doesn't terminate, but in Haskell, the list constructor : is lazy, it doesn't evaluate its second argument (the corecursive call) to return a result. The computation of the second argument is postponed until it is actually needed, hence the name *lazy* evaluation, or *call by need.* We say the corecursive call from (x+1) is guarded by a constructor, *i.e.* it is a direct child of the constructor (:), and is a sufficient condition of productivity: the property that the infinite datastructure is well defined.

We can compute finite prefixes of from x:

```
take :: Integer → [a] → [a]
take n xs | n ≤ 0     = []
          | null xs   = []
          | otherwise = head xs : take (n-1) (tail xs)
```

The function take is structurally recursive on its inductive argument n and always terminates, even when the second argument xs is an infinite list.

Formally, lazy datastructures are elements of *coinductive types*, which are dual to inductive types. In Haskell, the language makes no distinction between inductive and coinductive types, we must make that distinction ourselves in the way we think about our programs. I believe that the theoretical underpinnings of inductive and coinductive types can guide us in our programming, and help us design useful programming abstractions.

1.1.1. Coinductive types in Type theory

Functional programming and formal mathematics converge in type theory, a functional language where types are values, and where types and mathematical propositions are identified [ML84]. Terms are the proofs of their types viewed as propositions. Arbitrarily complex types can be designed to capture the semantics of a program, and a program with such a type comes with the proof that it behaves according to the semantics. Furthermore, the type-checker actually validates the proof. Such certified programs are very valuable for critical tasks where programming errors could cost human lives, cause physical damage, or the loss of financial assets.

Coinductive types play a major role in this context. Type theories only allow restricted recursion schemes that are known to terminate. However, many practical programs are not terminating by nature: control systems, operating systems, network servers, etc. Coinductive types allow us to formalise their behaviour in type theory. The recursive scheme used to define coinductive values ensures the programs are *productive* which means all possible observations on them terminate. Capretta and Bove gave an embedding of general recursion using a coinductive type for partial computations [Cap05]; so the behaviour of general recursive programs can be formalised and their correctness proven.

Coinductive types have received less attention than their dual inductive types [GJ98], however, the importance of coinductive types shouldn't be ignored given that they fully explain lazy datastructures, and allow us to formalise infinite definitions in type-theory.

1.1.2. Coinduction and Infinity

The theory of coinduction gives the theoretical foundation to reason about infinite datastructures. When a program expression has a coinductive type, the value it denotes may be infinite, like the sequence of all prime numbers, the Fibonacci sequence, Pascal's triangle, continued fractions, etc. Furthermore, we can identify the expression with its infinite value.

This makes lazy functional programming a very high level paradigm: we can define and manipulate infinite objects as if they existed as mathematical entities, outside of time in their infinite form, even though in practice only a finite part can ever be actually constructed in the finite memory of a computer. In addition we needn't be concerned about the actual steps it takes to progressively compute this concrete portion of the infinite value. In operational terms, the mechanism involved is *laziness*. It is supported by default in non-strict languages like Haskell [Mar10], or with primitives *delay* and *force* in strict languages like ML.

This notion of computable infinity is also what intuitionist mathematicians use [Dum77, §3.1 p. 40]:

> All infinity is potential infinity: there is no completed infinite. [...] It means, simply, that to grasp an infinite structure is to grasp the process which generates it, that to refer to such a structure is to refer to that process, and that to recognise the structure as being infinite is to recognise that the process will not terminate.

The process is given by the program expression. The infinite structure is the abstract idea of the data that we would construct in memory by executing the program forever in a Utopian computer with infinite memory and contemplating this data at the end of time.

1.1.3. Productivity

In the presence of general recursion, we may write programs with semantically undefined values, operationally blocked in a loop without returning a result. For instance:

```
diverge :: Integer → Integer
diverge x = x + diverge (x + 1)
```

Compare this definition with `from` given above: the main difference is that the recursive call here appears under a strict operation + that evaluates both its arguments, while the corecursive call of `from` was under a lazy constructor which delayed its evaluation. Consequently the computation here diverges: there is an infinite number of recursive calls to evaluate before a result is returned.

In denotational semantics, the result here is either an concrete integer or an undefined value, written $\perp$, the semantics associated to a diverging computation, we call such a domain *flat*. By contrast, when a result is structured, and in particular coinductive, the domain contains all the partial values, approximating a fully defined value. We can thus distinguish between an undefined list and a list of undefined elements. In broad terms, we say that a value is productive if it is fully defined. Whereas the stream `ints` above was productive, the two following streams are not productive:

```
partial_stream = 1 : 2 : 3 : undefined
stream_with_undefined_elements = 1 : undefined : 3 : from 4
undefined = undefined
```

Since we deal with potentially infinite values, *fully defined* must be defined with caution. It means we can compute arbitrarily large finite approximations.

The formal definition of productivity, given in §2.2 is a bit tricky because coinductive datastructures may contain other coinductive data, consequently the definition of productivity is coinductive!

In this thesis we will study in particular the productivity of stream functions. For a function to be productive, it must preserve productivity. In this particular case, they must map productive streams to productive streams. A productive stream is a fully determined stream: all of its elements converge to a definite value. Productivity is a semantic property: it characterises the behaviour of programs rather than their concrete definition. It is a property of good behaviour, ensuring that a piece of program, when combined with other program fragments cannot be blamed for non-termination. This is illustrated with productive functions: by preserving productivity, they preserve the good behaviour; if we respect their contract and provide a well behaving (productive) input they will ensure us a well behaving output. Similarly, productive data structures (coinductively or inductively defined) only hold productive data.

1.1.4. Approach in the Thesis

Although the problem of deciding the productivity of a general corecursive equation is undecidable, there might be a big enough set of equations for which a decision procedure could be designed. We tried to find such a set, with enough generality to be of some use but with enough restrictions that we would hope for the productivity to be decidable. It seemed reasonable to only consider one simple coinductive type and one simple form of definitions. Amongst the simplest yet useful coinductive types is the stream, and amongst the simplest equations on streams are *pure stream equations* (§3.2) which define functions parametric over the stream elements. Such functions do not seem to do very complex computations: being polymorphic, they only operate on the structure of streams, infinite sequences of elements, without inspection of the contained data: they cannot test the elements, and just shuffle them around, possibly discarding or duplicating some of them. Pure stream equations define a natural yet restrictive class of polymorphic stream functions as a system of equations using only stream constructors, destructors and recursive calls.

1.2. Circular Traversals

Circular traversals are the focus of the second part of the thesis. Lazy functional languages allows us to make circular definitions to define some data. Some circular definitions can be used to define coinductive values, for instance the Fibonacci sequence:

```
(fib, tail_fib) = (0 : tail_fib, 1 : fib <+> tail_fib)
  where (x:xs) <+> (y:ys) = x+y : xs <+> ys
```

In Part II, we study a particular type of circular definitions that arises when a computation over a tree depends on its own result. This case usually stems from the combinations of multiple non-circular traversals over the same tree. In fact, the circular traversals were originally designed after transforming a multi-traversal computation [Bir84].

As an example, let us compute the number of occurrences of the maximum element of a list. We must traverse the list once to compute its maximum element, and then another time to compute the number of elements equal to that first result.

```
count_maximum xs = count_equal (maximum xs) xs
maximum [] = 0                                  -- only considering positive integers
maximum (x : xs) = max x (maximum xs)
count_equal y [] = 0
count_equal y (x : xs) = eq x y + count_equal y xs
eq x y = if x ≡ y then 1 else 0
```

The two traversals can in fact be combined in a single traversal computing their results simulataneously:

```
maximum_and_count_equal y [] = (0, 0)
maximum_and_count_equal y (x:xs) = (max x m, eq x y + c)
  where (m,c) = maximum_and_count_equal y xs
```

Now we can make a circular definition that binds the first component of the result (corresponding to the maximum) to the argument of the traversal:

```
count_maximum_circ xs = c
  where (m,c) = maximum_and_count_equal m xs
```

This transformation is not modular: by combining each independent traversal we do not promote code reuse. The problem studied in Part II is to find a suitable representation of circular traversals together with combinators allowing them to be composed.

1.3. Structure of the Thesis

Chapter 2 defines coinductive types, coalgebras, bisimulations, and most importantly, productivity which is the central theme of the thesis.

Chapter 3 gives the background notions and common definitions necessary for the rest of the thesis: we define pure stream equations 3.2, polymorphism, and indexing functions.

Chapter 4 gives a proof that the productivity of pure stream equation systems (PSES) is Π^0_2 complete. PSES define polymorphic functions and productivity is equivalent to totality of the indexing function (Corollary 3.2.8), so the problem is Π^0_2. But is it Π^0_2-hard? After all, the equations might not capture all polymorphic functions and the functions that they capture might not be so complicated that the problem is Π^0_2-hard. In fact it is: we found a set of indexing functions whose totality problem is undecidable and fully captured by PSES, i.e. there exist a PSES specifying the corresponding polymorphic function associated with the indexing function. Such a set is given by a generalisation of the Collatz problem.

Pure stream equation systems (PSES) define polymorphic stream functions. In Chapter 5 we asked the question: can they define *all* polymorphic stream functions? Since polymorphic functions are completely characterised by their indexing function (see §3.1.2) which are partial (computable) functions from $\mathbb{N}$ to $\mathbb{N}$, we showed that the indexing functions of PSES cover all the computable functions; meaning that PSES form a Turing complete computation model. In §5.1 we construct a PSES from a counter machines implementing the corresponding indexing function. In §5.2 we strengthen the previous result by giving a more intricate encoding using a strict subset of PSES consisting of only unary equations.

In Part II we abstracted a corecursive pattern that occurs in the technique of *circular programming* [Bir84]. This programming pattern relies on lazy evaluation to fuse many traversals of data structures into a single traversal. However, by following this traditional approach we lose the advantages of modularity: succinctness, clarity, reusability, unit testing and easier proofs of correctness. We show in this chapter that this is not necessarily so: we present a generic library of combinators for modular circular programs based on the `Arrow` abstraction.

We give two implementations of that abstraction, one as multiple traversals, one as single circular traversals. Initially, in Chapter 6 we use algebras as the representation of primitive traversals, but we realise that the product of higher-order algebras doesn't combine their traversals.

Seeking for a finer representation, in Chapter 7 we found a new first class implementation of attribute grammars, using containers which for the first time have been implemented in Haskell. Containers promise new possibilities for generic programming: they give a semantic view of types rather than the usual syntactic construction using sums and products and generic programs based on them enjoy parametricity properties.

In Chapter 8 we discuss our research on circular traversals in a broader context of research. We discuss the performance of circular programs, and optimisation in general; we look at other aspects of containers; we present our AG DSL designed on top of the container based AG abstraction, and we discuss the related works, specifically regarding circular traversals and attribute grammars.

1.4. Main Contributions of the Thesis

Productivity of Pure Stream Equations

- Noting that stream equations define computable polymorphic functions, we reduce the productivity problem to the *totality* of a computable function characteristic of the polymorphic property and that we call *indexing function* (Definition 3.1.6): for a stream equation to be productive, its indexing function must be total.

- In §5.1 we show with a more complex encoding that *all* polymorphic functions are definable by stream equations: this class of programs consists of exactly the computable polymorphic stream functions.

- In §5.2, we strengthen the previous result and show that stream programs consisting of unary equations without mutual recursion can define all polymorphic functions, yielding an elegant model of Turing-completeness.

Circular Traversals Compositionally

- In Chapter 6 we define a novel programming abstraction for combining multiple circularly dependent catamorphisms into a single catamorphism in a lazy functional language. The circular technique was well-known, but it was based on a program transformation which impeded modular programming. First class attribute grammars [dMBS00, VSS09] was an existing solution for modularly defining the circular traversals, but it suffered either from a lack of type-safety [dMBS00] or for a big notational overhead and other inconveniences [VSS09]. On the other hand, our solution stays within the functional paradigm. Using the abstract interface allows the programmer to execute the same program using multiple traversals or circularly in one traversal. Using multiple traversals corresponds to the multi-pass compilation which is usually determined by dependency analysis.

- In§7.3.2 we show that dependent type families can be viewed as the display maps of GADTs. This is a new insight with interesting applications, among which the encoding of containers in Haskell.

- In §7.3.2, we give an encoding of containers and indexed containers in Haskell, using GADTs, offering new possibilities for parametric generic programming.

- In §7.4.2, we give a recursive pattern based on containers, capturing exactly (with strong type safety) the computation of an attribute grammar.

- In §8.4, we present a first class attribute grammar embedding in Haskell based on containers and using dependent type programming and type-classes to achieve the modularity of an earlier design [dMBS00] with the advantage of type safety, and all the while retaining simple types for the user; Furthermore, the DSL doesn't rely on template Haskell for its syntax.

- In §8.4, as part of the attribute grammar implementation, we present a generalisation of datatype à la carte [Swi08] using container functors.

1.5. Related Work

1.5.1. On The Productivity of Stream Equations

Most proofs of undecidability and complexity results for stream equations, like the ones of Roşu [Roş06] (discussed in depth in §4.3.1) and Simonsen [Sim09a], use straightforward encodings of Turing machines, representing the infinite band of symbols as two streams, one each for the left and right side of the band relative to the head, with canonical rewrite rules pattern matching on the current symbol. This central dispatching mechanism is unavailable in our setting. Still, we are able to recover all of these results even in the unary setting as direct corollaries of Theorem 5.2.10.

As a first taste, Theorem 5.1.3 states that our limited systems are nevertheless still sufficient to define every computable polymorphic stream function. Although the construction requires some imagination, the simulation of counter machines is quite direct and mainly intended to give the reader some intuition for the long road towards the proof of our key result, Theorem 5.2.10, which improves upon this by restricting systems to unary stream functions without mutual recursion. This is the main contribution of our work, which seems surprising given the very limited expressiveness of the syntax.

Endrullis et al. [EGH08, EGH09a, EGH$^+$07] strive to decompose rewriting into a stream layer and a data layer in such a way as to encapsulate just so much complexity into the data layer that the productivity of streams becomes decidable while still retaining usefulness of computation. Our work can be seen as a another extreme, eradicating the data layer and showing that polymorphic unary stream functions attain computational completeness. For example, our results imply that the lazy stream formats of Endrullis et al. [EGH09a] can actually be restricted to (general) unary stream functions with productivity still retaining Π^0_2-completeness (in the non-unary case, a hint of Theorem 5.1.3 can be found in their encoding of FRACTRAN-programs). We note that their notions of lazy stream specifications and data-oblivious analysis shares some points with our polymorphism restriction: choosing the unit type for the data type leaves no possibility of analysing the input. We also note that the flat stream specifications, for which the authors develop an algorithm for semi-deciding productivity, present a major difference by restricting recursion, whereas we allow general nested calls.

See Simonsen [Sim09b] for a good survey of some of the complexity analysis on stream rewriting that our developments generalise. We hope that our Turing-complete unary recursive systems, in their simplicity, may be used as a computational model in further reduction proofs (*e.g.* of complexity results) not only in rewriting theory. We conclude by remarking that all our proofs are constructive, i.e. algorithmically implementable.

Capretta [Cap10] gives a partial algorithm to recognise productive pure stream equations by generating a bisimulation relation between two solutions of the equation. If the generation terminates, then the solution is unique, and the equations productive.

1.5.2. On Circular Traversals and Attribute Grammars

The circular programming technique to implement attribute grammars and combine multiple traversals in a lazy functional language has untraceable origins. Many authors cite Bird's article as the first publication mentioning it: [Bir84] but he states that the method was known before he wrote about it and popularised it.

Deforestation can be facilitated if we are able to combine many traversals. Most deforestation techniques can fuse a consumer and a producer, and are not applicable if the intermediate data structure is consumed many times: there is clearly no fusion site. By combining the consumers, we create fusion opportunities [LS95, Nem00].

The transformation has been automatised [CGK99], as well as its inverse: to remove the circularity in order to make the programs stricter (create less thunks) and faster [FSSV11].

Attribute grammars [Knu68] are implemented in lazy functional languages with the same circular programming technique [KS87]. The authors argue that attribute grammars form the best formalism to describe such circular programs modularly. The same argument is repeated by Swierstra [Swi05]. The advantages of attribute grammars as a formalism to design complex functional programs, including combinator libraries and domain specific languages are illustrated in a number of articles, for instance [SAAS99] [SS03].

A more recent translation of attribute grammars in Haskell uses the zipper datastructure [Hue97] which consists of a sub-tree in context. Attributes are translated to mutually recursive functions on zippers [MFS13]. In [UV05], the comonadic structure of the attributed trees and zippers is exploited.

Whereas the previous articles gave a translation of attribute grammars to functional programs, other research explored the embedding of attribute grammars as first class values in Haskell. This embedding has many benefits: it is more flexible than a attribute grammar system since it is directly extensible: it is possible to define attribute grammars combinators that capture common programming patterns like the copy rule or the chain rule. The first embedding [dMBS00] has the advantage of simplicity and conciseness but is somehow lenient, and relies on dynamic checks to reject bad attribute grammars.

A more recent embedding [VSS09] uses the equally recent language extensions of GHC to enforce statically more properties of well-defined attribute grammars by means of dependent programming techniques.

The same approach allows very powerful combinators to be written, such as a macro system [VS12] in which the attributions of a non-terminal can be automatically derived from the expression of that non-terminal in terms of others.

Part I.

Productivity of Pure Stream Equations

Chapter 2.

Coinduction and Productivity

2.1. Coinductive types and Coalgebras

We use basic notions of category theory to give an abstract definition of coinductive types. Functions producing coinductive types are co-iterations of coalgebras. They are known as anamorphisms to functional programmers. The categorical explanation of inductive and coinductive types gives useful insight for structuring programs, generic programming, fusion, program calculation and avoiding boilerplate[Hag87, MFP91, Ven00, UV99, BdM97]. For a good introduction with emphasis on the categorical semantics, see [JR97]. Further details are given in Rutten's monograph on coalgebras. [Rut00].

Definition 2.1.1 (F-coalgebra)**.** *Given an endo functor F on a category $\mathcal{C}$, a F-coalgebra is a pair (A, a) of an object A and an arrow $a : A \to FA$. A homomorphism of coalgebras from a to $b : B \to FB$ is an arrow $h : A \to B$ such that $F\,h \circ a = b \circ h$.*

$$\begin{tikzcd}
F\,B & F\,A \arrow[l, "F\,h"'] \\
B \arrow[u, "b"] & A \arrow[l, "h"] \arrow[u, "a"']
\end{tikzcd}$$

For example, the base functor of Stream A – the coinductive type of infinite lists of elements of type A – is $FX = A \times X$. A stream-coalgebra is any function of type $B \to A \times B$, for any type B.

2.1.1. Terminal Coalgebra

F-coalgebras and their homomorphisms constitute a category, named Coalg_F. A terminal (often called final) coalgebra ν-out : $\nu F \to F\,(\nu F)$ is a terminal object in this category. It has the following universal property: for any F-coalgebra a, there exists a unique homomorphism $\nu\text{-it}\,a : a \to \nu$-out; The name ν-it stands for *coiteration*. The function `unfold` above is the coiteration operator for streams. The unique homomorphism is called an *anamorphism*, it is also notated $\nu\text{-it}\,a = [\![a]\!]$ with the convention in [MFP91].

$$\begin{tikzcd}
F\,(\nu F) & F\,A \arrow[l, "F\,[\![a]\!]"'] \\
\nu F \arrow[u, "\nu\text{-out}"] & A \arrow[l, dashed, "[\![a]\!]"] \arrow[u, "a"']
\end{tikzcd}$$

Because of this universal property, terminal coalgebras are unique up to isomorphism.

For instance, the type Stream A is the terminal coalgebra for the functor $FX = A \times X$. there is a function `unfold` that builds a stream from a stream-coalgebra and an element of B:

```
data Stream a = a :< Stream a
unfold :: (b → (a,b)) → b → Stream a
unfold f b = x :< unfold f b'
  where (x,b') = f b
```

In the category of sets, terminal coalgebras exist for all strictly positive functors.

Definition 2.1.2 (Strictly positive functor (SPF)). *SPF are inductively defined:*

$\overline{S}$ *is a SPF when* S *is a set*	*(constant functor)*
$\mathrm{Id}_{\mathrm{SPF}}$ *is a SPF*	*(identity functor)*
$\Sigma(i : I)F_i$ *is a SPF when* I *is a set and all* F_i *are SPF*	*(sum)*
$\Pi(i : I)F_i$ *is a SPF when* I *is a set and all* F_i *are SPF*	*(product)*
$F_1 \times F_2$ *is a SPF when* F_1 *and* F_2 *are SPF*	*(finite product)*

Most datatypes in practice are strictly positive – meaning that their base functor is strictly positive. Furthermore, in total programming languages, only strictly positive datatypes are allowed, because non-strict datatypes make a theory inconsistent: we may define an element of the empty type.

2.1.2. Fixed-Points

The categorical characterisation of a fixed-point is expressed in terms of isomorphism. In this section, we show Lambek's lemma: $F\,(\nu F) \cong \nu F$. The dual result $F\,(\mu F) \cong \mu F$ follows trivially. We write $\nu\text{-in} \triangleq \nu\text{-out}^{-1}$ and $\mu\text{-out} \triangleq \mu\text{-in}^{-1}$.

Theorem 2.1.3 (Lambek's lemma [Lam68]). *If $\omega : \Omega \to F\,\Omega$ is a terminal F-coalgebra, then ω is also an isomorphism and its inverse is the unique coalgebra homomorphism from $F\,\omega$ to ω.*

Proof. $F\,\omega : F\,\Omega \to F\,(F\,\Omega)$ is a coalgebra and ω is terminal so there is a unique homomorphism $h : F\,\omega \to \omega$. We show that, in the base category, $h : F\,\Omega \to \Omega$ is also the inverse arrow of ω by proving $h \circ \omega = \mathrm{id}_\Omega$ and $\omega \circ h = \mathrm{id}_{F\,\Omega}$.

The following diagram commutes because the left square characterises the coalgebra homomorphism $h : F\,\omega \to \omega$ and the right square is a trivial equality.

$$\begin{tikzcd}
F\,\Omega & F^2\,\Omega \arrow[l, "F\,h"'] & F\,\Omega \arrow[l, "F\,\omega"'] \\
\Omega \arrow[u, "\omega"] & F\,\Omega \arrow[l, "h"] \arrow[u, "F\,\omega"] & \Omega \arrow[l, "\omega"] \arrow[u, "\omega"']
\end{tikzcd}$$

As a consequence, the external square commutes and gives the diagram of a coalgebra homomorphism $h \circ \omega : \omega \to \omega$.

$$\begin{tikzcd}
F\,\Omega & F\,\Omega \arrow[l, "F\,(h \circ \omega)"'] \\
\Omega \arrow[u, "\omega"] & \Omega \arrow[l, "h \circ \omega"] \arrow[u, "\omega"']
\end{tikzcd}$$

Now id_Ω is already a coalgebra homomorphism corresponding to the same diagram and the terminality of ω implies that such a homomorphism is unique, so out of necessity $h \circ \omega = \mathrm{id}_\Omega$. Then, looking at the left square in the first diagram, we reason:

$$\omega \circ h = Fh \circ F\omega = F(h \circ \omega) = F\ \mathrm{id}_\Omega = \mathrm{id}_{F\,\Omega} \qquad \square$$

2.1.3. Bisimulation and Bisimilarity

A coalgebra $\alpha : A \to FA$ can be seen as a state machine or transition system [Jac05]. When the object A represents the set of states of the machine, and the coalgebra α gives the dynamics of the system, it maps a state to some observations and the successor states after a transition. Two coalgebras of the same signature functor F can be compared: we compare their (finite) observations. The behaviour of a state with respect to a coalgebra is the infinite accumulation of all the observations of this state and its successors.

Bisimulations are relations between states of two coalgebras that relate states with identical behaviour. Bisimilarity is the biggest bisimulation: it relates *all* behaviourally identical states.

A coalgebra satisfies the principle of coinduction when behaviourally identical states are equal: there is a one to one correspondence between state and behaviour.

A coalgebra is terminal if it captures *all* possible behaviours in a unique way: each possible behaviour corresponds to one state of the terminal coalgebra. In particular, a terminal coalgebra satisfies the principle of coinduction, but the converse doesn't necessarily hold.

Consequently, the coiterative operation: ν-it computes the behaviour (an element of the terminal coalgebra) of a given coalgebra for a given state.

Example 2.1.4 (Streams)**.** *A stream-coalgebra $f : B \to A \times B$ is a state machine where states are elements of B, observations are elements of A. f captures the state transitions and observable output. In the transition described by $fx = (o, y)$, x is the current state, o the output (observation), y the next state.*

The function `unfold` *computes the behaviour of the state machine: the infinite sequence of observations from a starting state $x : B$, and following the transitions given by f.*

The exposition that follows generalises [Cap11]. Another categorical presentation is given in [Bar03]. A good presentation of bisimulation with practical examples is also available in Jacob and Rutten's tutorial [JR97].

Relations and Spans

Categorically, we can represent a relation as a pair of arrows corresponding to the projections of the related components, this is called a span. In the category of sets,

$R \subseteq A \times B$ and $xRy \iff \langle x, y\rangle \in R \iff \exists r \in R.\ x = \pi_1 r \ \wedge\ y = \pi_2 r$

$$\begin{tikzcd} A & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \end{tikzcd}$$

The extension of the relation to a functor F is the image of the relation by the functor: $R^F \triangleq FR$

$$\begin{tikzcd} FA & FR \arrow[l, "F\pi_1"'] \arrow[r, "F\pi_2"] & FB \end{tikzcd}$$

Bisimulations

Definition 2.1.5. *Let α, β, F-coalgebras. a relation $\langle R, \pi_1, \pi_2\rangle$ is a $\langle \alpha, \beta\rangle$-bisimulation iff there is a coalgebra $\rho : R \to F\,R$ such that π_1 and π_2 are coalgebra morphisms $\pi_1 : \rho \to \alpha$ and $\pi_2 : \rho \to \beta$.*

$$\begin{tikzcd} F\,A & F\,R \arrow[l, "F\pi_1"'] \arrow[r, "F\pi_2"] & F\,B \\ A \arrow[u, "\alpha"] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[u, "\rho"] & B \arrow[u, "\beta"'] \end{tikzcd}$$

$\langle \alpha, \alpha\rangle$-bisimulations are simply called α-bisimulations.

In the category of sets this means

$$\forall x \forall y.\quad xRy \implies (\alpha\,x) R^F (\beta\,y)$$

Indeed, the previous statement can be written in terms of the projections:

$$\forall r \in R.\quad \exists t \in F\,R.\quad \alpha(\pi_1\,r) = F\,\pi_1\,t \quad \wedge \quad \beta(\pi_2\,r) = F\,\pi_2\,t$$

Which is equivalent to

$$\exists \rho : R \to F\,R.\quad \forall r \in R.\quad \alpha(\pi_1\,r) = F\,\pi_1\,(\rho\,r) \quad \wedge \quad \beta(\pi_2\,r) = F\,\pi_2\,(\rho\,r)$$

Corresponding to the commutative diagrams

$$\exists \rho : R \to F\,R.\quad \alpha \circ \pi_1 = F\,\pi_1 \circ \rho \quad \wedge \quad \beta \circ \pi_2 = F\,\pi_2 \circ \rho$$

Coinduction and Bisimilarity

Definition 2.1.6 (Principle of Coinduction)**.** *A coalgebra $\langle A, \alpha\rangle$ satisfies the principle of coinduction if and only if every α-bisimulation $\langle R, \pi_1, \pi_2\rangle$ verifies $\pi_1 = \pi_2$.*

In the category of sets, this is equivalent to saying that all bisimulations R are included in the diagonal relation.

$$\forall R : \alpha\text{-bisimulation}.\quad \forall x \forall y.\quad xRy \implies x = y$$

Let $\alpha : A \to FA$, $\beta : B \to FB$; we define the bisimilarity relation on $\langle \alpha, \beta\rangle$, written $\sim_{\alpha,\beta}$. It relates the elements of A and B whose behaviour is observationally identical with respect to coalgebras α and β, i.e. there exists a $\langle\alpha,\beta\rangle$-bisimulation that relates them.

Definition 2.1.7 (Bisimilarity)**.** *Bisimilarity is the large relation defined as the union of all $\langle\alpha,\beta\rangle$-bisimulations*

$$\begin{aligned} \sim_{\alpha,\beta} &= \bigcup\{R \subseteq A \times B \mid R \textit{ is a } \langle\alpha{,}\beta\rangle\textit{-bisimulation}\}, \\ \sim_{\alpha} &= \sim_{\alpha,\alpha} \end{aligned}$$

Coalgebras satisfying the principle of coinduction have the property that their bisimilarity relation is the identity relation.

Property 2.1.8. *A coalgebra $\langle A,\alpha\rangle$ satisfies the principle of coinduction if and only if $\sim_\alpha = \{\langle x,x\rangle \mid x \in A\}$. In other words, $\forall x \forall y\,.\quad x \sim_\alpha y \implies x = y$.*

Theorem 2.1.9. *If a coalgebra is terminal then it satisfies the principle of coinduction.*

Proof. If $\langle R,\pi_1,\pi_2\rangle$ is a α-bisimulation, then there is a coalgebra ρ such that $\pi_1,\pi_2 : \rho \to \alpha$. If α is terminal then $\pi_1 = \pi_2$. □

The reciprocal doesn't hold. The principle of coinduction is true for the coalgebras whose elements are in one to one correspondence with their behaviours. But that is not enough for a coalgebra to be terminal: it also must contain all the possible behaviours. For example, the set of monotone streams of natural numbers with *head* and *tail* is a coalgebra and is a strict-subset and restriction of the terminal coalgebra for the stream functor: it satisfies the principle of coinduction but isn't terminal.

Theorem 2.1.10. *If a F-coalgebra $\langle A,\alpha\rangle$ satisfies the principle of coinduction then for every F-coalgebra $\langle B,\beta\rangle$, if there exists a morphism from β to α it is unique.*

Proof. Let $f,g : \beta \to \alpha$ be two coalgebra morphisms. Then $\langle B,f,g\rangle$ is a α-bisimulation and α verifies the principle of coinduction so $f = g$. □

A slightly different formulation of the principle of coinduction is given in [Bar03]. It expresses very clearly that bisimilarity captures behavioural equivalence: two bisimilar elements x and y have equal behaviours $[\![\alpha]\!]x = [\![\beta]\!]y$ in the terminal coalgebra.

Theorem 2.1.11 (Second Coinduction Principle)**.** *Let F be an endo functor in Set that has a terminal coalgebra. Let $\alpha : A \to FA$, $\beta : B \to FB$, $x : A$, $y : B$ then*

$$x \sim_{\alpha,\beta} y \implies [\![\alpha]\!]x = [\![\beta]\!]y$$

Note that this is an equivalence, since $\{\langle x,y\rangle \mid [\![\alpha]\!]x = [\![\beta]\!]y\}$ is a $\langle\alpha,\beta\rangle$-bisimulation.

Proof. If x and y are bisimilar, they are related by a bisimulation $\langle R,\pi_A,\pi_B\rangle$, there is a coalgebra $\langle R,\rho\rangle$ such that $\pi_A : \rho \to \alpha$ and $\pi_B : \rho \to \beta$ in Coalg_F. By the terminality property, $[\![\alpha]\!] \circ \pi_A = [\![\beta]\!] \circ \pi_B = [\![\rho]\!]$ hence $[\![\alpha]\!]x = [\![\alpha]\!](\pi_A\langle x,y\rangle) = [\![\beta]\!](\pi_B\langle x,y\rangle) = [\![\beta]\!]y$. □

2.2. Productivity

Productivity is essentially the notion of well-definedness for coinductive values. It is best understood by studying the particular case of infinite lists, before explaining the general case of any coinductive type.

Although he didn't call it productivity, Dijkstra gave one of the earlier formalisations of the concept [Dij80], for the case of infinite binary trees (which he presented as string concatenation).

Basically, infinite lists are productive if all its finite prefixes are well-defined. The finite prefixes are approximations of the infinite list which is their limit.

In the general case, coinductive values are (possibly) non-well founded trees, meaning they may have arbitrary long paths. Since the tree is gradually constructed from the root, we can compute finite approximations to a certain depth. A tree is productive if it has arbitrary large well-defined finite approximations.

In fact if it is not productive, the defined object is not properly an element of the coinductive type. Such non-productive values nevertheless deserve our attention since they correspond to valid programs in weak (not strongly normalising) functional languages. In those languages the types do not correspond exactly to the coinductive types since they also contained the non-productive values. This can be a source of confusion since we still refer to them as coinductive types.

Proofs of productivity in simple cases are by induction on the length (or depth) of the approximations. In more complex cases where the coinductive result is defined by a function having both recursive and corecursive calls, we may need to use specific fixed-point theorems based on *complete ordered families of equivalences*, [DGM03] which generalise *converging equivalence relations* [Mat99].

2.2.1. Productivity and Denotational Semantics

Sijtsma wrote one of the earliest article about proving the productivity of lazy lists [Sij89]. He defined productivity as a property of the denotational semantics of lazy functional programs. His definition applies to values of *any* type, including functions and non-recursive types. Although lazy lists were the only recursive type he considered, we will generalise his definition to any recursive datatype.

The principle of denotational semantics is to map an abstract meaning to syntactic object of a language: we call *domains* the semantics of types. They are sets with some structure, whose elements are the semantics of program expressions of the corresponding type. The domains used to denote lazy programs are usually Scott domains. We quickly cover the basic definitions necessary to define Scott domains. See [AJ94] for more details on domain theory and see [Sch86, Win93] for application of domain theory to denotational semantics, and see [CD82] for the denotational semantics of lazy functional languages.

Definition 2.2.1 (partially ordered set)**.** *$(D, \sqsubseteq)$ is a (partially) ordered set, also called a poset if and only if $\sqsubseteq$ is a binary relation on D which is:*

1. *reflexive: $\forall x \in D,\ x \sqsubseteq x$,*
2. *anti-symmetric: $\forall x, y \in D,\ x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$,*

3. *transitive:* $\forall x, y, z \in D,\ x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$.

We define basic notions on ordered sets. Each notion has a dual one obtained by using the symmetric partial order relation $\sqsupseteq$.

Definition 2.2.2 (upper, lower bound)**.** *Let* $(D, \sqsubseteq)$ *a poset and* $A \subseteq D$.

1. *An element* $x \in D$ *is a* upper bound *of* A*, written* $A \sqsubseteq x$ *if and only if all elements of* A *are below it:* $\forall y \in A, y \sqsubseteq x$*. The dual is a* lower bound*, written* $x \sqsubseteq A$*. We say a set is* bounded *when it has an upper bound.*

2. *A* maximal *element* x *of* A *is an element which doesn't have greater elements in* A*:* $\forall y \in A, x \sqsubseteq y \implies x = y$*. A* minimal *element is dual.*

3. *The* greatest element *of* A *is an element of* A *which is also an upper bound of* A*. The* least element *is dual. By anti-symmetry, maximal and minimal elements are unique if they exist.*

4. *If the the set of upper bounds of* A *has a least element, it is called the* supremum *of* A*, written* $\sqcup A$*. Dually, we write* $\sqcap A$ *the* infimum *of* A *when it exists.*

5. A *is* directed *if and only if every finite subset of* A *has a supremum.*

Definition 2.2.3 (directed-complete partial order)**.** $(D, \sqsubseteq)$ *is a directed-complete partial order, abbreviated dcpo, if and only if it is a poset and every directed subset of* D *has a supremum.*

Definition 2.2.4 (Approximations, compact elements)**.** *Let* $(D, \sqsubseteq)$ *be a dcpo, let* $x, y \in D$*. We say* x *is an* approximation *of* y*, written* $x \ll y$ *if and only if, for all directed subset* $A \subseteq D$*, if* $y \sqsubseteq \sqcup A$ *then there is an element* $a \in A$*, such that* $x \sqsubseteq a$*.* x *is a* compact element *of a dcpo* $(D, \sqsubseteq)$ *if and only if* $x \ll x$*.*

Definition 2.2.5 (Scott domain)**.** *A Scott domain is a set* D *with a partial order* $\sqsubseteq$ *which is:*

1. *bounded-complete: all bounded subsets of* D *have a supremum;*
2. *directed-complete: all directed subsets of* D *have a supremum;*
3. *algebraic: every element is the supremum of a directed set of* compact *elements of* D*.*

All Scott domains D have a least element $\bot_D$: the supremum of the empty set which is trivially bounded. We write $\bot$ without the subscript when the domain is implicit.

The algebraic property of Scott domains is what allows us to define infinite objects as the limit of their finite (compact) approximations.

We will define productivity as a property of domain elements. We can subsequently define productivity on program expression as the productivity of their semantics.

We consider some domain constructions that can be used to assign semantics to types: the product, sum, exponential of domains, and a fixed-point operator. Each construction preserves the property of Scott-domains. To keep the presentation concise, we omit the verifications and many details which can be found in the previous references.

Discrete dcpo A discrete dcpo is such that its order relation is the identiy.

Lifted domain Given a domain D, we can lift it by adding a new infimum $\bot \notin D$:

$$D_\bot = \{\bot\} \cup D \qquad\qquad x \sqsubseteq_{D_\bot} y \iff x = \bot \vee x \sqsubseteq_D y$$

A *flat domain* is a lifted discrete dcpo.

Product Given two domains $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, the Cartesian product of the sets and the order relations $(A \times B, \sqsubseteq_A \times \sqsubseteq_B)$ is a domain. The suprema are computed coordinate-wise. If $X \subseteq A \times B$ has a supremum, then $\sqcup X = \sqcup(\pi_A X), \sqcup(\pi_B X)$, where π_A and π_B project the first and second coordinates.

Sums Given two domains $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, their disjoint sum is a domain, defined as:

$$A + B = \{(0, x) \mid x \in A\} \cup \{(1, y) \mid y \in B\}$$
$$(j, x) \sqsubseteq (k, y) \iff (j = k = 0 \wedge x \sqsubseteq_A y) \vee (j = k = 1 \wedge x \sqsubseteq_B y)$$

Exponentials Given two Scott domains $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, The set $A \to B$ of Scott-continuous functions is a Scott domain.

Fixed-points We can build domains which are solutions of mutually recursive domain equations of the form

$$\begin{aligned} D_1 &\cong F_1(D_1, \cdots, D_n) \\ &\cdots \\ D_n &\cong F_n(D_1, \cdots, D_n) \end{aligned}$$

Where each F_k is a domain operator made of the previous constructions: product, sums, lifting, exponentials.

The domains that are solution of the fixed-point equations can be defined as the inverse limits of retractions sequences [Sch86]. Their elements are sequences of approximations $(x_n)_{n \in \mathbb{N}}$. An element is compact iff it is a finite sequence, i.e. it is constant after some n.

Definition 2.2.6 (Productivity)**.** *We define productivity by induction on the construction of the domain.*

1. *Elements x of flat domains are productive iff $x \neq \bot$.*
2. *Elements $x \in D_\bot$ of lifted domains are productive iff $x \neq \bot$ and $x \in D$ is productive.*
3. *Functions are productive iff they preserve productivity, i.e. productive arguments yield productive results.*
4. *Tuples are productive iff all their elements all productive.*
5. *Elements of sums (l, x) are productive iff x is productive.*

6. *Elements of recursive datatypes are either compact or the supremum of a directed set of compacts approximating it. The productivity of compact elements is defined by the previous cases. An element which is not compact is productive iff all the compacts approximating it are productive.*

Productivity compared to Maximality Productivity and maximality are in many specific cases equivalent: streams of integers are productive if and only if they are maximal for instance. However, they are not comparable: a program can be productive without being maximal, and conversely. The following examples are taken from Sijtsma [Sij89].

An example of the the first case is the strict function `force_eval` :: `a` → `()` which is constant `()` for all defined arguments and undefined otherwise. It can be defined in haskell by `force_eval x = x ‘seq‘ ()`. It is not maximal since the non-strict definition: `unit` :: `a` → `()`, `unit x = ()` is greater, yet `force_eval` is productive since it does preserve productivity: productive arguments are necessarily defined and are mapped on `()`.

On the other hand, the inductively defined function `length` :: `[a]` → `Integer` on lazy lists can be proven maximal yet is unproductive since the computation diverges on all infinite lists including productive ones – a productive function must preserve productivity.

2.2.2. Productivity of Polymorphic Definitions

Polymorphic programs are productive iff their specialisation to the unit domain is productive: every type parameter is instantiated with the unit domain $\mathbb{1}_\perp$ the flat domain with a single non-bottom element.

For instance, the function `head` on streams is productive, since it maps the unique productive streams of units to unit. However, `head` on Haskell lists (they are in fact co-lists) is not productive because it diverges on the empty list.

2.2.3. Productivity and Strict Languages

Strict languages, unless they have support for lazy datastructures have the property that datastructures are productive iff their computation terminates and functions are productive iff they're total (they converge given non-bottom inputs). So productivity corresponds to termination and totality.

2.2.4. Strong Languages, Inductive Types

We defined productivity as a property of domain elements, which are the semantic values that we give to program fragments in a weak functional language. On the other hand, programs in a strongly normalising language can be modelled with sets: since the rules of the language prohibit diverging programs, there is no need for the additional structure of pointed dcpo's. Every program in a strong language must be productive if it is interpreted with partial semantics.

We can easily extend the previous definition of productivity to cover inductive types, in that case, only compact elements of the domain are productive, but not the limits (which correspond to infinite elements).

There is a caveat though: strong languages can enforce the distinction between inductive and coinductive types but weak languages cannot. So according to our previous definition, the inductively defined function `length` :: `[a]` $\rightarrow$ `Integer` on lazy lists, wouldn't be productive with partial semantic because the interpretation of finite lists with dcpos includes infinite lists and the function `length` diverges on them. However the same definition is productive in a strong language if the argument is an inductive list. On the other hand, if we want coinductive lists, then we need to make the result coinductive as well, by using coinductive natural numbers for the result. Then the function would be productive in both settings (weak and strong language).

2.2.5. Syntactic Criteria

The type checker of strong languages must ensure that coinductive definitions are productive. The most common approach used in Coq and Agda, is to require that corecursive calls are guarded by constructors, this is equivalent to giving a coalgebra, more precisely such definitions are called futumorphisms by [UV99] and the recursion pattern is dual to a course of value iteration.

The guardedness criterion could be relaxed to accept many more definitions if we based it on *lambda coiteration* [Bar03] instead. Lambda coiteration captures a corecursion pattern based on a term-coalgebra which is much slower when we must explicitly implement it. Many articles show some examples of term-coalgebras for stream computations [PE98, Rut03a, Hin11] and in Agda [Dan10].

Other works aim to expand the guardedness criterion, by the use of explicit type annotations capturing to which extent a call needs to be guarded and letting the type-checker do the work [Abe10]; or in a simply typed language where this precision cannot be achieved using types, an algorithm performs the check [TT97a, TT97b].

2.2.6. Non Productive Definitions in Type Theory

We can formalise general recursive functional programs in type theory using the partiality monad [Cap05]: partial functions from `A` to `B` are formalised as functions from `A` to `P B` where `P : Set` $\rightarrow$ `Set` is a coinductive type with two constructors: `Return : X` $\rightarrow$ `P X` and `Delay : P X` $\rightarrow$ `P X` (thus $PX = \nu T\,.\,X + T$). The `Delay` constructor acts as a guard for arbitrary recursive calls. If the computation diverge, there will be an infinite number of `Delay`, whereas if it converges, there will eventually be a `Return`. If we can prove that a computation converge, we can convert a `P X` to `X`, by extracting the value of at the `Return`. Therefore if we can prove that a function `A` $\rightarrow$ `P B` is total, we can import it in the theory, as a function `A` $\rightarrow$ `B`.

We can use the same principle to formalise general corecursive functions. A corecursive function is (trivially) productive if its result is guarded by constructors or equivalently, we can define its result as the coiteration of a coalgebra. When it is not defined in this way, we can formalise it with the *productivity monad* (a name I coined): given the base functor `F` of the coinductive result, we define a coinductive type `D F : Set` with two constructors: `Now : F (D F)` and `Later : D F`. A general recursive function `A` $\to$ ν `F` is formalised in type theory with `A` $\to$ `D F`. We can then import it to the theory as a function `A` $\to$ ν `F` if we can prove that there is *always eventually* a constructor `Now` along all paths of the result, viewed as a tree.

Bertot [Ber05] used the linear logic predicates *always eventually* to implement the function `filter` on streams: it cannot accept any streams, but only those for which the filtering predicate is always eventually true for some element. However, with our approach, we would write `filter` in the `D` monad, apply it to a predicate and any stream, and then prove that the resulting stream *always eventually* has some elements in order to cast the `D (Stream A)` to a `Stream A`.

Chapter 3.

Pure Stream Equations

Streams over some arbitrary set $\mathbb{D}$ are the basic example of a polymorphic coinductive data type, serving as case study for almost every technique dealing with infinite data structures. Although being well-explored coalgebraic objects [Dij80, Buc05, Rut03b], they have recently re-emerged in the specific setting of term rewriting [Zan10, EGSZ11]. They are usually introduced as the coinductive data type $\mathbb{S}_{\mathbb{D}}$ generated by the constructor $\cdot :: \cdot : \mathbb{D} \times \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$ (cons), appending an element to the front of a stream, and come with destructors $\mathsf{head} : \mathbb{S}_{\mathbb{D}} \to \mathbb{D}$ and $\mathsf{tail} : \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$, respectively selecting and removing the front element, and forming a terminal coalgebra. Algebraically, they can be characterised as an infinite term model parameterised by the value type $\mathbb{D}$ modulo observational equivalence on $\mathbb{D}$, also known as bisimilarity (see Definition 2.1.7).

Since this work is concerned with computability and partiality, we choose to work in a semantics of *partial streams*, adding a bottom element $\perp$ to the underlying data type. Technically, such streams are just functions of type $\mathbb{N} \to \mathbb{D}_\perp$. Note that with this terminology, if an element of a stream equals $\perp$, further elements in the sequence can still be proper inhabitants of $\mathbb{D}$. Also, when speaking of computable functions, we always mean partial computable functions. Everywhere the term stream is used in the following chapters, partial stream must be understood.

Our problem, addressed in Chapter 4 and Chapter 5, is to decide productivity of pure stream equations. They specify polymorphic stream functions $f : \forall\alpha\,.(\mathbb{S}_\alpha)^n \to \mathbb{S}_\alpha$; where $\mathbb{S}_\alpha$ is the type of streams (infinite sequences) of elements of type α. The purpose of the current chapter is to give the notations, definitions and properties used in the later technical chapters.

Overview The chapter has two sections, §3.1 gives important characterisations of polymorphic stream functions in terms of their *indexing functions* which are essential in many of our proofs. §3.2 defines pure stream equations (PSE) which is the object of study in Chapter 2 and Chapter 5. Examples and properties of PSE are also given in that section and will be used in the later chapters.

3.1. Streams and Polymorphic Functions

We define the abstract notion of polymorphism (§3.1.1) and we show what the polymorphic property specifically means in the case of streams. We study polymorphism from two different viewpoints: the semantical view of streams as functions on natural numbers (§3.1.2) and the coinductive view of streams as coalgebras (§3.1.4) and coalgebraic equations as a specification of polymorphic stream functions, preparing the reader for syntactic approach of §3.2. In §3.1.3 we define *indexing functions*, which characterise polymorphic stream functions. §3.1.5 concludes by explaining how the different notions introduced in the section will be used in later chapters to prove the productivity of polymorphic streams equations.

3.1.1. Polymorphism

The word *polymorphism* literally means *having many shapes*. We use it to characterise the functions which can be applied to arguments of different types. This can be realised in two contrasting manners [Str00].

1. By writing specialised implementations for different types. We call this polymorphism *ad hoc*. Haskell type classes [WB89] fall in this category. There is no common property between the specific instances of the same function.

2. By writing a universal definition over abstract parameter types, for instance the projection $\pi : a \times b \to a$; $\pi\langle x, y\rangle = x$. We call this polymorphism *parametric*; All such functions enjoy a parametricity property which reflects the fact that the computation cannot rely on the actual values of the abstract types. It corresponds to the categorical notion of naturality.

Definition 3.1.1 (Parametricity property). *If F and G are two type constructors, functorial over the category of types, then a function $\phi : \forall\alpha\,.\,F\,\alpha \to G\,\alpha$ is parametrically polymorphic if and only if it is natural in α.*

$$\forall\alpha,\beta\,.\,\forall f : \alpha \to \beta\,.\,\forall x \in F\,\alpha\,.\,\phi\,(\mathrm{map}_F\;f\,x) = \mathrm{map}_G\;f\,(\phi\,x)$$

We now study two representations for streams and view how parametrically polymorphic stream functions can be characterised in each case.

3.1.2. Polymorphic Streams Functions

For our study of stream productivity, we work in a set semantics. The equations that we consider can define non-productive streams with diverging elements. To model those undefined elements we add a special semantic value $\bot$ that doesn't belong to any set and which is used to denote partial or non-terminating computation on the object-level of stream reduction. A partial function from A to B is written $f : A \to B_\bot$, and for function composition we adopt the convention of implicitly extending the domain of f to $A_\bot$ by setting $f(\bot) = \bot$.

Streams are modelled as *partial* functions on natural numbers. The set of (partial) streams over $\mathbb{D}$ is defined as $\mathbb{S}_\mathbb{D} := \mathbb{N} \to \mathbb{D}_\bot$. A stream is productive iff its denotation s is a total function i.e. $s(k) \neq \bot$ for all $k \in \mathbb{N}$.

Stream functions are modelled as (total) functions on partial streams, for instance $f : \mathbb{S}_A \to \mathbb{S}_B$. We do not need to lift the result set with a bottom element, because there is already a value in $\mathbb{S}_B$ that stands for an undefined value: the completely undefined stream that we can write as $\bot$ and defined by $\forall x \in B\,.\,\bot\,x = \bot$. That stream functions are total isn't very helpful: they are defined for all partial streams and return another partial stream. The interesting stream functions, when restricted to the total streams $\mathbb{N} \to A$ yield total streams $\mathbb{N} \to B$. we call such stream functions *productive*, according to Definition 2.2.6.

Definition 3.1.2. *A* polymorphic stream function*, abbreviated PSF, is a family of functions*

$$f_{\mathbb{D}} : \mathbb{S}_{\mathbb{D}} \times \cdots \times \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$$

*natural (*parametric*) in the domain argument $\mathbb{D}$, i.e. for all sets α, β and functions $g : \alpha \to \beta_{\perp}$, we have*

$$\operatorname{map} g \circ f_{\alpha} = f_{\beta} \circ \langle \operatorname{map} g, \cdots, \operatorname{map} g \rangle$$

Equivalently,

$$\forall s_1 \cdots s_n \in \mathbb{S}_{\alpha}^n . \quad \operatorname{map} g \, (f_{\alpha} \langle s_1, \cdots, s_n \rangle) = f_{\beta} \langle \operatorname{map} g \, s_1, \cdots, \operatorname{map} g \, s_n \rangle$$

where $\operatorname{map} g : \mathbb{S}_{\alpha} \to \mathbb{S}_{\beta}; \operatorname{map} g \, s \triangleq g \circ s$.

Example 3.1.3. *As an example, the stream function $f\, s = \lambda n \,.\, s(2 \times \lfloor n/2 \rfloor)$ computes the stream consisting of the input elements with an even index, each given twice:*

$$f\,(s_0 : s_1 : s_2 : s_3 : s_4 : s_5 : s_6 : \cdots) = s_0 : s_0 : s_2 : s_2 : s_4 : s_4 : s_6 : s_6 : \cdots$$

Let us verify that f is polymorphic. For any types α, β, and function $g : \alpha \to \beta$, $g \circ (f\, s) = \lambda n \,.\, g(f\, s\, n) = \lambda n \,.\, g(s(2 \times \lfloor n/2 \rfloor)) = \lambda n \,.(g \circ s)(2 \times \lfloor n/2 \rfloor) = f(g \circ s)$.

As this example illustrates, polymorphic stream functions are parametric in the data type $\mathbb{D}$ and therefore cannot apply any operation on elements of $\mathbb{D}$; in particular case analysis on the elements is not possible. Consequently, the only operation polymorphic stream functions do is discarding, duplicating, and reordering the elements of its input streams. Therefore every element of the output stream necessarily occurs in the input stream. We show in the next section that polymorphic stream functions are characterised by an *indexing function* mapping output indices to input indices so that every element of the output stream corresponds to an element of an input stream. In the previous example, the indexing function is $\lambda n \,.\, 2 \times \lfloor n/2 \rfloor$.

3.1.3. Indexing Functions

The correspondence between polymorphic stream functions and their indexing function is a direct application of the Yoneda lemma.

Lemma 3.1.4 (Yoneda)**.** *Let F be a covariant endofunctor in a category $\mathcal{C}$. Let A be an object of $\mathcal{C}$. Let $\mathrm{h}^A : \mathcal{C} \to \mathrm{Set}$, the covariant functor defined by $\mathrm{h}^A\, X = X \to A$*

and $\mathrm{h}^A\, u = \lambda v\,.\, u \circ v$. *Then the natural transformations* $\mathrm{h}^A \dot{\to} F$ *are isomorphic to* $F\,A$. *The isomorphism is given by:*

$$\begin{array}{ll} \bar{\cdot} : (\mathrm{h}^A \dot{\to} F) \to F\,A & \hat{\cdot} : F\,A \to (\mathrm{h}^A \dot{\to} F) \\ \overline{f} = f(\mathrm{id}_A) & \hat{x} = \lambda u\,.\, F\,u\,x \end{array}$$

Proof. We prove that the functions $\bar{\cdot}$ and $\hat{\cdot}$ are inverse of each other.
$\overline{\hat{x}} = \hat{x}(\mathrm{id}_A) = F\ \mathrm{id}_A\ x = \mathrm{id}_{FA}\, x = x$.
$\hat{\overline{f}} = \lambda u.Fu\overline{f} = \lambda u.Fu(f\,\mathrm{id}_A) = \lambda u.f\,(\mathrm{h}^A\, u\, \mathrm{id}_A) = \lambda u.f(u \circ \mathrm{id}_A) = \lambda u.fu = f$. □

We first consider the simpler case of unary functions. As we explained earlier, they are polymorphic and thus correspond to the natural transformations from functor $\mathbb{S}$ to $\mathbb{S}$. The functor $\mathbb{S}$ is in fact isomorphic to the hom functor: $\mathrm{h}^{\mathbb{N}}$. The Yoneda lemma states that the natural transformations $h^{\mathbb{N}} \to \mathbb{S}$ are isomorphic to $\mathbb{S}_(\mathbb{N})$ thus $\mathbb{N} \to \mathbb{N}$. The isomorphism is given by

$$\begin{array}{ll} \bar{\cdot} : (\mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}) \to (\mathbb{N} \to \mathbb{N}) & \hat{\cdot} : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}) \\ \overline{f} = f(\mathrm{id}_{\mathbb{N}}) & \hat{i} = \lambda s\,.\, s \circ i \end{array}$$

Note that we identified streams with functions, and that $s \circ i$ equals map $s\,i$ and corresponds to the functorial map for streams.

For the general case of n-ary stream functions $f : \forall \mathbb{D}\,.(\mathbb{S}_{\mathbb{D}})^n \to \mathbb{S}_{\mathbb{D}}$, we identify the set $(\mathbb{S}_{\mathbb{D}})^n$ with $I_n \to \mathbb{D}$ where $I_n = \{1, \cdots, n\} \times \mathbb{N}$ is the set of input indices of an n-ary PSF. The elements $\langle i, j\rangle$ of I_n are pairs of the index of a stream in the argument n-uple and an index of an element in this stream. We will identify I_1 with $\mathbb{N}$. Again applying the Yoneda lemma states that the set of n-ary PSF is isomorphic to $\mathbb{S}_{I_n}$, or equivalently, functions $\mathbb{N} \to I_n$: an indexing function takes an output index and returns an input index.

We can characterise indexing functions concretely using an n-ary indexing operation that projects one element from a tuple of streams: $.!!_{n,\mathbb{D}}\,.\, : (\mathbb{S}_{\mathbb{D}})^n \to (I_n)_\perp \to \mathbb{D}_\perp$. We usually omit the subscript since it is clear from the context.

$$\begin{array}{lcll} \langle s_1, \cdots, s_n\rangle & !! & \langle i, j\rangle & \triangleq\ s_i(j) \\ \langle s_1, \cdots, s_n\rangle & !! & \perp & \triangleq\ \perp \end{array}$$

We define some streams of indices: For $k \in [1 \cdots n]$, let $\mathrm{id}_k : \mathbb{S}_{I_n}, \mathrm{id}_k(j) \triangleq (k, j)$.

Property 3.1.5. *For all* $x \in I_n$, $\langle \mathrm{id}_1, \cdots \mathrm{id}_n\rangle !! x = x$

Proof. $\langle \mathrm{id}_1, \cdots, \mathrm{id}_n\rangle !! \langle i, j\rangle = \mathrm{id}_i(j) = \langle i, j\rangle$ □

Definition 3.1.6 (Indexing Functions)**.** *If f is a polymorphic stream function of arity n, its indexing function is the function* $\overline{f} : \mathbb{N} \to (I_n)_\perp = \mathbb{S}_{I_n}$ *defined by* $\overline{f} \triangleq f_{I_n}(\mathrm{id}_1, \cdots, \mathrm{id}_n)$*, it verifies:* $\forall \alpha\,.\forall s_1 \cdots s_n \in (\mathbb{S}_\alpha)^n\,.\forall i, j, k \in \mathbb{N}\,.$

$$f_\alpha(s_1, \cdots, s_n)\, k = \langle s_1, \cdots, s_n\rangle !! \overline{f}(k) \tag{3.1.1}$$

Equivalently,

$$\begin{aligned}\overline{f}(k) = \langle i, j\rangle &\iff f_\alpha(s_1, \cdots, s_n)\, k = s_i\, j \\ \overline{f}(k) = \bot &\iff f_\alpha(s_1, \cdots, s_n)\, k = \bot\end{aligned}$$

Reciprocally, for any $\overline{f} \in \mathbb{S}_{I_n}$, we can define a PSF $f_{\mathbb{D}}$ such that $\overline{f}$ is its indexing function. We simply read equation (3.1.1) as a definition:

$$f_{\mathbb{D}}(s_1, \cdots, s_n) = \lambda k\,.\langle s_1, \cdots, s_n\rangle !! \overline{f}(k)$$

Polymorphic stream functions and indexing functions of the same arity are in bijective correspondence. It might come as a surprise that a set of stream functions is bijective to a set of streams!

We conclude the section with some remarks and examples.

Productivity and Indexing Functions

Lemma 3.1.7. *A polymorphic stream function is productive if and only if its indexing function is total.*

It is an interesting insight that the productivity of a function on streams is reduced to the totality on a function on natural numbers. In subsequent chapters we use this equivalence to show different properties of polymorphic stream functions.

Contravariance It is worth noting that indexing functions represent an inversion of the usual notions of input and output for stream functions. This contravariance is reflected in $\overline{f \circ h} = \overline{h} \circ \overline{f}$ for polymorphic $f_{\mathbb{D}}, h_{\mathbb{D}} : \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$, a fact we use repeatedly in Chapter 5.

Computable Functions Let us stress that this thesis is only concerned with computable functions, so we mean computable stream functions, computable indexing functions, and computable streams.

Container Morphisms The *containers* [AAG05] generalise the representation of streams as function types to every strictly positive type. What we call indexing function is a container morphism for streams using their terminology.

Examples Some basic polymorphic stream functions are the tail operation:

$$\begin{aligned}\mathrm{tail}_{\mathbb{D}} &: \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}} \\ \mathrm{tail}(s) &= \lambda i\,.\, s(i+1)\end{aligned} \quad \text{with indexing function:} \quad \overline{\mathrm{tail}}(i) = i + 1$$

and the combined head-cons operation:

$$\begin{aligned}(\mathrm{head}(\cdot) :: \cdot)_{\mathbb{D}} &: \mathbb{S}_{\mathbb{D}} \times \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}},\\ \mathrm{head}(s) :: t &= \lambda i \,.\begin{cases} s(0) & \text{if } i = 0,\\ t(i-1) & \text{otherwise}\end{cases}\end{aligned}$$

with indexing function:

$$\overline{\mathrm{head}(\cdot) :: \cdot}(i) = \begin{cases} (0,0) & \text{if } i = 0,\\ (1, i-1) & \text{otherwise.}\end{cases}$$

3.1.4. Coalgebraic Stream Equations

Streams were presented in §3.1.2 as functions from natural numbers, but we can also view them as infinite sequences. In programming they are implemented as lazy datastructures. Formally, infinite sequences are a coinductive type as in §2.1.

Coinductive types are characterised by their final coalgebra: a function that let us deconstruct a value to extract some finite information about it. The two primitive operations to inspect a stream are $\mathrm{head}_{\mathbb{D}} : \mathbb{S}_{\mathbb{D}} \to \mathbb{D}$ and $\mathrm{tail}_{\mathbb{D}} : \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$. The head yields an observation in $\mathbb{D}$: the first element of the stream; and the tail is the rest the stream which in turn can be inspected. $\langle \mathrm{head}_{\mathbb{D}}, \mathrm{tail}_{\mathbb{D}}\rangle$ form a stream coalgebra. A $\mathbb{D}$-stream coalgebra is a function c which maps a state in A to an output element and a next state in $\mathbb{D} \times A$. The type of streams $\mathbb{S}_{\mathbb{D}}$ over $\mathbb{D}$ captures the behaviours of stream coalgebra, the existence and unicity of a behaviour map from any coalgebra make $\langle \mathrm{head}_{\mathbb{D}}, \mathrm{tail}_{\mathbb{D}}\rangle$ a terminal coalgebra: for any coalgebra $c : A \to \mathbb{D} \times A$, there is a unique mapping from A to $\mathbb{S}_{\mathbb{D}}$, called behaviour or coiteration of c and written $\nu\text{-it}\, c : A \to \mathbb{S}_{\mathbb{D}}$, which recursively inspects the state in the result of c and preserves the observations on $\mathbb{D}$.

$$\mathrm{head}_{\mathbb{D}} \circ \nu\text{-it}\, c = \pi_1 \circ c \;; \qquad \text{and} \qquad \mathrm{tail}_{\mathbb{D}} \circ \nu\text{-it}\, c = \nu\text{-it}\, c \circ (\pi_2 \circ c)$$

The representation of streams as functions from natural numbers is justified because $\lambda(f \in \mathbb{D}^{\mathbb{N}})\,.\langle f\,0, \lambda n\,.\,f\,(n+1)\rangle$ is also a terminal stream coalgebra.

Following the coalgebraic approach, stream functions are specified by laying down equations on heads and tails of their input and output streams. A function has the parametricity property when stream elements are never inspected nor any operation be carried on them. To illustrate such equations, we give the simple example of a function which swaps every other element of its input stream. For all domain type $\mathbb{D}$ and all stream $s \in \mathbb{S}_{\mathbb{D}}$:

$$\begin{aligned}\mathrm{head}(f s) &= \mathrm{head}(\mathrm{tail}\, s)\\ \mathrm{head}(\mathrm{tail}(f s)) &= \mathrm{head}\, s\\ \mathrm{tail}(\mathrm{tail}(f s)) &= f(\mathrm{tail}(\mathrm{tail}\, s))\end{aligned}$$

In §3.2, we define pure stream equations. They are parametric coalgebraic equations defined in terms of the polymorphic stream co-constructor $.::. : \mathbb{D} \to \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$, coalgebraically defined by:

$$\forall d \in \mathbb{D}, \forall s \in \mathbb{S}_{\mathbb{D}}, \quad \mathrm{head}(d :: s) = d \quad \mathrm{tail}(d :: s) = s$$

With the co-constructor we can express the previous example as a single equation.

$$f s = \mathrm{head}(\mathrm{tail}\, s) :: \mathrm{head}\, s :: f(\mathrm{tail}(\mathrm{tail}\, s))$$

Stream equations of this form define a total coalgebra only if they verify the guardedness criterion §2.2.5.

Partial Streams Partial streams are coalgebraically defined as above with a domain $\mathbb{D}_\perp$ extending $\mathbb{D}$ with a bottom element $\perp$. A partial stream coalgebra over domain set $\mathbb{D}$ and carrier A is thus a stream coalgebra $A \to \mathbb{D}_\perp \times A$. $\langle \mathrm{head}_{\mathbb{D}_\perp}, \mathrm{tail}_{\mathbb{D}_\perp} \rangle$ and $\lambda(f \in (\mathbb{D}_\perp)^{\mathbb{N}}) . \langle f\, 0, \lambda n\, .\, f\,(n+1) \rangle$ are terminal coalgebras.

To specify a partial stream the guardedness criterion is no longer necessary, and general recursive equations can be used. Pure stream equations define partial streams coalgebraically using general recursion.

3.1.5. The Indexing Functions of Coalgebraic Equations

Our main interest for the thesis is the productivity of coalgebraic stream equations. All our theoretical results are derived from the observation that indexing functions characterise polymorphic functions: thus we can prove properties about coalgebraic stream equations by looking at their indexing functions.

In Chapter 4 we associate to any pure stream equation a (weak) functional program which computes its (partial) indexing function. This program is total if and only if the pure stream equation is productive.

In §5.1 we start from an indexing function and try to derive a pure stream equation whose semantics has the same indexing function, thus answering the question: do the pure stream equations capture all polymorphic stream functions? This strengthens (implies) the previous result.

In §5.2 we strengthen the result even more by finding subsets of pure stream equations that can capture all polymorphic stream functions. The subsets are obtained by giving syntactic limitations.

3.2. Pure Stream Equations

Chapter 4 and Chapter 5 are concerned with the study of pure stream equations. They can be viewed as very restricted functional programs on streams. The restrictions are needed so that we can focus on specific aspects of corecursive programs, our goal being to identify which elements, which property of programs cause productivity to be undecidable: where can we draw the frontier between undecidable and decidable productivity? Can we express this frontier as a syntactic criterion?

Overview Pure stream equations systems (PSES) are introduced by means of an example in §3.2.1. PSES are defined as syntactic objects in §3.2.2 then given a semantics as polymorphic stream functions in §3.2.3. We then define in §3.2.4 a family of stream operations for interleaving many streams and projecting one of the interleaved streams. Those operations allow us to deal with tuples of streams as single streams, and thus PSES can be equivalently formulated as a single equation involving interleaving and projections, which we call a zip-proj equation §3.2.5.

3.2.1. Introduction

Pure stream equations systems (PSES) specify polymorphic stream functions. They are constituted of mutually recursive equations involving only primitive operations on streams: head and tail, and the binary operator $::$. $\langle \text{head}, \text{tail}\rangle : \mathbb{S}_\mathbb{D} \to \mathbb{D} \times \mathbb{S}_\mathbb{D}$ is a final coalgebra and $(.::.) : \mathbb{D} \times \mathbb{S}_\mathbb{D} \to \mathbb{S}_\mathbb{D}$ is its inverse.

As a representative example, consider the system:

Example 3.2.1 (Hanoi)**.**

$$\begin{aligned}
\text{const}(s) &= \text{head}(s) :: \text{const}(s),\\
\text{zip}_2(s,t) &= \text{head}(s) :: \text{zip}_2(t, \text{tail}(s)),\\
\text{hanoi}(s) &= \text{zip}_2(\text{hanoi}(\text{tail}(s)), \text{const}(s)).
\end{aligned}$$

The function "const" returns a constant stream of the first element of its argument, "zip_2" produces the interleaving of two streams: elements from each stream are given alternatively. Through evaluation, which will be elaborated in §3.2.4 p. 38, we find that:

$$\text{hanoi}(s) = \bot :: s(0) :: s(1) :: s(0) :: s(2) :: s(0) :: s(1) :: s(0) :: s(3) :: \cdots.$$

The corresponding indexing function is $\overline{\text{hanoi}}(k) = \max\{v \text{ such that } 2^v \text{ divides } k\}$ where $\max\{\mathbb{N}\} := \bot$. To explain the naming, if the type of elements $\mathbb{D}$ is instantiated with the set of disks of an infinite Tower of Hanoi and $s \in \mathbb{S}_\mathbb{D}$ is a list of the disks sorted by increasing size, then $\text{tail}(\text{hanoi}(s))$ is a walkthrough for coinductively solving the puzzle, the k-th stream element being the disk to be moved in the k-th step, with the smallest disk always moving in the same direction [Hin89].

We want to study the productivity of pure stream equations. The main restriction we considered is polymorphism. A productive stream must have converging elements. A non-polymorphic function has many possibilities to create diverging elements because of the specific operations of the elements' type. Without restriction at all on the elements' type, we would already have an undecidable problem to prove that the head of the stream converges, let alone to prove that the whole stream is productive. Polymorphism, as explained in §3.1.1 by virtue of the naturality property imposes a good behaviour to the stream function: it can take elements of its own input streams and place them in the output stream, reorganising them around. If the inputs are assumed productive (the hypothesis of function productivity, see Definition 2.2.6) then their elements are converging. So where would divergence in the result stream come from? Recursive calls! Indeed, the simplest non-productive stream equation is:

$$f\,s = f\,s$$

Read as a specification, it imposes no restriction to the solutions of the equations: no observation on the elements is given. The least solution, yields the undefined stream. $\forall \sigma \in \mathbb{S}_{\mathbb{D}} \,.\, \forall n \in \mathbb{N} \,.\, (f_{\mathbb{D}} \sigma)\, n = \bot$ with undefined indexing function $\forall n \,.\, \overline{f}(n) = \bot$.

Pure stream equations would be extremely boring and trivially productive if there weren't recursive calls. In fact we prove in the following chapters that the addition of recursive calls make productivity undecidable (when considering at least two functions). Here are a few obviously non-productive polymorphic stream functions (non-mutually recursive). The first two don't even use their input elements.

$$\begin{aligned}
\text{bottom}\, s &= \text{bottom}\, s \\
\text{stream-of-bottoms}\, s &= \text{head}(\text{stream-of-bottoms}\, s) :: \text{stream-of-bottoms}\, s \\
\text{partial-stream}\, s &= \text{head}\, s :: \text{tail}(\text{partial-stream}\, s) \\
\text{holes}\, s &= \text{head}\, s :: \text{head}(\text{tail}(\text{holes}\, s)) :: \text{holes}(\text{tail}\, s)
\end{aligned}$$

Their least solution, given a productive input $s_0 :: s_1 :: s_2 :: \cdots$.

$$\begin{aligned}
\text{bottom}\, s &= \bot \\
\text{stream-of-bottoms}\, s &= \bot :: \bot :: \bot :: \bot :: \cdots \\
\text{partial-stream}\, s &= s_0 :: \bot \\
\text{holes}\, s &= s_0 :: \bot :: s_1 :: \bot :: s_2 :: \bot :: \cdots
\end{aligned}$$

Remark 3.2.2. *In Haskell where the product is lifted, we make a semantic distinction between* bottom *and* stream-of-bottom*: when forcing their evaluation through a case expression matching* bottom *fails, but* stream-of-bottoms *succeeds and binds its* head *and* tail *to undefined values that would diverge if they are evaluated.*

However, if we extend the notion of bisimilarity (see §2.1.3) to partial streams, bottom *and* stream-of-bottoms *are bisimilar: they're both undefined everywhere. With the set semantics we use in this chapter, we consider them equal.*

3.2.2. The Syntax of Pure Stream Equations

We define a simple form of a recursive system of equations for specifying polymorphic stream functions. We call them pure stream equation systems, abbreviated PSES. They are syntactic objects defined over four sets of disjoint alphabets providing symbols for function variables (f_k), stream parameters (s_j), constants (head, tail, ::) and meta-symbols (equal sign, parenthesis, and spaces). A system is a family, indexed by $k \in [1 \cdots n]$, of stream equations of the following form:

$$f_k(s_0, \cdots, s_{p_k - 1}) = \sigma_k$$

where the right hand side σ_k is a stream term, defined below. In addition, we require that there is only one equation per function variable, i.e. the f_k are distinct; for each equation, the parameters are distinct; and any parameter in σ_k has to be listed on the left hand side. Each function has an arity p_k equal to the number of parameters it has on its defining equation. Function calls may occur on right hand side under the condition that it is defined by one equation and that it is given a number of stream terms arguments equal to its arity.

Stream terms are defined by the following generative grammar.

σ	$::=$	s_i	a stream parameter ($i \in [0..p_k - 1]$)
	$\mid$	$\mathrm{tail}(\sigma)$	a stream stripped of its first element
	$\mid$	$\mathrm{head}(\sigma) :: \sigma'$	the head of stream σ followed by a stream σ'
	$\mid$	$f_k(\sigma_0, \cdots, \sigma_{p_k-1})$	a call to a stream function ($k \in [1..n]$)

We take the convention that $\mathrm{head}\,\sigma :: \sigma'$ means $\mathrm{head}(\sigma) :: \sigma'$. We use the shortcut $\mathrm{tail}^k(\sigma)$ for $\mathrm{tail}(\mathrm{tail}(\cdots(\mathrm{tail}(\sigma))))$ with tail composed k times.

Definition 3.2.3 (Unary PSES). *A system is called* unary *if all defined stream functions are unary, i.e.* $p_k = 1$ *for all* k.

We stress that there are no further restrictions such as guardedness on the form of the stream terms σ_k since we specifically deal with non-productive equations using our partiality semantics (see §3.1.4). By a canonical application of the Kleene fixed-point theorem [GHK+03], each stream equation system of size n gives rise to n corresponding polymorphic stream functions, the least fixed-point of the given system of equations with respect to the partial ordering on $\mathbb{D}_\perp$ generated by $\perp < d$ for $d \in \mathbb{D}$ when the tail and head-cons operations are interpreted according to §3.1.3. An equation in the system is called *productive* if the polymorphic stream function it defines is productive. In what follows, we will usually use the same symbols to denote syntactic occurrences and their semantic counterparts as their meaning is always clear from the context.

In general many productive stream functions are solution of a pure equation. Only when the least solution is productive then it is unique. For instance the equation $fs = fs$ has all polymorphic stream functions (productive or not) as solutions.

3.2.3. Semantics of Pure Stream Equations

We can view a pure stream equation system as an executable specification for functions on partial streams. We give here an operational semantics to the equations that constructs the least solution. For this, we formalise the computation of stream elements using a functional (reduction) relation $\leadsto$ on stream terms and indices. We note $\sigma\ !\ n$ for the pair of a stream term σ and an index $n \in \mathbb{N}$. We write $\leadsto^m$ for the composition of $\leadsto$ with itself m times, $\leadsto^+$ for its transitive closure, $\leadsto^*$ for its reflexive and transitive closure.

Definition 3.2.4 (Indexing Reduction $\leadsto$). *The definition is inductive on stream* *terms and all the variables are universally quantified.*

$$\begin{aligned}
\mathrm{tail}\,\sigma \;!\; n &\rightsquigarrow \sigma \;!\; n+1 \\
\mathrm{head}(\sigma) :: \sigma' \;!\; n &\rightsquigarrow \begin{cases} \sigma \;!\; 0 & \textit{if } n = 0 \\ \sigma' \;!\; n-1 & \textit{otherwise} \end{cases} \\
f(\sigma_0, \cdots, \sigma_{p-1}) \;!\; n &\rightsquigarrow \rho[\forall i \,.\, s_i \leftarrow \sigma_i] \;!\; n
\end{aligned}$$

Where $f(s_0, \cdots, s_{p-1}) = \rho$ is an equation of the system and $\rho[\forall i \,.\, s_i \leftarrow \sigma_i]$ is the parallel substitution in term ρ of parameters s_i with σ_i, for all $i \in [0, p-1]$.

This is effectively equivalent to introducing a rewriting system on constructs of the form $\mathrm{head}(\mathrm{tail}^k(\sigma))$ based on the following rules with a deterministic outermost rewriting strategy.

$$\begin{aligned}
\mathrm{head}(\mathrm{head}(\sigma) :: \sigma') &\to \mathrm{head}(\sigma), \\
\mathrm{tail}(\mathrm{head}(\sigma) :: \sigma') &\to \sigma'
\end{aligned}$$

Note that indexing a stream variable is irreducible, meaning we reached a point when the value is an element of a stream argument. Every reduction sequence is unique, this allows us to express the semantics of the equation as a polymorphic stream function. Let f be a function symbol of arity p in a PSES. For a given domain $\mathbb{D}$, we note $f_{\mathbb{D}}$ the least solution of f.

Definition 3.2.5 (Least Solution of a Pure Stream Equation)**.**

$$f_{\mathbb{D}}(s_1, \cdots, s_p) \triangleq \lambda x \,.\, \begin{cases} s_i(j) & \textit{if } f(s_1, \cdots, s_p) \;!\; x \rightsquigarrow^* s_i \;!\; j, \\ \bot & \textit{otherwise} \end{cases}$$

for $j \in \{0, \cdots, n-1\}$. It is easy to see that this is indeed the smallest solution to the given specification and fulfils the parametricity property. From this, we can express the indexing function $\overline{f} \in \mathbb{S}_{I_p}$ as

Definition 3.2.6 (Indexing Function of a Pure Stream Equation)**.**

$$\overline{f}(x) \triangleq \begin{cases} (i, j) & \textit{if } f(s_1, \cdots, s_p) \;!\; x \rightsquigarrow^* s_i \;!\; j, \\ \bot & \textit{otherwise.} \end{cases}$$

It is clear that $\overline{f}$ is the indexing function of $f_{\mathbb{D}}$ as expressed in Definition 3.1.6.

Definition 3.2.7 (Productivity of a Pure Stream Equation)**.** *A pure stream equation f is productive if and only if its least solution $f_{\mathbb{D}}$ is productive.*

Corollary 3.2.8. *A PSE f is productive if and only if its indexing function $\overline{f}$ is total.*

Proof. By Definition 3.2.7 and Lemma 3.1.7. □

Property 3.2.9. *A polymorphic function is productive iff any specialisation to a concrete type is productive.*

Proof. In Definition 3.2.5 it appears that the well-definedness of the output stream elements doesn't depend on actual values of input stream elements (assuming input streams are productive). □

3.2.4. Interleaving and Projection

We define some stream equations that we need for the developments in the next chapters. They are all productive and we give their (total) indexing function.

We start by proving a property of the indexing reduction.

Lemma 3.2.10. $\mathrm{head}\,\sigma_i :: \cdots :: \mathrm{head}\,\sigma_{i+n-1} :: \sigma \ ! \ r \ \rightsquigarrow^* \begin{cases} \sigma_{i+r} \ ! \ 0 & \textit{if } r < n \\ \sigma \ ! \ r-n & \textit{otherwise} \end{cases}$

Proof. An induction on n. The base case $n = 0$ follows from reflexivity of $\rightsquigarrow^*$. Assume the previous property holds for some $n \in \mathbb{N}$ (for all σ, i, r) we prove it holds for $n+1$ by case analysis on $r \in \mathbb{N}$.

If $r = 0$, $\qquad \mathrm{head}\,\sigma_i :: \cdots :: \mathrm{head}\,\sigma_{i+n} :: \sigma \ ! \ 0 \ \rightsquigarrow \ \sigma_i \ ! \ 0$

If $1 \leq r < n+1$,

$$\begin{aligned} \mathrm{head}\,\sigma_i :: \cdots :: \mathrm{head}\,\sigma_{i+n} :: \sigma \ ! \ r &\rightsquigarrow \mathrm{head}\,\sigma_{i+1} :: \cdots :: \mathrm{head}\,\sigma_{(i+1)+n-1} :: \sigma \ ! \ r-1 \\ &\rightsquigarrow^* \sigma_{(i+1)+r-1} \ ! \ 0 \qquad \text{(by hyp.)} \\ &= \sigma_{i+r} \ ! \ 0 \end{aligned}$$

If $r \geq n+1$,

$$\begin{aligned} \mathrm{head}\,\sigma_i :: \cdots :: \mathrm{head}\,\sigma_{i+n} :: \sigma \ ! \ r &\rightsquigarrow^* \mathrm{head}\,\sigma_{i+n} :: \sigma \ ! \ r-n \qquad \text{(by hyp.)} \\ &\rightsquigarrow \sigma \ ! \ r-n-1 \\ &= \sigma \ ! \ r-(n+1) \end{aligned}$$ □

Interleaving

For $n \in \mathbb{N}, n > 0$ we define a pure stream equation zip_n which takes n streams and interleaves their elements, outputting the first element of each stream before interleaving their tails:

$$\mathrm{zip}_n(s_1, \cdots, s_n) = \mathrm{head}\,s_1 :: \cdots :: \mathrm{head}\,s_n :: \mathrm{zip}_n(\mathrm{tail}\,s_1, \cdots, \mathrm{tail}\,s_n)$$

Equivalently, it starts with the first element of the first stream and interleaves the others streams and the tail of the first:

$$\mathrm{zip}_n(s_1, \cdots, s_n) = \mathrm{head}\,s_1 :: \mathrm{zip}_n(s_2, \cdots s_n, \mathrm{tail}\,s_1)$$

In the previous equation the recursive call is *guarded* by the stream constructor and therefore is productive; see §2.2.5. Its indexing function is thus total, with:

Lemma 3.2.11. $\overline{\mathrm{zip}_n}(k) = (k \bmod n, \lfloor k/n \rfloor)$

Proof. By Definition 3.2.6 it is enough to prove by induction on q that:

$$\mathrm{zip}_n(\sigma_0, \cdots, \sigma_{n-1}) \;!\; q \times n + r \;\rightsquigarrow^*\; \sigma_r \;!\; q \qquad \text{when } r < n$$

Base case:

$$\begin{aligned} & \mathrm{zip}_n(\sigma_0, \cdots \sigma_{n-1}) \;!\; 0 \times n + r \\ \rightsquigarrow\; & \mathrm{head}\,\sigma_0 :: \cdots :: \mathrm{head}\,\sigma_{n-1} :: \mathrm{zip}_n(\mathrm{tail}\,\sigma_0, \cdots, \mathrm{tail}\,\sigma_{n-1}) \;!\; r \\ \rightsquigarrow^*\; & \sigma_r \;!\; 0 \qquad \text{(by Lemma 3.2.10)} \end{aligned}$$

Assuming the property true for $q \in \mathbb{N}$, we show it for $q + 1$:

$$\begin{aligned} & \mathrm{zip}_n(\sigma_0, \cdots, \sigma_{n-1}) \;!\; (q+1) \times n + r \\ \rightsquigarrow^*\; & \mathrm{zip}_n(\mathrm{tail}\,\sigma_0, \cdots, \mathrm{tail}\,\sigma_{n-1}) \;!\; q \times n + r \qquad \text{(by Lemma 3.2.10)} \\ \rightsquigarrow^*\; & \mathrm{tail}\,\sigma_r \;!\; q \qquad \text{(by induction hypothesis)} \\ \rightsquigarrow\; & \sigma_r \;!\; q + 1 \end{aligned}$$

□

Projection

We define a polymorphic stream function (PSF) to extracts the first stream argument from an interleaving of streams. It satisfy, for $n > 0$ and all variables universally quantified:

$$\mathrm{proj}_{n,\mathbb{D}}(\mathrm{zip}_{n,\mathbb{D}}(s_0, \cdots, s_{n-1})) = s_0$$

The following pure stream equation uniquely defines a PSF having the previous property. For $n > 0$,

$$\mathrm{proj}_n\, s = \mathrm{head}\, s :: \mathrm{proj}_n(\mathrm{tail}^n\, s) \tag{3.2.1}$$

Where tail^n is n compositions of tail. The recursive call is guarded so the function is productive. A proof by induction on x shows that:

$$\overline{\mathrm{proj}_n}\, x = n \times x \tag{3.2.2}$$

When $n = 1$, the interleaving and projection functions are the identity, $\mathrm{zip}_1 = \mathrm{proj}_1 = \mathrm{id}_{\mathbb{S}}$. When $n = 0$, the equation (3.2.1) for proj_n still makes sense if we set $\mathrm{tail}^0 = \mathrm{id}_{\mathbb{S}}$, it defines a function which outputs a constant stream consisting of the first element of its input:

$$\mathrm{const}\, s = \mathrm{head}\, s :: \mathrm{const}\, s \;\; .$$

Its indexing function is the constant null: $\forall k \in \mathbb{N} \,.\, \overline{\mathrm{const}}(k) = 0$.

For convenience, we also define the projection for the other stream arguments:

For i < n, define $\quad \mathrm{proj}_{n,i}(s) \triangleq \mathrm{proj}_n(\mathrm{tail}^i(s))$

Its indexing function is $\quad \overline{\mathrm{proj}_{n,i}}(k) = n \times k + i$

proj and zip enjoy the two following properties:

$$\mathrm{proj}_{n,i,\mathbb{D}}(\mathrm{zip}_{n,\mathbb{D}}(s_0, \cdots, s_{n-1})) = s_i$$

$$s = \mathrm{zip}_{n,\mathbb{D}}\left((\mathrm{proj}_{n,i,\mathbb{D}}(s))_{i\in\{0,\cdots,n-1\}}\right)$$

Hanoi

We are now ready to return to Example 3.2.1. Recall that

$$\mathrm{hanoi}(s) = \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s))$$

We will prove that $\overline{\mathrm{hanoi}}(2^v(2m+1)) = v$ by induction on v. In the base case, we have

$$\begin{aligned} \mathrm{hanoi}(s) \ ! \ 2k+1 &\rightsquigarrow \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s)) \ ! \ 2k+1 \\ &\rightsquigarrow^* \mathrm{const}(s) \ ! \ k \\ &\rightsquigarrow^* s \ ! \ 0 \end{aligned}$$

This proves $\overline{\mathrm{hanoi}}(2k+1) = 0$. In the induction step, we have

$$\begin{aligned} \mathrm{hanoi}(s) \ ! \ 2^{v+1}(2m+1) &\rightsquigarrow \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s)) \ ! \ 2^{v+1}(2m+1) \\ &\rightsquigarrow^* \mathrm{hanoi}(\mathrm{tail}(s)) \ ! \ 2^v(2m+1) \\ &\rightsquigarrow^* \mathrm{tail}(s) \ ! \ v \\ &\rightsquigarrow s \ ! \ v+1 \end{aligned}$$

This proves $\overline{\mathrm{hanoi}}(2^v(2m+1)) = v$ implies $\overline{\mathrm{hanoi}}(2^{v+1}(2m+1)) = v+1$.

It is interesting to try and compute the element in the first stream position,

$$\mathrm{hanoi}(s) \ ! \ 0 \rightsquigarrow \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s)) \ ! \ 0 \rightsquigarrow^+ \mathrm{hanoi}(\mathrm{tail}(s)) \ ! \ 0,$$

leading to an infinite chain of reduction

$$\mathrm{hanoi}(s) \ ! \ 0 \rightsquigarrow^+ \mathrm{hanoi}(\mathrm{tail}(s)) \ ! \ 0 \rightsquigarrow^+ \mathrm{hanoi}(\mathrm{tail}^2(s)) \ ! \ 0 \rightsquigarrow^+ \cdots,$$

showing that $\overline{\mathrm{hanoi}}(0) = \bot$.

3.2.5. Zip-Proj Equations

Using the operations of interleaving and projection we can greatly simplify the study of PSES. We show that any system of mutually recursive equations over multi argument stream functions can be reduced to an equivalent single unary equation. We define a subset of pure stream equation systems and prove they are in bijection.

Definition 3.2.12. *A zip-proj equation consists of a single equation $\phi\, s = \sigma$ on a unary polymorphic stream function $\phi_{\mathbb{D}} : \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$. The right hand side of a zip-proj equation is called a zip-proj stream term and belongs to the language generated by* *the following grammar.*

$$\begin{array}{rcll}
\sigma & ::= & s & \textit{input stream} \\
& \mid & (\mathrm{tail}\,\sigma) & \textit{stream stripped of its first element} \\
& \mid & (\mathrm{head}\,\sigma :: \sigma') & \textit{first element of stream } \sigma \textit{ followed by } \sigma' \\
& \mid & (\mathrm{zip}_n\,\sigma_0 \cdots \sigma_{n-1}) & n > 0\textit{, interleaving of } n \textit{ streams} \\
& \mid & (\mathrm{proj}_n\,\sigma) & n \in \mathbb{N}\textit{, first projection of } n \textit{ interleaved streams} \\
& \mid & (\phi\,\sigma) & \textit{recursive call}
\end{array}$$

A zip-proj equation together with the equations for all the zip_n and proj_n called in its right hand side form a pure stream equation system. A zip-proj equation defines a polymorphic function and an indexing function corresponding to the semantics of the pure stream equation system.

Remark 3.2.13. *In a zip-proj equation the corecursion only happens in the definition of ϕ,* zip *and proj. There is no mutual recursion.*

Lemma 3.2.14. *Let $(f_{k,\mathbb{D}})_{k\in[1,n]}$ be the solutions of a system of pure stream equations, we can construct a zip-proj equation defining a function $\phi_{\mathbb{D}} : \mathbb{S}_{\mathbb{D}} \to \mathbb{S}_{\mathbb{D}}$ such that: $\forall k \in [1,n], \forall s, s_1, \cdots, s_p \in \mathbb{S}_{\mathbb{D}}$,*

$$f_{k,\mathbb{D}}(s_1, \cdots, s_{p_k}) = \mathrm{proj}_{n,k-1,\mathbb{D}}(\phi_{\mathbb{D}}(\mathrm{zip}_{p,\mathbb{D}}(s_1, \cdots, s_p)))$$
$$\phi_{\mathbb{D}}(s) = \mathrm{zip}_{n,\mathbb{D}}\left((f_{k,\mathbb{D}}(\mathrm{proj}_{p,0,\mathbb{D}}(s), \cdots, \mathrm{proj}_{p,p_k-1,\mathbb{D}}(s))_{k\in[1,n]}\right)$$

Proof. The construction uses interleaving and projections to gather multiple streams arguments into a single one and multiple functions into one. Given a system of n equations

$$f_k s_1 \cdots s_{p_k} = \sigma_k$$

of arity p_k for $k \in [1,n]$, let $p = \max\{p_k \mid k \in [1,n]\}$, we define a zip-proj equation

$$\phi\, s = \mathrm{zip}_n\, [\![\sigma_1]\!] \cdots [\![\sigma_n]\!]$$

as the interleaving of the right hand sides of $(f_k)_{k\in[1,n]}$ by recursively applying the following transformation on stream terms:

$$\begin{array}{rcll}
[\![s_i]\!] & = & \mathrm{proj}_p(\mathrm{tail}^{i-1}\, s) & i \in [1,p] \\
[\![f_k\,\sigma_1 \cdots \sigma_{p_k}]\!] & = & \mathrm{proj}_n(\mathrm{tail}^{k-1}(\phi(\mathrm{zip}_p\,[\![\sigma_1]\!] \cdots [\![\sigma_{p_k}]\!]\, s \cdots s))) & k \in [1,n]
\end{array}$$

Where s is repeated $(p - p_k)$ times in the call to zip_p. □

Corollary 3.2.15. *A pure stream equation is productive if and only if a corresponding zip-proj equation is productive.*

Thus, we can work in the simpler setting of zip-proj equations, without losing the generality of PSES.

Chapter 4.

PSES Productivity is Π^0_2-Complete

In this chapter we study problem of the productivity of pure stream equations systems (PSES). PSES define polymorphic functions and productivity is equivalent to totality of the indexing function (Corollary 3.2.8), so the problem is Π^0_2. But is it Π^0_2-hard? After all, the equations might not capture all polymorphic functions and the functions that they capture might not be so complicated that the problem is Π^0_2-hard. In fact it is: we found a set of indexing functions whose totality problem is undecidable and fully captured by PSES, i.e. there exist a PSES specifying the corresponding polymorphic function associated with the indexing function. Such a set is given by a generalisation of the Collatz problem.

A previous proof of the same result, relies on a different encoding of Collatz functions [EGH09a]. I discuss this article in §4.3.1.

Overview §4.1 gives some background on decision problems, computability, the arithmetical hierarchy and reduction. §4.2 defines the generalised Collatz problem. In §4.3 we reduce the problem of pure stream productivity (PPSE) to the generalised Collatz problem (GCP). In §4.4 we prove the Π^0_2 hardness of GCP, implying the Π^0_2 hardness of PPSE. This proof is once again a reduction from another slightly different generalisation of the Collatz problem whose arithmetic class was already known.

4.1. Terminology: Problems, Computability and Reductions

Minimal knowledge of computability is necessary to understand this chapter, the reader must know about the arithmetical hierarchy and many-one reductions. This section gives a short overview of that knowledge. For a textbook on computability see [Coo03].

Decision Problems

Decision problems are questions to which we can answer yes or no. Solving the problem is giving the set of its solutions. If there is an algorithm to decide whether an element is solution then the problem is decidable, unfortunately most interesting problems are undecidable. We often identify a problem with the set of its solutions.

Some problems we refer to in the chapter:

- The halting problem is to decide whether a Turing machine terminates on a given input (initial configuration).
- The totality problem is to decide whether a Turing machines terminates on all its inputs;
- PPSE which we consider in this chapter is to decide whether a pure stream equation system is productive.
- GCP is the generalised collatz problem defined in §4.2.2 is used to prove PPSE.

One purpose of computability theory is to classify sets. Usually, we consider subsets of natural numbers. We must use some encoding to transform a problem ranging over another domain.

For instance, Gödel encoding are based on the factorisation into prime divisors. Given n distinct prime numbers $p_1, \cdots, p_n$ we can encode tuples of natural numbers $x_1, \cdots, x_n$ as $\prod_{k=1}^{n} p_k^{x_k}$, this is called a Gödel encoding. We use such an encoding explicitly in §5.1.2.

Computability

We classify sets and relations on natural numbers according to their computability. A set is *primitive recursive* iff its characteristic function is primitive recursive. A set is *recursive* (synonyms: computable, decidable), iff its characteristic function is recursive. A set is *recursively enumerable* (synonyms: computably enumerable, semi-decidable) iff it is empty or the range of a total recursive function; equivalently iff it is the domain of a partial recursive function. A set is *co-recursively enumerable* iff its complement in $\mathbb{N}$ is recursively enumerable. An interesting result is that a set is recursive iff it is both recursively and co-recursively enumerable.

The Church-Turing thesis, widely accepted allows us to identify *computable* with *recursive.*

We may indistinguishably use the previous terminology to qualify sets, relations, problems and predicates which can easily be identified as the sets of the elements that satisfy them, encoding tuples of natural numbers if necessary.

Sets are further classified in the arithmetical hierarchy.

Arithmetical Hierarchy

First order formulas and the relations they define are classified in an *arithmetical hierarchy* according to their quantifier forms. We define by induction two classes of predicates: Σ^0_p and Π^0_p indexed by natural numbers. In the definition, a and x are tuples of natural numbers.

- $\Sigma^0_0 = \Pi^0_0$ are the primitive recursive relations.
- $\Sigma^0_{(p+1)}$ are the relations P satisfying $P(a) \iff \exists x \,.\, Q(a, x)$. where $Q \in \Pi^0_p$.
- $\Pi^0_{(p+1)}$ are the relations P satisfying $P(a) \iff \forall x \,.\, Q(a, x)$. where $Q \in \Sigma^0_p$.

We also define Δ^0_p as the intersection of Σ^0_p and Π^0_p.

The subscript corresponds to the number of alternations of unbounded quantifiers, and the class says which is the first: existential for Σ, universal for Π. The negation of a formula and the complement of a relation belong to the other class: $P \in \Pi^0_p \iff \neg P \in \Sigma^0_p$. Each class is strictly included in both of the next: $\Sigma^0_p \cup \Pi^0_p \subset \Delta^0_{p+1}$.

What is important is the smallest class in which a relation is.

- Σ^0_1 are the recursively enumerable relations;
- Π^0_1 are the co-recursively enumerable relations;
- Δ^0_1 are the recursive relations (decidable);
- Π^0_2 is the class of relations of the form $P(a) \iff \forall x\,.\,\exists y\,.\,R(a,x,y)$ where R is primitive recursive; and strictly extends Σ^0_1 and Π^0_1.

Membership, Hardness, Completeness

The halting problem is Σ^0_1 since given a Turing machine M and an initial configuration c, M terminates on c iff there exists a finite number k of state transitions from the starting state in configuration c to a final state.

The totality problem is Π^0_2 since it is equivalent to: for all initial configuration the machine terminates.

Note that those arguments only justify membership to a class. To get a precise characterisation, we need to show that the problem doesn't lie in any smaller class of the hierarchy. We say the problem is *strictly* in that class.

For instance if a problem is strictly Π^0_2, we cannot enumerate either the elements for which it is true nor those for which it is false, since it is neither recursively (Σ^0_1) nor co-recursively (Π^0_n) enumerable.

Reducibility

We can reduce a problem Q to a problem P by providing a function $f : Q \to P$ such that Q is equivalent to $P \circ f$. So that we can solve Q whenever we can solve P.

Definition 4.1.1 (Reducibility). *We say Q is many-one reducible to P ($Q \leq_m P$) iff there is is a total recursive function f such that $\forall x\,.\,Q(x) \iff P(f(x))$.*

We say f is a reduction from Q to P.

Property 4.1.2. *The many-one reduction is a preorder on subsets of the natural numbers.*

Property 4.1.3. *If C is one of Σ, Π, Δ, and $n \geq 1$, if Q is reducible to P then $P \in C_n \implies Q \in C_n$.*

Definition 4.1.4 (Hardness). *If all problems of a class C_n are reducible to a problem P, then we say P is C_n-hard.*

By transitivity of $\leq_m$, the following holds.

Property 4.1.5. *If $Q \leq_m P$ and Q is C_n-hard then so is P.*

We use that last property twice in this chapter: 1) to prove that GCP is Π^0_2-hard by reduction from KSP whose hardness is known from [KS07]. 2) to prove that PPSE is Π^0_2-hard by reduction from GCP.

Property 4.1.6. *If P is C_n-hard then P is not in any lower classes.*

Proof. Let Q be a relation whose lowest class is C_n, then it isn't in any of C_m, for $m < n$. By the contraposition of Property 4.1.3 P isn't in any of the C_m either. □

Definition 4.1.7 (Completeness)**.** *If P is both C_n-hard and in C_n, we say P is C_n-complete.*

By Property 4.1.6, a C_n-complete problem is also strictly in C_n, but the reverse is not necessary.

4.2. A Generalisation of the Collatz Problem

The Collatz problem, also called the $3n + 1$ conjecture, states that iterating the simple process of halving even numbers and adding one to the triple of odd numbers, always eventually reaches one. This problem can be generalised to functions that are linear on modulo classes.

In this section we define generalised Collatz functions and the generalised Collatz problem. A similar generalisation [KS07] was proven Π^0_2. In §4.4 we reduce their problem to ours to prove that it is also Π^0_2.

4.2.1. The Collatz Problem

Let $\mathbb{N}^+$ the set of strictly positive natural numbers. Consider the function $f : \mathbb{N}^+ \to \mathbb{N}^+$ defined by:

$$f\,x = \begin{cases} x/2 & \text{when } x \text{ is even } ; \\ 3 \times x + 1 & \text{when } x \text{ is odd } . \end{cases} \tag{4.2.1}$$

The Collatz problem is to know if the iterations of f converge to 1. If it does, then further iterations would yield a loop: $1, 4, 2, 1, 4, 2, 1,$ *etc.* We write f^k for f composed with itself k times:

$$f^0 = \mathrm{id}_{\mathbb{N}^+} \qquad\qquad f^{k+1} = f \circ f^k$$

Conjecture 4.2.1 (Collatz)**.** $\forall x > 0 \,.\, \exists k \in \mathbb{N} \,.\, f^k\,x = 1$

The conjecture is still unproven to this day despite extensive computer simulations and research [Lag06].

4.2.2. A Generalisation

We generalise the Collatz problem with the intent of reducing it to the productivity of pure stream equations.

Definition 4.2.2 (Collatz Functions)**.** *Let $f : \mathbb{N} \to \mathbb{N}$, and a natural $m > 0$, we say f is m-Collatz when it is linear on the equivalence classes modulo m.*

Therefore, a Collatz function is given by the data of m and for all $i < m$, two real numbers a_i and b_i (which are in fact necessarily rational), such that,

$$\forall x \in \mathbb{N} \,.\, x \equiv i \pmod{m} \implies f\,x = a_i \times x + b_i \ \ .$$

Problem 4.2.3 (GCP: Generalised Collatz Problem)**.** *GCP is the set of Collatz functions f such that every natural x is mapped to zero by an iteration of f.*

$$\mathrm{GCP}(f) \iff \forall x \in \mathbb{N} \,.\, \exists k \in \mathbb{N} \,.\, f^k\,x = 0 \ \ .$$

Note, that the problem takes as input a representation of a collatz function given by its modulus and coefficients. This can be encoded as a natural number, and we can view the problem as a subset of $\mathbb{N}$.

Theorem 4.2.4. *GCP is Π_2^0 complete.*

Proof. Given in §4.4 □

Example 4.2.5 (Original Collatz problem)**.** *Define g the 2-Collatz function given by $m = 2$, $a_0 = 3$, $b_0 = 3$, $a_1 = 1/2$, $b_1 = -1/2$. Then $\mathrm{GCP}(g)$ is equivalent to the original Collatz problem.*

Proof. Let f be the original Collatz function from (4.2.1).

If $x > 0$ is even, $\exists p \in \mathbb{N} \,.\, x = 2 \times p + 2$, and $f(x) = p + 1$, and $g(x-1) = a_1 \times (2 \times p + 1) + b_1 = p + \frac{1}{2} - \frac{1}{2} = p = f(x) - 1$.

If $x > 0$ is odd, $\exists p \in \mathbb{N} \,.\, x = 2 \times p + 1$, and $f(x) = 6 \times p + 4$, and $g(x-1) = a_0 \times (2 \times p) + b_0 = 6 \times p + 3 = f(x) - 1$.

We proved that $\forall x > 0 \,.\, g(x-1) = f(x) - 1$. By induction on n, it follows that $\forall x, n \in \mathbb{N} \,.\, g^n(x-1) = f^n(x) - 1$. Thus the convergence of iterations of g to zero is equivalent to the convergence of iterations of f to one on their respective domains. □

4.2.3. Properties of Collatz Functions

The remainder of the section is dedicated to proving properties of Collatz functions.

Lemma 4.2.6. *The coefficients a_i of a Collatz function are positive.*

Proof. Otherwise we could find a big enough x for which the value of f would be negative. □

For the reduction to pure stream equations, it is useful to define the Collatz functions only in terms of natural numbers. We will thus prove the following result.

Proposition 4.2.7. *The Collatz functions are exactly the functions $f : \mathbb{N} \to \mathbb{N}$ such that there exists a natural number $m > 0$, and for all natural numbers $i < m$, two natural numbers c_i and d_i, such that,*

$$\forall y \in \mathbb{N} . \forall i \in [0, m-1] . f\,(y \times m + i) = c_i \times y + d_i \ .$$

Proof. Let f be a Collatz function. We will express coefficients c_i and d_i of Proposition 4.2.7 in terms of the data m, a_i, b_i of Definition 4.4.1. Let $i \in [0, m-1]$, we define $c_i \triangleq a_i \times m$ and $d_i \triangleq a_i \times i + b_i$. Then, for $y \in \mathbb{N}$,

$$\begin{aligned} f\,(y \times m + i) &= a_i \times (m \times y + i) + b_i \\ &= [a_i \times m] \times y + (a_i \times i + b_i) \\ &= c_i \times y + d_i \end{aligned}$$

We need to show that c_i and d_i are natural numbers. For all $i \in [0, m-1]$, $d_i = f\,i \in \mathbb{N}$ and $c_i = f\,(m+i) - d_i \in \mathbb{Z}$. This establishes that c_i is an integer, but it is positive because $c_i = a_i \times m$ and $a_i \geq 0$ (Lemma 4.2.6).

Reciprocally, every function with m, c_i, d_i as in Proposition 4.2.7 is a Collatz function by taking $a_i \triangleq c_i/m$ and $b_i \triangleq d_i - c_i \times i/m$. □

Corollary 4.2.8. *The coefficients a_i, b_i of a Collatz function are rational numbers.*

Proof. $a_i = c_i/m$ and $b_i = d_i - c_i \times i/m$, with c_i, m, d_i, $i \in \mathbb{N}$. □

Example 4.2.9. *The function g of Example 4.2.5 corresponding to the original Collatz problem is defined with the coefficients: $m = 2$, $c_0 = 6$, $d_0 = 3$, $c_1 = 1$, $d_1 = 0$. For all x, $g(x \times 2) = 6 \times x + 3$ and $g(x \times 2 + 1) = x$. Remember that function f of (4.2.1) is translated. For $x > 1$,*

$$\begin{aligned} f(x) &= g(x-1) + 1 \\ &= \begin{cases} (6 \times y + 3) + 1 & \textit{if } x - 1 = y \times 2 \\ y + 1 & \textit{if } x - 1 = y \times 2 + 1 \end{cases} \\ &= \begin{cases} 3 \times x + 1 & \textit{if } x = (y+1) \times 2 - 1 \\ x/2 & \textit{if } x = (y+1) \times 2 \end{cases} \end{aligned}$$

4.3. Generalised Collatz is Reduced to PSES Productivity

The problem of pure stream productivity, noted PPSE is precisely stated.

Problem 4.3.1 (PPSE: Productivity of Pure Stream Equations)**.** PPSE *is the set of systems of pure stream equations whose least solution consists of productive polymorphic stream functions.*

We will now describe how a stream equation $\phi = \sigma_h$ can be defined from a Collatz function h, so that solving GCP(h) is equivalent to solving PPSE($\phi = \sigma_h$).

The principle of this transformation is that collatz functions arise naturally as indexing functions of pure stream equations thanks to zip_m whose indexing function computes the quotient and remainder modulo m, and $\mathrm{proj}_{a,b}$ whose indexing function multiplies by a and adds b.

Example 4.3.2. *We give a zip-proj equation corresponding to the original Collatz function:*

$$\phi\, s = \mathrm{head}\, s :: \mathrm{tail}\left(\mathrm{zip}_2\left(\mathrm{proj}_{6,3}(\phi\, s),\ \mathrm{proj}_{1,0}(\phi\, s)\right)\right)$$

Note that the indices of the projections correspond to the coefficients c_i and d_i of the Collatz function g. Looking at the indexing relation: $\phi\, s\ !\ 0 \rightsquigarrow^* s\ !\ 0$ and for $x \geq 1$,

$$\begin{aligned}
\phi\, s\ !\ x &\rightsquigarrow^* \mathrm{tail}\left(\mathrm{zip}_2\left(\mathrm{proj}_{6,3}(\phi\, s),\ \mathrm{proj}_{1,0}(\phi\, s)\right)\right)\ !\ x-1\\
&\rightsquigarrow^* \mathrm{zip}_2\left(\mathrm{proj}_{6,3}(\phi\, s),\ \mathrm{proj}_{1,0}(\phi\, s)\right)\\
&\rightsquigarrow^* \begin{cases} \mathrm{proj}_{6,3}(\phi\, s)\ !\ y & \textit{if } x = 2\times y\\ \mathrm{proj}_{1,0}(\phi\, s)\ !\ y & \textit{if } x = 2\times y+1\end{cases}\\
&\rightsquigarrow^* \begin{cases} \phi\, s\ !\ 6\times y+3 & \textit{if } x = 2\times y\\ \phi\, s\ !\ y & \textit{if } x = 2\times y+1\end{cases}
\end{aligned}$$

This shows that $\overline{\phi}\, 0 = 0$ is well defined and when $x \geq 1$, $\overline{\phi}\, x = \overline{\phi}\,(g\, x)$ where g is the Collatz function for the original problem, as defined in Example 4.2.9. So $\overline{\phi}$ is total (and ϕ productive) iff g verifies the generalised Collatz problem, i.e. its iterations converge to 0 for any $x \geq 1$.

We now consider the general case.

Reduction Given a Collatz function h, we define a stream function ϕ_h with a pure stream equation such that the indexing function is total iff h has the Collatz property.

Lemma 4.3.3. *Let $m > 0$, for $i < m$, let $c_i, d_i \in \mathbb{N}$, defining the Collatz function h by*

$$\forall i < m, \forall y \in \mathbb{N}, \qquad h\,(y\times m + i) = c_i \times y + d_i$$

We define a zip-proj equation specifying a polymorphic function ϕ_h:

$$\phi_h\, s = \mathrm{head}\, s :: \mathrm{tail}\left(\mathrm{zip}_m((\mathrm{proj}_{c_i,d_i}(\phi_h\, s))_{i\in[0,m-1]})\right)$$

The indexing function of ϕ_h satisfies:

$$\overline{\phi_h}\, x = \begin{cases} 0 & \textit{if } x = 0\ ;\\ \overline{\phi_h}\,(h\, x) & \textit{otherwise} \end{cases}.$$

Proof. First, $\phi_h\, s \; ! \; 0 \rightsquigarrow s \; ! \; 0$. Then, for $x \geq 1$,

$$\begin{aligned}
\phi_h\, s \; ! \; x \rightsquigarrow \;& \mathrm{tail}\left(\mathrm{zip}_m((\mathrm{proj}_{c_i,d_i}(\phi_h\, s))_{i\in[0,m-1]})\right) \; ! \; x-1 \\
\rightsquigarrow^* \;& \mathrm{zip}_m((\mathrm{proj}_{c_i,d_i}(\phi_h\, s))_{i\in[0,m-1]}) \; ! \; x \\
\rightsquigarrow^* \;& \mathrm{proj}_{c_j,d_j}(\phi_h\, s) \; ! \; y \qquad \text{where } y \triangleq \lfloor x/m \rfloor \text{ and } j \triangleq x \bmod m \\
\rightsquigarrow^* \;& \phi_h\, s \; ! \; c_j \times y + d_j \\
= \;& \phi_h\, s \; ! \; h\, x
\end{aligned}$$ □

Equivalence We show that the equation is productive if and only if h has the Collatz property.

Property 4.3.4. $\mathrm{PPSE}(\phi_h) \iff \mathrm{GCP}(h)$.

Proof. 1. $\mathrm{GCP}(h) \implies \mathrm{PPSE}(\phi_h)$.
Assume $\mathrm{GCP}(h)$ holds, then we can define a function $k : \mathbb{N} \to \mathbb{N}$ which for all x yields the smallest i such that $h^i\, x = 0$. Then $\overline{\phi_h}$ is constant null because $\forall x \geq 1 \,.\, \overline{\phi_h}\, x = \overline{\phi_h}\,(h^{k(x)} x) = \overline{\phi_h}\, 0 = 0$. So $\overline{\phi_h}$ is total and $\mathrm{PPSE}(\phi_h)$ follows from Corollary 3.2.8.

2. $\neg\,\mathrm{GCP}(h) \implies \neg\,\mathrm{PPSE}(\phi_h)$.
If $\mathrm{GCP}(h)$ doesn't hold, then there is $x \in \mathbb{N}$ such that $\forall k \in \mathbb{N} \,.\, h^k\, x > 0$. And there is an infinite chain of indexing reductions from $\phi_h s \; ! \; x$ showing that $\overline{\phi_h} x = \bot$. Since the indexing function is not total, the stream equation is not productive. □

The Main Result

Theorem 4.3.5. PPSE *is* Π^0_2 *complete*

Proof. 1. $\mathrm{PPSE} \in \Pi^0_2$ because productivity of a PSE is equivalent to totality of its indexing function (Corollary 3.2.8).

2. The Π^0_2 hardness of GCP is established in §4.4. The reduction of §4.3 from GCP to PPSE shows that PPSE is Π^0_2 hard. □

4.3.1. Comparison with Other Proofs of Undecidability

Complexity of FracTran and Productivity

The proof [EGH09a] has a very similar flavor. It is a reduction from the uniform halting problem (UHP) for FracTran programs, which in addition they had to prove by reduction from UHP for Turing machines. The similarity of their transformation is explained by the fact that each computation step of a FracTran program is realised by iterating a Collatz function. The reciprocal being false: not every Collatz function is the step function of a FracTran program. For a subset (of so called non-immortal) FracTran programs, UHP is equivalent to GCP of its step function, thus giving a reduction from FracTran UHP to GCP as another proof of Π^0_2-hardness of GCP.

Stream equality

Roşu [Roş06] established that the problem of proving two streams equal is Π^0_2 complete. The two streams are defined using a finite number of reduction rules allowing pattern matching on the elements of the stream.

Reducing the Totality Problem to Stream Equality To prove Π^0_2-hardness of the problem, Roşu encodes a Turing machine M as a set mutually recursive functions $q_i : \mathbb{S} \times \mathbb{S} \to \mathbb{B}$ for each state q_i of the machine M. $\mathbb{B}$ is the type of bits and $\mathbb{S}$ the type of bit-streams. The arguments of q_i represent the current configuration of the tape and the current position of the scanner along it, the infinite tape being split in two halves each one represented by a bit-stream. The functions are defined by pattern matching on the head of the right stream, corresponding to the bit being scanned. Depending on this value, the function q_i may call another function q_j, reflecting a state change, with the two stream arguments reflecting the changes on the tape. A halting state will in addition have a clause $q_h\langle L, R\rangle = 1$ to terminate the computation. Let q_s be the function encoding the starting state, and $\text{zeros} = 0 :: \text{zeros}$ the constant stream of 0. It is equivalent for the machine M to halt on input $b_1 b_2 \cdots b_n$ and for the term $q_s\langle \text{zeros}, b_1 :: b_2 :: \cdots :: b_n :: \text{zeros}\rangle$ to reduce to 1. Roşu then proceeds to define a stream function $t : \mathbb{S} \to \mathbb{S}$,

$$t\,(R) = q_s(\text{zeros}, R) :: t\,(1 :: R) \ .$$

Although the recursive call is guarded, each element $q_s(\text{zeros}, R)$ of the result stream may be undefined because q_s is partial.

Then, the stream equality $t\,(0 :: 1 :: \cdots :: 1 :: \text{zeros}) = \text{ones}$ (with k ones in the argument of t) holds if and only if the Turing machine M halts on input $1^j 0 1^k$ for all $j \geq 0$. Solving this equation corresponds to solving the totality problem when we take M a universal Turing machine that computes $f_k(j)$ on input $1^j 0 1^k$, where f_k is the partial function computed by the Turing machine of Gödel number k.

Reducing PPSE to Streams Equality Pure stream equations are a subset of the Roşu's equation systems. It seemed uncertain whether PPSE was Π^0_2 because the restriction of polymorphism prevented us from testing elements as Roşu did to simulate Turing machines. Having established that PPSE is Π^0_2, we can derive a new proof of Roşu's result by reducing PPSE to stream equality. The pure stream equations $\phi\, s_1 \cdots s_n = \sigma$ are already in the form of Roşu's equation systems, and the equality on streams $\phi\ \text{ones} \cdots \text{ones} = \text{ones}$ is equivalent to ϕ being productive, as we hinted in Property 3.2.9.

4.4. The Generalised Collatz Problem is Π^0_2-Complete

In §4.3, we showed that the problem of solving pure stream equations was Π^0_2, assuming that the generalised Collatz problem was Π^0_2. We now prove this assumption.

Kurtz and Simon made a slightly different generalisation of the Collatz problem and proved it Π_2^0 [KS07]. by a reduction from FracTran, building on a previous result by Conway [Con72]. Their formulation of the problem doesn't lend itself naturally to a reduction to the pure stream equation problem (PSE), but a very close formulation does.

We call KSP the generalised Collatz problem as formulated by Kurtz and Simon, and GCP with our formulation. We give a reduction from KSP to GCP.

4.4.1. Kurtz and Simon's Generalisation

The Collatz problem and our generalisation was presented in §4.2. We refer the reader to this section for the definitions. KSP is defined in terms of a subset of the Collatz functions:

Definition 4.4.1 (Collatz Functions with Positive Coefficients). *We call $\mathcal{C}$ the set of Collatz functions, and $\mathcal{C}^+$ the set of Collatz functions whose coefficients b_i are all positive.*

Problem 4.4.2 (KSP: Generalised Collatz Problem of Kurtz and Simon). KSP *is the set of Collatz functions c such that every non-null natural x is mapped to one by iterating c.*

$$\mathrm{KSP} = \{c \in \mathcal{C}^+ \mid \forall x \in \mathbb{N}^+ . \exists k \in \mathbb{N} . c^k\, x = 1\} \ .$$

KSP is slightly different from GCP because it is stated in terms of a convergence towards 1 (*resp.* 0) from strictly positive arguments (*resp.* positive), and the function is in $\mathcal{C}^+$ (*resp.* $\mathcal{C}$).

Example 4.4.3 (Original Collatz Problem). *The function f of the original problem is a Collatz functions according to Definition 4.4.1. It is given by $m = 2$, $a_0 = 1/2$, $b_0 = 0$, $a_1 = 3$, $b_1 = 1$. All the coefficients are positive so* KSP(f) *is equivalent to the original Collatz problem.*

Justifying the choice of GCP rather than KSP Seeing how Collatz functions happen naturally as indexing functions of stream equations, we turned towards the KSP problem that was already proven Π_2^0. However, their formulation introduces a difficulty in the reduction that GCP removes, namely convergence from 0 is not tested. Trouble follow when a non null number is mapped to zero. To convince himself of the difficulties, the reader is invited to try and reduce KSP to PPSE. The difficulty vanishes if we only consider Collatz functions whose restriction to $\mathbb{N}^+$ is $\mathbb{N}^+$. This comes down to imposing that all $i < m$, either a_i or b_i is not null. In fact, by a trivial bijection, GCP is equivalent to KSP$^+$ which is the restriction of KSP to those functions. Unfortunately, restricting the domain of a problem may well change its complexity, so we must proceed now to prove that the complexity is the same.

4.4.2. Reduction from KSP to GCP

We show that the problem KSP is contained in the problem GCP by constructing a reduction.

Definition 4.4.4. *We define a transformation (notation overline)* $\bar{\cdot} : \mathcal{C}^+ \to \mathcal{C}$*. Let* $c \in \mathcal{C}^+$*, given by* m*,* a_i *and* b_i *as in Definition 4.4.1. We define* $\bar{m}$*,* $\bar{a}_i$ *and* $\bar{b}_i$ *for* $\bar{c} \in \mathcal{C}$*. Set* $\bar{m} \triangleq m$ *and for* $i \in [0, m-1]$*, let* $j \in [0, m-1]$ *with* $j \equiv i + 1 \bmod m$*;*

1. *if both* a_j *and* b_j *are null,*
   a) *if* b_0 *is null, set* $\bar{a}_i \triangleq 1$ *and* $\bar{b}_i \triangleq 0$*;*
   b) *if* b_0 *is non-null, set* $\bar{a}_i \triangleq 0$ *and* $\bar{b}_i \triangleq b_0 - 1$*;*
2. *otherwise, (*a_j *or* b_j *is non-null), set* $\bar{a}_i \triangleq a_j$ *and* $\bar{b}_i \triangleq a_j + b_j - 1$*.*

Lemma 4.4.5. *Let* $c \in \mathcal{C}^+$ *and* $x \in \mathbb{N}$*,*

$$\bar{c}\,x = \begin{cases} x & \textit{if } c(x+1) = 0 \textit{ and } c\,0 = 0 \ ; \\ c\,0 - 1 & \textit{if } c(x+1) = 0 \textit{ and } c\,0 \neq 0 \ ; \\ c\,(x+1) - 1 & \textit{if } c(x+1) \neq 0 \ . \end{cases}$$

Proof. The lemma follows directly from the definition if we write down the following properties. If $x \equiv i \bmod m$ then $x + 1 \equiv j \bmod m$ iff $i + 1 = j \bmod m$. By Definition 4.4.1, $\bar{c}\,x = a_i \times x + b_i$; $c\,0 = b_0$; and $c\,(x+1) = a_j \times x + a_j + b_j$. Thus, $c(x+1) = 0$ iff $a_j = 0$ and $b_j = 0$. □

Lemma 4.4.6. *Let* $c \in \mathcal{C}^+$ *such that either* KSP(c) *or* GCP$(\bar{c})$ *holds. If* c *is null in zero, then it is non-null for all* x *strictly greater than 1.*

Proof. We prove the contrapositive statement. Let $c \in \mathcal{C}^+$ and $x > 1$ with $c\,0 = 0$ and $c\,x = 0$. Then,

1. $\neg$KSP(c) because $c^0\,x = x \neq 1$ and if $k > 0$, $c^k\,x = 0 \neq 1$.
2. $\neg\,$GCP$(\bar{c})$ because $\bar{c}\,(x-1) = x - 1$, so $\forall k \,.\, \bar{c}^k\,(x-1) = x - 1 \neq 0$. □

Lemma 4.4.7. *If* $c \in \mathcal{C}^+$ *with* $c\,0 \neq 0$ *and* $x \in \mathbb{N}$*, then for all* $j \in \mathbb{N}$*, there is a* $k \in \mathbb{N}$*, such that* $\bar{c}^j\,x = c^k\,(x+1) - 1$ *.*

Proof. Induction on j. The base case trivially holds. Let $j \,.\, k \in \mathbb{N}$, with $\bar{c}^j\,x = c^k\,(x+1) - 1$.

$$\bar{c}^{j+1}\,x = \bar{c}(c^k\,(x+1) - 1) = \begin{cases} c\,0 - 1 = c^{k+2}(x+1) - 1 & \text{if } c^{k+1}\,(x+1) = 0 \ ; \\ c^{k+1}(x+1) - 1 & \text{otherwise} \ . \end{cases}$$

□

Lemma 4.4.8. *If* $c \in \mathcal{C}^+$ *with* $c\,0 \neq 0$ *and* $x \in \mathbb{N}$*, then for all* $k \in \mathbb{N}$ *such that* $c^k\,(x+1) \neq 0$*, there is a* $j \in \mathbb{N}$*, such that* $\bar{c}^j\,x = c^k\,(x+1) - 1$ *.*

Proof. The proof is a course of value induction on k. Let $c \in \mathcal{C}^+$ with $c\,0 \neq 0$ and let $x \in \mathbb{N}$.

1. When $k = 0$, choose $j = 0$, then $\bar{c}^j\, x = x = (x+1) - 1 = c^k\,(x+1) - 1$.
2. Assume the property holds for all $n \leq k$. Assume $c^{k+1}\,(x+1) \neq 0$.
   a) If $c^k\,(x+1) = 0$, then $k \neq 0$ (because $x + 1 \neq 0$) and $c^{k-1}\,(x+1) \neq 0$ (because $c\,0 \neq 0$). By induction hypothesis, there is a j such that $\bar{c}^j\, x = c^{k-1}\,(x+1) - 1$. So $\bar{c}^{j+1}\, x = \bar{c}\,(c^{k-1}\,(x+1) - 1) = c\,0 - 1$ (by Lemma 4.4.5.b) $= c^{k+1}\,(x+1) - 1$.
   b) If $c^k\,(x+1) \neq 0$ then there is a j such that $\bar{c}^j\, x = c^k\,(x+1) - 1$ (induction hypothesis). $\bar{c}^{j+1}\, x = \bar{c}\,(\bar{c}^j\, x) = c\,(1 + \bar{c}^j\, x) - 1$ (by Lemma 4.4.5.c) $= c\,(c^k\,(x+1)) - 1 = c^{k+1}\,(x+1) - 1$.

   This concludes the inductive case. □

Lemma 4.4.9. *If $c \in \mathcal{C}^+$ and A is a part of $\mathbb{N}^+$ on which c is non-null; let $x \in A$ and $k \in \mathbb{N}$,*

$$[\forall j < k \,.\, c^j\, x \in A] \implies \bar{c}^k\,(x-1) = c^k\, x - 1 \ .$$

Proof. This is an induction on k. The base case $k = 0$ is trivial, with the right hand side reducing to $x - 1 = x - 1$. Assume the property holds for $k \in \mathbb{N}$. If we have $\forall j < k+1 \,.\, c^j\, x \in A$, then in particular $c^k\, x \in A$, so $c^{k+1}\, x \neq 0$ (c is non-null on A by hypothesis) thus $\bar{c}\,(c^k\, x - 1) = c^{k+1}\, x - 1$ (by Lemma 4.4.5.c). By induction hypothesis $\bar{c}^k\,(x-1) = c^k\, x - 1$, therefore $\bar{c}^{k+1}(x-1) = c^{k+1}\, x$, which concludes the inductive case. □

Lemma 4.4.10. $(\forall c \in \mathcal{C}^+)\quad \mathrm{KSP}(c) \implies \mathrm{GCP}(\bar{c})$

Proof. Let $c \in \mathcal{C}^+$ and assume $\mathrm{KSP}(c)$ holds.

1. If $c\,0 \neq 0$. Let $x \in \mathbb{N}$. By $\mathrm{KSP}(c)$, there is a k such that $c^k\,(x+1) = 1$. By Lemma 4.4.8, there is j such that $\bar{c}^j\, x = c^k\,(x+1) - 1 = 0$. This is true for all x, so $\mathrm{GCP}(\bar{c})$ holds.
2. If $c\,0 = 0$ then by Lemma 4.4.6, $\forall x > 1 \,.\, c\,x \neq 0$. We can thus apply Lemma 4.4.9 with $A = \{x \in \mathbb{N} \mid x > 1\}$. Let $x \in \mathbb{N}$; if $x > 0$, then $x + 1 \in A$ and there is k such that $c^k\,(x+1) = 1$ (by $\mathrm{KSP}(c)$), so $\bar{c}^k\, x = 0$ (by Lemma 4.4.9); if $x = 0$, the requested property holds trivially with $k = 0$ ($\bar{c}^k\, x = x = 0$), therefore $\mathrm{GCP}(\bar{c})$ holds. □

Lemma 4.4.11. $(\forall c \in \mathcal{C}^+)\quad \mathrm{GCP}(\bar{c}) \implies \mathrm{KSP}(c)$

Proof. Let $c \in \mathcal{C}^+$ and assume $\mathrm{GCP}(\bar{c})$ holds.

1. If $c\,0 \neq 0$ Let $x \in \mathbb{N}^+$. By $\mathrm{GCP}(\bar{c})$, there is a j such that $\bar{c}^j\,(x-1) = 0$. By Lemma 4.4.7, there is k such that $\bar{c}^j\,(x-1) = c^k\, x - 1$, so $c^k\, x = 1$. This is true for all x, so $\mathrm{KSP}(c)$ holds.
2. If $c\,0 = 0$ then by Lemma 4.4.6, $\forall x > 1 \,.\, c\,x \neq 0$. We can thus apply Lemma 4.4.9 with $A = \{x \in \mathbb{N} \mid x > 1\}$. Let $x \in \mathbb{N}^+$; if $x > 1$, and there is k such that $\bar{c}^k\,(x-1) = 0$ (by $\mathrm{GCP}(\bar{c})$), so $c^k\, x = 1$ (by Lemma 4.4.9); if $x = 1$, the requested property holds trivially with $k = 0$ ($\bar{c}^k\, x = x = 1$), therefore $\mathrm{KSP}(c)$ holds. □

Theorem 4.4.12. *The generalised Collatz problem* GCP *is* Π^0_2 *complete.*

Proof.

1. GCP is Π^0_2 hard. The function $c \mapsto \bar{c}$ reduces the problem KSP (Π^0_2 hard [KS07]) to GCP (by Lemmas 4.4.10 and 4.4.11).
2. GCP $\in \Pi^0_2$. The predicate $\text{GCP} = \{c \in \mathcal{C} \mid \forall x \in \mathbb{N}\,.\, \exists k \in \mathbb{N}\,.\, c^k\, x = 0\}$ is Π^0_2 because the predicate $R_c = \{(k, x) \in \mathbb{N}^2 \mid c^k(x) = 0\}$ is primitive recursive, since c can be computed by a primitive recursive function. □

4.4.3. Discussion

Even though the original Collatz problem isn't expressed verbatim with the formulation given by GCP, we think it's a minor detail, and a matter of starting counting from zero or one.

In fact, in the proof [KS07], the transformation from a counter machine to a Collatz function only produces functions f such that $x > 0 \implies f x > 0$. As a consequence, their proof without any modification shows the undecidability of KSP$^+$ which is the set of functions $f : \mathbb{N}^+ \to \mathbb{N}^+$ such that an extension in zero is a Collatz function f_0 and KSP(f_0) hold.

The reduction from KSP$^+$ to GCP is trivial, because endo-functions f of $\mathbb{N}^+$ are in bijection with endo-functions g of $\mathbb{N}$ by $\forall x \in \mathbb{N}\,.\, f\,(x+1) = g\,x + 1$, it follows, $\forall x, k \in \mathbb{N}\,.\, f^k\,(x+1) = g^k\,x + 1$, and trivially, KSP$^+(f) \iff$ GCP(g).

Chapter 5.

Expressivity of PSES

The content of this chapter is mostly contained in an article [SB12] written in collaboration with Christian Sattler.

Pure stream equation systems (PSES) define polymorphic stream functions. In the present chapter we ask the question: can they define *all* polymorphic stream functions? Definability implies Π_2^0-completeness of productivity of PSES, indeed, productivity is equivalent to totality of the indexing function (Corollary 3.2.8), this is a stronger result than Chapter 4's for which it was enough to find an undecidable set of indexing functions that was representable as PSES. Here we say all of them are representable.

Since polymorphic functions are completely characterised by their indexing function (see §3.1.2) which are partial computable functions from $\mathbb{N}$ to $\mathbb{N}$, we show that the indexing functions of PSES cover all the computable functions; meaning that PSES form another Turing complete computation model.

Overview In §5.1 we construct a PSES from a counter machines implementing the corresponding indexing function. In §5.2 we strenghten the previous result by giving a more intricate encoding using a strict subset of PSES consisting of only unary equations.

5.1. PSE Definability

In Chapter 4, the reduction from GCP relies on the possibility to represent every Collatz function as the indexing function of a PSES. Collatz functions are the step function of a Turing complete language called FracTran [KS07, EGH09a]. The indexing function of the PSES that we exhibited in our reduction was iterating this step function indefinitely for every $x \in \mathbb{N}^+$, thus simulating the computation of the FracTran program. However, the result of this computation was lost because we iterated the step function until the value 1 was reached meaning termination, but FracTran programs return the value just before. Another small difficulty before proving definability, is that FracTran programs encode their inputs and outputs as powers of 2 and 3. FracTran computational model being very similar to counter machines, we resolved to use the latter for our encoding; in fact we use the same method of encoding counters as powers of prime numbers as in the reduction from counter machine to FracTran [KS07].

We give a transformation from counter machines to PSES such that the indexing function of the PSES is equal to the function computed by the counter machine, thus proving that every polymorphic stream function is definable as a PSES.

5.1.1. Counter Machines

Definition 5.1.1. *A counter machine [Min67] is given by $\langle N, L, I\rangle$ where*

N *is the number of registers;*

L *the length of the program;*

I *is the program, consisting of L instructions $I_1 \cdots I_L$ which are either:*

- $\mathrm{inc}(r)$*,* $r \in [1, N]$*, for incrementing the value of register* r*;*
- $\mathrm{jzdec}(r, i)$*,* $r \in [1, N]$*,* $i \in [1, L]$*, for testing the value of register r, decrementing it if positive, or branching to instruction i if null.*

Execution The machine is started with the first instruction in a state where all registers are null except for the first which is initialised with the input value. Unless a branching occurs, the instructions are executed sequentially. The program terminates when sequentially leaving the last instruction. The value computed by the machine is the value of the last register.

Formally, we describe the execution of the counter machine in terms of its state: $\langle \rho, i\rangle$ which is the content of the registers $\rho : [1, N] \to \mathbb{N}$ and the current instruction number $i \in [1, L + 1]$. Note that there is no instruction $L + 1$ in the program, it corresponds to an implicit $I_{L+1} = \mathrm{halt}$ instruction. We write $\rho[r \leftarrow x]$ for the function equal to ρ everywhere except on r which is mapped to x.

We define a functional relation on states $\langle \rho, i\rangle \to \langle \rho', i'\rangle$ such that we reach instruction i' with register values ρ' after executing instruction i with register values ρ. It is the smallest relation verifying, for all $i, j \in [1, L]$, $r \in [1, N]$ and $\rho : [1, N] \to \mathbb{N}$:

- If $I_i = \mathrm{inc}(r)$ then $\langle \rho, i\rangle \to \langle \rho[r \leftarrow \rho(r) + 1], i + 1\rangle$.
- If $I_i = \mathrm{jzdec}(r, j)$ then $\rho(r) > 0 \implies \langle \rho, i\rangle \to \langle \rho[r \leftarrow \rho(r) - 1], i + 1\rangle$ and $\rho(r) = 0 \implies \langle \rho, i\rangle \to \langle \rho, j\rangle$.

Lemma 5.1.2. *For all $\rho : [1, N] \to \mathbb{N}$, $i \in [1, L]$, there is a unique successor state $\langle \rho, i\rangle \to \langle \rho', i'\rangle$.*

We write $\to^\star$ for the reflexive and transitive closure of this relation. We write $s \downarrow y$ when $s \to^\star \langle \rho, L + 1\rangle$ and $\rho(N) = y$. If such a y exists it is unique, a consequence of Lemma 5.1.2. If it doesn't, we write $s \uparrow$, and say that the machine diverges from state s.

A counter machine M computes a partial function $\phi_M : \mathbb{N} \to \mathbb{N}_\perp$ such that $\forall x, y \in \mathbb{N} \,.\, \phi_M(x) = y$ if $s_0 \downarrow y$ and $\phi_M(x) = \perp$ if $s_0 \uparrow$ where $s_0 = \langle \rho_x, 1\rangle$ is the initial state, ρ_x maps 1 to x and the rest of the registers to 0.

5.1.2. Construction of PSE Definability

Theorem 5.1.3. *Every polymorphic stream function can be defined by pure stream equations.*

Polymorphic stream functions are characterised by their indexing function. Given a counter machine M, we build a pure stream equation system with main function ψ such that its indexing function $\overline{\psi}$ is equal to the partial function ϕ_M computed by M. This entails Theorem 5.1.3 because counter machines are Turing Complete [Min67].

We use a Gödel encoding of the state of all registers. Let $P_1 \cdots P_N$ be the first N prime numbers. $\rho : [1, N] \to \mathbb{N}$ is represented by $\widehat{\rho} = \prod_{r\in[1,N]}(P_r)^{\rho(r)}$. In the proofs, we use the following property: $\widehat{\rho[r \leftarrow \rho(r) + 1]} = P_r \times \widehat{\rho}$.

The pure equations are defined for function symbols f_i, zip_{P_r} and proj_{P_r}, proj_0 and ψ (for $i \in [0, L + 1]$ and $r \in [1, N]$).

$$\psi\, s = f_0(f_1(s))$$
$$f_0\, s = \text{head}(\text{tail}\ s) :: f_0(\text{proj}_{P_1}\ s)$$

For $i \in [1, L]$,
$$f_i\, s = \begin{cases} \text{proj}_{P_r}(f_{i+1}\, s) & \text{if } I_i = \text{inc}(r); \\ \text{zip}_{P_r}(f_{i+1}\, s, \text{proj}_{P_r}(\text{tail}(f_j\, s)), & \text{if } I_i = \text{jzdec}(r, j). \\ \qquad \text{proj}_{P_r}(\text{tail}^2(f_j\, s)), & \\ \qquad \ldots, & \\ \qquad \text{proj}_{P_r}(\text{tail}^{P_r-1}(f_j\, s))) & \end{cases}$$

$$f_{L+1}\, s = \text{zip}_{P_N}(f_{L+1}(\text{tail}\, s), \text{proj}_0\ s, \cdots, \text{proj}_0\ s)$$
$$\text{zip}_{P_r}(s_0, \cdots, s_{P_r-1}) = \text{head}\, s_0 :: \cdots \text{head}\, s_{P_r-1} :: \text{zip}_{P_r}(\text{tail}\, s_0, \cdots, \text{tail}\, s_{P_r-1})$$
$$\text{proj}_{P_r}\ s = \text{head}\, s :: \text{tail}^{P_r}(\text{proj}_{P_r}\ s)$$
$$\text{proj}_0\ s = \text{head}\ s :: \text{proj}_0\ s$$

5.1.3. Proof

The stream functions f_i have been designed so that their indexing reduction follows the flow of execution of the program from instruction I_i according to the Gödel encoded registers. $\overline{f_{L+1}}$ extracts the result of the computation when the program ends: it has to get the value of register N from the Gödel encoding. $\overline{f_0}$ builds the initial values of the registers $\widehat{\rho_0} = P_1^x$ for the input x. The main function ψ composes f_0 and f_1 so that the computation may start with the proper state: initialised registers and first instruction.

Lemma 5.1.4. *A single execution step of the machine corresponds to a few steps of the indexing reduction. Let $i \in [1, L]$, $i' \in [1, L + 1]$, $\rho, \rho' : [1, N] \to \mathbb{N}$.*

$$\langle \rho, i\rangle \to \langle \rho', i'\rangle \implies f_i\, s\ !\ \widehat{\rho} \leadsto^+ f_{i'}\, s\ !\ \widehat{\rho'}$$

Proof. 1. If $I_i = \text{inc}(r)$ then $\langle \rho, i\rangle \to \langle \rho[r \leftarrow \rho(r) + 1], i + 1\rangle$ and

$$f_i\, s\ !\ \widehat{\rho} \ \leadsto\ \text{proj}_{P_r}(f_{i+1}\, s)\ !\ \widehat{\rho} \ \leadsto^*\ f_{i+1}\, s\ !\ P_r \times \widehat{\rho} \ =\ f_{i+1}\, s\ !\ \widehat{\rho[r \leftarrow \rho(r) + 1]}$$

2. If $I_i = \text{jzdec}(r, j)$ then
   a) If $\rho(r) > 0$, then $\langle \rho, i\rangle \to \langle \rho[r \leftarrow \rho(r) - 1], i + 1\rangle$. We use Lemma 3.2.11 with $\widehat{\rho} = \widehat{\rho[r \leftarrow \rho(r) - 1]} \times P_r + 0$. Thus,
   $$f_i\, s \ !\ \widehat{\rho} \ \rightsquigarrow^*\ f_{i+1}\, s \ !\ \widehat{\rho[r \leftarrow \rho(r) - 1]}$$
   b) If $\rho(r) = 0$, then $\langle \rho, i\rangle \to \langle \rho, j\rangle$ and $\widehat{\rho} = a \times P_r + b$ where a and b are quotient and reminder, with $b \neq 0$. Therefore,
   $$f_i\, s \ !\ \widehat{\rho} \ \rightsquigarrow\ \text{proj}_{P_r}(\text{tail}^b\,(f_j\, s)) \ !\ \widehat{\rho} \ \rightsquigarrow^*\ f_j\, s \ !\ b{+}P_r{\times}a \ =\ f_j\, s \ !\ \widehat{\rho} \quad \square$$

Lemma 5.1.5. *The indexing of f_{L+1} extract the result from the encoding of the registers.* $\langle \rho, L + 1\rangle \downarrow y \iff \overline{f_{L+1}}\, \widehat{\rho} = y$ *for all $\rho : [1, N] \to \mathbb{N}$, $y \in \mathbb{N}$.*

Proof. From the definition of $\downarrow$ it follows that $\langle \rho, L + 1\rangle \downarrow y$ iff $y = \rho(N)$. So we must show $f_{L+1}\, s \ !\ \widehat{\rho} \rightsquigarrow^* \rho(N)$. We proceed inductively on the value of $\rho(N)$. Let a and b be the quotient and reminder of $\widehat{\rho}$ by P_N.

- If $\rho(N) = 0$, then $b \neq 0$, so $f_{L+1}\, s \ !\ \widehat{\rho} \ \rightsquigarrow^*\ \text{proj}_0\ s \ !\ a \ \rightsquigarrow^*\ s \ !\ 0$.
- If $\rho(N) > 0$, then $b = 0$ and $a = \widehat{\rho[N \leftarrow \rho(N) - 1]}$;

$$\begin{aligned} f_{L+1}\, s \ !\ \widehat{\rho} &\rightsquigarrow^* f_{L+1}(\text{tail}\, s) \ !\ \widehat{\rho[N \leftarrow \rho(N) - 1]} \\ &\rightsquigarrow^* \text{tail}\, s \ !\ (\rho[N \leftarrow \rho(N) - 1])\, N \qquad \text{(by induction hypothesis)} \\ &= \text{tail}\, s \ !\ \rho(N) - 1 \\ &\rightsquigarrow s \ !\ \rho(N) \end{aligned}$$
$\square$

Lemma 5.1.6. *The indexing of f_0 builds the initial state of M. For all $x \in \mathbb{N}$, $\overline{f_0}\, x = \widehat{\rho_x}$, where ρ_x maps 1 to x and the rest of the registers to 0.*

Proof. The Gödel encoding of ρ_x is P_1 to the power x, P_1^x. An induction on x shows $f_0\, s \ !\ x \ \rightsquigarrow^*\ s \ !\ P_1^x$.

$$f_0\, s \ !\ 0 \ \rightsquigarrow\ \text{tail}\, s \ !\ 0 \ \rightsquigarrow\ s \ !\ 1 \ =\ s \ !\ P_1^0$$
$$f_0\, s \ !\ x + 1 \ \rightsquigarrow^*\ f_0(\text{proj}_{P_1}\ s) \ !\ x \ \rightsquigarrow^*\ \text{proj}_{P_1}\ s \ !\ P_1^x \ =\ s \ !\ P_1 \times P_1^x \ =\ s \ !\ P_1^{x+1} \quad \square$$

Lemma 5.1.7. *An execution sequence in M entails a longer indexing reduction. Let $i, i' \in [1, L + 1]$, $\rho, \rho' : [1, N] \to \mathbb{N}$, $n \in \mathbb{N}$,*
$$\langle \rho, i\rangle \to^n \langle \rho', i'\rangle \implies \exists m \geq n \,.\, f_i\, s \ !\ \widehat{\rho} \rightsquigarrow^m f_i'\, s \ !\ \widehat{\rho'} \ .$$

Proof. We proceed by induction on the length of the execution sequence.
If $n = 0$, the property holds with $m = 0$, by reflexivity.

If $n > 0$, we decompose the sequence in $\langle \rho, i\rangle \to \langle \rho', i'\rangle$ and $\langle \rho', i'\rangle \to^\star \langle \rho'', i''\rangle$. Conclusion follows from Lemma 5.1.4 and the induction hypothesis. □

Lemma 5.1.8. *When M is converging from a state $\langle \rho, i\rangle$, the corresponding indexing function $\overline{f_i}$ is defined on the encoding of ρ and computes the value returned by M. For all $i \in [1, L+1]$, $\rho : [1, N] \to \mathbb{N}$, $y \in \mathbb{N}$,*

$$\langle \rho, i\rangle \downarrow y \implies \overline{f_i}\, \widehat{\rho} = y \ .$$

Proof. Assume $\langle \rho, i\rangle \downarrow y$ then $\exists \rho_h . \langle \rho, i\rangle \to^\star \langle \rho_h, L+1\rangle$ and $\rho_h(N) = y$. By Lemma 5.1.7, then Lemma 5.1.5, $f_i\, s \ ! \ \widehat{\rho} \ \rightsquigarrow^* \ f_{L+1}\, s \ ! \ \widehat{\rho_h} \ \rightsquigarrow^* \ \rho_h(N)$. □

Lemma 5.1.9. *When M diverges from a state $\langle \rho, i\rangle$, the indexing function $\overline{f_i}$ is undefined for the encoding of ρ. For all $i \in [1, L]$, $\rho : [1, N] \to \mathbb{N}$, $y \in \mathbb{N}$,*

$$\langle \rho, i\rangle \uparrow \implies \overline{f_i}\, \widehat{\rho} = \bot \ .$$

Proof. If $\langle \rho, i\rangle \uparrow$ then there is an infinite execution sequence which never reaches a halting state $\langle \rho_h, L+1\rangle$. From Lemma 5.1.7, it follows that the indexing reduction sequence is also infinite and the indexing function is undefined. □

Lemma 5.1.10. $\langle \rho_x, 1\rangle \downarrow y \implies \overline{\psi}\, x = y$ *for all $x, y \in \mathbb{N}$.*

Proof. Follows from the conjunction of Lemma 5.1.6 and Lemma 5.1.8.

$$\begin{aligned} \psi\, s \ ! \ x \ &\rightsquigarrow \ f_0 (f_1 s) \ ! \ x \\ &\rightsquigarrow^* \ f_1 s \ ! \ \widehat{\rho_x} \qquad \text{(by Lemma 5.1.6)} \\ &\rightsquigarrow^* \ s \ ! \ y \qquad\quad \text{(by Lemma 5.1.8)} \end{aligned}$$

□

Lemma 5.1.11. $\langle \rho_x, 1\rangle \uparrow \implies \overline{\psi}\, x = \bot$ *for all $x \in \mathbb{N}$.*

Proof. Conjunction of Lemma 5.1.6 and Lemma 5.1.9. □

Lemma 5.1.12. $\overline{\psi} = \phi_M$

Proof. The conjunction of Lemma 5.1.10 and Lemma 5.1.11 gives the same characterisation of $\overline{\psi}$ as ϕ_M. In particular, the contrapositive of each lemma is the converse of the other one. □

5.2. Unary Definability

The previous construction from counter machines relied on interleaving (function zip) for implementing conditional execution: its indexing function does a case distinction on modulo classes. Noting that zip_p with p prime were the only stream function with more than one parameter in the construction, we considered the computational consequences of only allowing unary stream functions to be defined, we call such PSES *unary systems* (Definition 3.2.3). In fact, allowing interleaving is synonymous to allowing non-unary stream functions since we can use interleaving to merge any number of stream arguments into a single one, see §3.2.5.

What is the expressivity of unary systems, are there some unary polymorphic functions that cannot be defined by them? Can we characterise their indexing functions? Amazingly all polymorphic functions are definable with unary stream equations. In order to prove this more general definability result in the unary setting, entirely different techniques need to be developed to address conditional execution since we do not have zip available. Our solution is to separate conditional execution and unbounded looping into orthogonal concepts. Each one is studied in its own section before we give the construction.

5.2.1. Properties of some Indexing Functions

We prove two lemmas which are used later to establish the indexing functions of several stream equations.

Lemma 5.2.1. *Given $h \geq 1$ and a stream equation of the form:*

$$f(s) = \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: f(v(s)) \ ,$$

then the indexing function of f statisfies, for all $k \in \mathbb{N}$:

$$\overline{f}(k) = \overline{v}^{\lfloor k/h \rfloor}(a_{k \bmod h}) \ .$$

Proof. The proof is by induction on $k \in \mathbb{N}$. For $k < h$, we have

$$\begin{aligned} f(s) \ ! \ k &= \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: f(v(s)) \ ! \ k \\ &\rightsquigarrow^{k} \mathrm{tail}^{a_k}(s) \ ! \ 0 \rightsquigarrow^{a_k} s \ ! \ a_k, \end{aligned}$$

yielding $\overline{f}(k) = a_k = \overline{v}^0(a_k) = \overline{v}^{\lfloor k/h \rfloor}(a_{k \bmod h})$. For $k \geq h$, we have

$$\begin{aligned} f(s) \ ! \ k &= \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: f(v(s)) \ ! \ k \\ &\rightsquigarrow^{h} f(v(s)) \ ! \ k - h. \end{aligned}$$

By induction hypothesis, it follows that

$$\begin{aligned} \overline{f}(k) &= (\overline{f \circ v})(k-h) = \overline{v}(\overline{f}(k-h)) \\ &= \overline{v}(\overline{v}^{\lfloor (k-h)/h \rfloor}(a_{(k-h) \bmod h})) \\ &= \overline{v}(\overline{v}^{\lfloor k/h \rfloor - 1}(a_{k \bmod h})) = \overline{v}^{\lfloor k/h \rfloor}(a_{k \bmod h}), \end{aligned}$$

where we exploited contravariance of the indexing operation in the second step. □

Lemma 5.2.2. *Let $h \geq 1$ be given with a stream equation*

$$f(s) = \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: \mathrm{tail}^{h}(u(f(v(s)))).$$

Fix $k \in \mathbb{N}$ and choose $c(k) \in \mathbb{N}$ minimal such that $d(k) := \overline{u}^{c(k)}(k) \in \{\perp, 0, \ldots, h-1\}$. If such a $c(k)$ exists and $d(k) \neq \perp$, then $\overline{f}(k) = \overline{v}^{c(k)}(a_{d(k)})$, otherwise $\overline{f}(k) = \perp$.

Proof. The proof is by induction on $c(k)$ if it exists. At the base, $c(k) = 0$ is equivalent to $k < h$. In this case, $f(s) \mathbin{!} k \rightsquigarrow^k \mathrm{tail}^{a_k}(s) \mathbin{!} 0 \rightsquigarrow^{a_k} s \mathbin{!} a_k$, therefore $\overline{f}(k) = a_k = \overline{v}^{c(k)}(a_{d(k)})$.

Now assume $k \geq h$. Note that:

$$f(s) \mathbin{!} k \rightsquigarrow^h \mathrm{tail}^h(u(f(v(s)))) \mathbin{!} k - h \rightsquigarrow^h u(f(v(s))) \mathbin{!} k.$$

If $\overline{u}(k) = \perp$, then $c(k) = 1$, $d(k) = \perp$, and $\overline{f}(k) = \perp$. In the remainder, we will assume $\overline{u}(k) \neq \perp$. Then, $f(s) \mathbin{!} k \rightsquigarrow^+ f(v(s)) \mathbin{!} \overline{u}(k)$, and $\overline{f}(k) = \overline{v}(\overline{f}(\overline{u}(k)))$.

If $c(k)$ is defined, then $c(k) = c(\overline{u}(k)) + 1$ and we can apply the induction hypothesis: if $d(k) = d(\overline{u}(k)) \neq \perp$, then $\overline{v}(\overline{f}(\overline{u}(k))) = \overline{v}(\overline{v}^{c(\overline{u}(k))}a_{d(k)}) = \overline{v}^{c(k)}(a_{d(k)})$, otherwise $\overline{v}(\overline{f}(\overline{u}(k))) = \overline{v}(\perp) = \perp$.

If $c(k)$ is undefined, then so is $c(\overline{u}(k))$, and with a second induction we can construct an infinite sequence $f(s) \mathbin{!} k \rightsquigarrow^+ f(v(s)) \mathbin{!} \overline{u}(k) \rightsquigarrow^+ f(v^2(s)) \mathbin{!} \overline{u}^2(k) \rightsquigarrow^+ \ldots$ showing non-termination and $\overline{f}(k) = \perp$. □

The inquiring reader will notice that this lemma can be seen as a generalisation of Lemma 5.2.1 with u defined in a particular way, namely

$$u(s) = \underbrace{\mathrm{head}(s) :: \ldots :: \mathrm{head}(s)}_{h \text{ times}} :: s.$$

5.2.2. Collatz Functions and If-Programs

Collatz functions were defined in Definition 4.4.1.

Lemma 5.2.3. *Given a Collatz function g, we can construct a non-mutually recursive unary system defining a stream function v such that $\overline{v} = g$.*

Before going into the details of the proof, note that, although it is quite clear that the above encoding of Collatz functions already enables an embedding of full computational power into unary stream equations, what is not at all obvious is whether we can actually define every computable unary stream function through a purely unary system.

Proof. Let modulus $n > 0$ and coefficients $a_i, b_i \in \mathbb{N}$ be as in the above definition. Define a stream function

$$\mathrm{add}(s) = \mathrm{head}(\mathrm{tail}^{n\times a_0+0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{n\times a_{n-1}+(n-1)}(s)) :: \mathrm{add}(\mathrm{tail}^n(s)).$$

Lemma 5.2.1 shows that $\overline{\mathrm{add}}(k) = \lfloor \frac{k}{n} \rfloor \times n + (n \times a_{k \bmod n} + (k \bmod n)) = k + n \times a_{k \bmod n}$ for $k \in \mathbb{N}$. The role of this function is to act as a crude replacement conditional for the unavailable zip, adding different constants depending on the equivalence class of the stream index modulo n.

Next, define a stream function

$$u(s) = \mathrm{head}(\mathrm{tail}^{n\times b_0+0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{n\times b_{n-1}+(n-1)}(s)) :: u(\mathrm{add}(s)).$$

Using Lemma 5.2.1, we derive for $q \in \mathbb{N}$:

$$\overline{u}(n \times q + i) = \overline{\mathrm{add}}^q(n \times b_i + i) = (n \times b_i + i) + q \times (n \times a_i) = n \times g(k) + i \ .$$

This function is an approximation to g, the only difference being that the output indices come pre-multiplied by n. We fix this by defining a stream function

$$\mathrm{div}(s) = \underbrace{\mathrm{head}(s) :: \ldots :: \mathrm{head}(s)}_{n \text{ times}} :: \mathrm{div}(\mathrm{tail}(s)) \ .$$

Applying Lemma 5.2.1 yields for $k \in \mathbb{N}$:

$$\overline{\mathrm{div}}(k) = \left\lfloor \frac{k}{n} \right\rfloor \ .$$

Finally defining $v(s) = u(\mathrm{div}(s))$ yields the Collatz function semantics that we want:

$$\overline{v} = \overline{\mathrm{div}} \circ \overline{u} = g \ . \qquad \square$$

For instance, the original Collatz function would be encoded as collatz $= \overline{v}$ as follows:

$$\begin{aligned}
\mathrm{add}(s) &= \mathrm{head}(\mathrm{tail}^2(s)) :: \mathrm{head}(\mathrm{tail}^{13}(s)) :: \mathrm{add}(\mathrm{tail}^2(s)),\\
u(s) &= \mathrm{head}(s) :: \mathrm{head}(\mathrm{tail}^9(s)) :: u(\mathrm{add}(s)),\\
\mathrm{div}(s) &= \mathrm{head}(s) :: \mathrm{head}(s) :: \mathrm{div}(\mathrm{tail}(s)),\\
v(s) &= u(\mathrm{div}(s)).
\end{aligned}$$

For further illustration, let us evaluate stream position 3 of $v(s)$:

$$\begin{aligned}
v(s)\ !\ 3 &\rightsquigarrow^* u(\mathrm{div}(s))\ !\ 3 \rightsquigarrow^* u(\mathrm{add}(\mathrm{div}(s)))\ !\ 1 \rightsquigarrow^* \mathrm{add}(\mathrm{div}(s))\ !\ 9\\
&\rightsquigarrow^* \mathrm{add}(\mathrm{tail}^2(\mathrm{div}(s)))\ !\ 7 \rightsquigarrow^* \ldots \rightsquigarrow^* \mathrm{add}(\mathrm{tail}^8(\mathrm{div}(s)))\ !\ 1\\
&\rightsquigarrow^* \mathrm{div}(s)\ !\ 21 \rightsquigarrow^* \mathrm{div}(\mathrm{tail}(s))\ !\ 19 \rightsquigarrow^* \ldots \rightsquigarrow^* \mathrm{div}(\mathrm{tail}^{10}(s))\ !\ 1\\
&\rightsquigarrow^* s\ !\ 10.
\end{aligned}$$

This is consistent with collatz$(3) = 3 \times 3 + 1 = 10$.

Note that this encoding is fundamentally different from our encoding encoding of Collatz functions in Chapter 4, the one used by Endrullis et al. [EGH09b], the essential difference being the unavailability of zip in the unary setting. The dispatch mechanism of interleaving, enabling differing treatment of stream positions based on the residue of their indices, makes the implementation of Collatz-like constructs rather straightforward. Since we are lacking even such basic conditional control flow mechanisms, we have to resort to highly indirect constructions such as the above.

The role of Collatz functions in our setting is to serve as an intermediate between indexing functions and the known world of computability. To make the latter link clearer, we will show how Collatz functions relate semantically to different register machine models under the prime factorisation register encoding introduced in the previous section.

Definition 5.2.4. *The inductive set of* IF-programs *is generated by concatenation* $A_0 \dots A_{n-1}$*, increments* $\operatorname{inc}(r)$*, decrements* $\operatorname{dec}(r)$*, and conditional clauses* $\operatorname{ifz}(r, A, B)$*, where* $n \in \mathbb{N}$*,* $r \in \mathbb{N}$ *designates a register, and* $A_0, \dots, A_{n-1}, A, B$ *are* IF*-programs.*

Although it is quite clear intuitively what the semantic effects of running an IF-program A on some register state $R \in \mathbb{N}^{(N)}$ are, we will formally introduce an associated semantics function $\chi_A : \mathbb{N}^{(\mathbb{N})} \to \mathbb{N}^{(\mathbb{N})}$ defined structurally as follows:

$$\begin{aligned}
\chi_{A_0 \dots A_{n-1}} &= \chi_{A_{n-1}} \circ \dots \circ \chi_{A_0}, \\
\chi_{\operatorname{inc}(r)}(R) &= R[r \leftarrow R(r) + 1], \\
\chi_{\operatorname{dec}(r)}(R) &= R[r \leftarrow \max(R(r) - 1, 0)], \\
\chi_{\operatorname{ifz}(r,A,B)}(R) &= \begin{cases} \chi_A(R) & \text{if } R(r) = 0, \\ \chi_B(R) & \text{else.} \end{cases}
\end{aligned}$$

Note that a decrement on a zero-valued register is ignored.

We use again the Gödel encoding for the registers $\widehat{\cdot} : \mathbb{N}^{(\mathbb{N})} \to \mathbb{N} \setminus \{0\}, R \mapsto \widehat{R} \triangleq \prod_{r \in \mathbb{N}, R(r) \neq 0} p_r^{R(r)}$ from the previous section. Translated to this setting, the semantics function of an IF-program A takes the form $\widehat{\chi}_A := \widehat{\cdot} \circ \chi_A \circ \widehat{\cdot}^{-1}$. Although this is an endofunction on the positive integers, to make the following treatment more uniform, we will extend it to the natural numbers by setting $\widehat{\chi}_A(0) := 0$.

Lemma 5.2.5. *Given an if-program* A*, its semantics* $\widehat{\chi}_A : \mathbb{N} \to \mathbb{N}$ *on the register encoding is a Collatz function.*

Proof. By induction on the structure of A, noting that:

- The composition of finitely many Collatz functions of moduli $m_0 \times \dots \times m_{n-1}$ is a Collatz function of modulus $m_0 \times \dots \times m_{n-1}$.
- Given a register state $R \in \mathbb{N}^{(\mathbb{N})}$, increment of register r corresponds to multiplication of $\widehat{R}$ with p_r, a Collatz function of modulus 1.
- Decrement of register r corresponds to division of $\widehat{R}$ by p_r if the former is divisible by p_r, and no change otherwise. This is a Collatz function of modulus p_r.
- Let $\widehat{\chi_A}$ and $\widehat{\chi_B}$ be Collatz functions of moduli m_A and m_B, respectively. A conditional clause $\operatorname{ifz}(r, A, B)$ corresponds first to case distinction depending on whether $\widehat{R}$ is divisible by p_r and subsequent application of either $\widehat{\chi_A}$ or $\widehat{\chi_B}$. This is a Collatz function of modulus the least common multiple of p_r, m_A, m_B. □

We note that the Collatz function $\widehat{\chi}_A$ in the previous lemma is special in that it is linear on each of its equivalence classes in the strict sense, i.e. with vanishing ordinate, corresponding to single multiplication with a fraction. Even though we do not make use of this fact in our developments, it shows the connection between IF-programs and the iteration steps of the FRACTRAN-programs of Conway [Con87], which are of equivalent expressive power.

5.2.3. Iteration-Programs and Their Encoding

Unsurprisingly, the expressive power of IF-programs by themselves is quite limited. To achieve computational completeness, we need an unbounded looping construct. The following definition intends to provide a minimal such model, enabling us to concentrate on the essential details of the conversion from Turing-complete programs to stream equation systems.

Definition 5.2.6. *An* ITERATION-program P *is a tuple* $(\mathrm{B}_P, \mathsf{input}_\mathrm{P}, \mathsf{output}_P, \mathsf{loop}_P)$ *consisting of an* IF*-program* B_P *called the* body *of* P *and designated and mutually distinct* input*,* output *and* loop *registers* $\mathsf{input}_\mathrm{P}, \mathsf{output}_P, \mathsf{loop}_P \in \mathbb{N}$.

The semantics of such a program is a computable function $\phi_P : \mathbb{N} \to \mathbb{N}_\perp$ defined as follows: given an input $i \in \mathbb{N}$, the register state $R_0 \in \mathbb{N}^{(\mathbb{N})}$ is initialised with $R_0(\mathsf{input}_\mathrm{P}) := i$, $R_0(\mathsf{loop}_P) := 1$, and $R_0(r) := 0$ for $r \neq \mathsf{input}_\mathrm{P}, \mathsf{loop}_P$. We iteratively execute the body of P, yielding $R_{n+1} := \chi_{\mathrm{B}_P}(R_n)$ for $n \in \mathbb{N}$. If there is n minimal such that $R_n(\mathsf{loop}_P) = 0$, then P is called *terminating* with *iteration count* $\operatorname{count} P(i) := n$ and output $\phi_P(i) := R_n(\mathsf{output}_P)$ for input i. Otherwise, $\operatorname{count}_P(i) := \perp$ and $\phi_P(i) := \perp$.

Intuitively, an ITERATION-program is just a WHILE-program [Per74] with a single top-level loop, a well-studied concept in theoretical computer science bearing resemblance to the normal form theorem for μ-recursive functions [Kle43], [Soa87] except that we do not even allow primitive recursion inside the loop.

Theorem 5.2.7. *Given a computable function* $\phi : \mathbb{N} \to \mathbb{N}_\perp$*, there is an* ITERATION*-program* P *with semantics* $\phi_P = \phi$.

Proof. This is a folk theorem [Har80], see Böhm and Jacopini [BJ66] and Perkowska [Per74] for more details. □

The reason behind our choice for this computationally complete machine model is that we already have the machinery to simulate a single execution of the body of such a machine via Collatz functions as indexing functions of stream equations using our prime factorisation exponential encoding on the register state. In fact, another option would have been the FRACTRAN-programs of Conway [Con72] [Con87], but we are in need of a more conceptual representation in light of what lies ahead of us.

We will now investigate how to translate a top-level unbounded looping construct into the recursive stream equation setting.

Using Lemmata 5.2.5 and 5.2.3, we can translate the encoded iteration step function $\widehat{\chi}_{\mathrm{B}_P} : \mathbb{N} \to \mathbb{N}$ of an ITERATION-program P to an indexing function of a stream equation for some u. We would like to use this stream function u as it appears in Lemma 5.2.2 in a way such that the minimal choice of $c(k)$ corresponds to the iteration count of P. Unfortunately, the equivalent of the stopping condition in the lemma, that the index be smaller than some constant h, corresponds to $\widehat{R} < h$ for the register state $R \in \mathbb{N}^{(\mathbb{N})}$, a statement which does not have a natural meaning for the registers of R individually, forestalling us from expressing the condition $p_{\mathsf{loop}_P} \mid \widehat{R}$ corresponding to the termination condition $R(\mathsf{loop}_P) = 0$. A second problem comes from our desire to somehow extract the value of $R(\mathsf{output}_P)$ after termination. But since at this point of time $\widehat{R}$ is limited to a finite set of values, there is no direct way of realising this.

What we can do is extract the iteration count for particularly nicely behaving programs.

Lemma 5.2.8. *Given an* ITERATION*-program Q such that whenever Q terminates, all its registers are zero-valued, i.e. $\chi_{\mathrm{B}_Q}^{\mathrm{count}_Q(i)} = (0,0,\ldots)$ for terminating input $i \in \mathbb{N}$, there is a non-mutually recursive unary system defining a stream function w such that $\overline{w} = \mathrm{count}_Q$.*

Proof. In anticipation of applying Lemma 5.2.2, we extend this system with a new equation

$$q(s) = \mathrm{head}(s) :: \mathrm{head}(s) :: \mathrm{tail}^2(v(q(\mathrm{tail}(s))).$$

Given an input $i \in \mathbb{N}$ and corresponding initial register state $R \in \mathbb{N}^{(\mathbb{N})}$, the termination condition in Lemma 5.2.2 can equivalently be expressed as follows:

$$\overline{v}^{c(\widehat{R})}(\widehat{R}) < 2 \iff \widehat{\chi}_{\mathrm{B}_Q}^{c(\widehat{R})}(\widehat{R}) = 1 \iff \chi_{\mathrm{B}_Q}^{c(\widehat{R})}(R) = (0,0,\ldots).$$

Now, by our assumption on the behaviour of Q, the first point in time all registers are zero equals the first point in time the loop register attains zero. But by our definition of the iteration count, this just means that $c(\widehat{R}) = \mathrm{count}_Q(i)$, and Lemma 5.2.2 shows that

$$\overline{q}(\widehat{R}) = \overline{\mathrm{tail}}^{c(\widehat{R})}(0) = c(\widehat{R}) = \mathrm{count}_Q(i) \ .$$

All that remains is to produce the initial register state $R_{(i)}$ with only $R_{(i)}(\mathsf{input_Q}) = i$ and $R_{(i)}(\mathsf{loop}_Q) = 1$ non-zero. For this, we define

$$r(s) = \mathrm{head}(\mathrm{tail}^{p_{\mathsf{loop}_Q}}(s)) :: r(\mathrm{proj}_{p_{\mathsf{input_Q}}}(s))$$

and we use Lemma 5.2.1 to prove that

$$\overline{r}(i) = \overline{\mathrm{proj}_{p_{\mathsf{input_Q}}}}^{i}(p_{\mathsf{loop}_Q}) = p_{\mathsf{input_Q}}^{i} \times p_{\mathsf{loop}_Q} = \widehat{R_{(i)}} \ .$$

Defining $w(s) = r(q(s))$, we verify that

$$\overline{w}(i) = \overline{q}(\overline{r}(i)) = \overline{q}(\widehat{R_{(i)}}) = \mathrm{count}_Q(R_{(i)}) \ . \qquad \square$$

Unfortunately, the set of possible iteration count functions constitutes only a small part of the set of all computable functions. Intuitively, this is because even very small values can be the result of prohibitively expensive operations. However, this range can still be seen as containing Turing-complete fragments under certain encodings. This is what we exploit in the next step by shifting the role of the output register to the iteration count under a particular such encoding. The trick is to have each possible output value correspond to infinitely many iteration counts in a controlled way such that after having computed the result, by being self-aware of the current iteration count, we can consciously terminate the loop at one of these infinitely many counts, no matter how long the computation took.

Lemma 5.2.9. *Given an* ITERATION*-program P, there is an* ITERATION*-program Q such that for every input i natural, Q terminates if and only if P terminates, and furthermore if P terminates with output $o \in \mathbb{N}$, then Q terminates after exactly $(3m+1) \times 3^{o+1}$ iterations with all registers zero-valued where $m \in \mathbb{N}$ depends on i.*

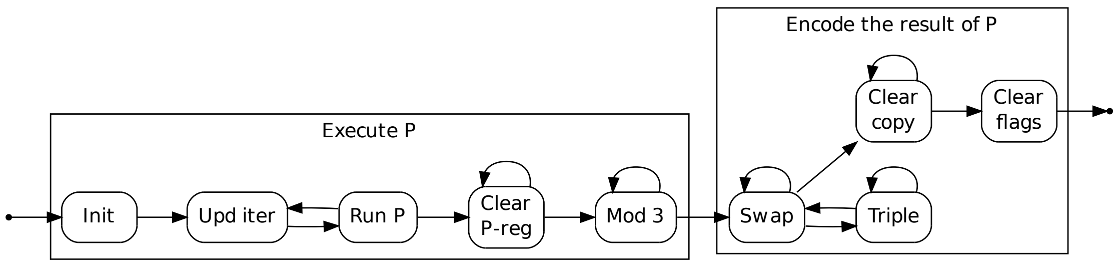
Proof. Let $r_0, \ldots, r_{k-1} \in \mathbb{N}$ denote all the registers occurring in B_P except for output_P and loop_P (but including $\mathsf{input}_\mathrm{P}$). We choose loop_Q as a fresh natural number distinct from all previously mentioned registers. Both programs will have the same input register, i.e. $\mathsf{input}_\mathrm{Q} := \mathsf{input}_\mathrm{P}$. The output register of Q is irrelevant since we aim to have all registers reset at termination.

The body of Q is listed in Listing 5.1. Note that the body of P is textually inserted at line 14. Register names main-phase, run-time, mod-three, swap-phase, copy also designate fresh natural numbers. To enhance readability, we used some lyrical freedom with the syntax: for example, **if** $R(\mathsf{output}_P) \neq 0$ **then** A **else** B **end if** translates to $\mathrm{ifz}(\mathsf{output}_P, B, A)$. Since the program is somewhat complex, we will describe its function in great detail.

```
1.  if $R$(main-phase) = 0 then
2.      if $R$(run-time) = 0 then
3.          inc(loop$_P$)
4.          inc(mod-three)
5.      end if
6.      inc(run-time)
7.      if $R$(mod-three) = 0 then
8.          inc(mod-three)
9.          inc(mod-three)
10.         inc(mod-three)
11.     end if
12.     dec(mod-three)
13.     if $R$(loop$_P$) $\neq 0$ then
14.         $\mathrm{B}_P$
15.     else if $R(r_0) \neq 0$ then
16.         dec($r_0$)
17.     [...]
18.     else if $R(r_{n-1}) \neq 0$ then
19.         dec($r_{n-1}$)
20.     else if $R$(mod-three) = 0 then
21.         inc(main-phase)
22.     end if
23. else if $R$(swap-phase) = 0 then
24.     dec(run-time)
25.     inc(copy)
26.     if $R$(run-time) = 0 then
27.         inc(swap-phase)
28.     end if
29. else
30.     dec(copy)
31.     if $R$(output$_P$) $\neq 0$ then
32.         inc(run-time)
33.         inc(run-time)
34.         inc(run-time)
35.         if $R$(copy) = 0 then
36.             dec(swap-phase)
37.             dec(output$_P$)
38.         end if
39.     else if $R$(copy) = 0 then
40.         dec(swap-phase)
41.         dec(main-phase)
42.         dec(loop$_Q$)
43.     end if
44. end if
```

Listing 5.1.: The body of Q from Lemma 5.2.9

To explain the flow of the program, we provide a state diagram, corresponding to different phases of the program Q, and a table giving for each phase the condition on the current register values just before an iteration of Q for the phase to be activated. Most phases necessitate more than one iteration of Q, this is why in the diagram, they have arrows pointing to themselves. Some transitions between phases happen within a single iteration of Q, this is the case of *init* immediately followed by *run P*, and the last iteration of *clear copy* followed by *clear flags* in the same iteration of Q.



The following table summarises the tests that select each phase. The columns correspond to phases, and the rows to registers. Each cell of the table is either 0 when the corresponding register needs to be null in this phase, > when it cannot be null, and nothing is written when the register is not tested by the phase. Note that the tests are mutually exclusive.

	Execute P				Encode the result of P			
	Init	Upd Iter & Run P	Clear (r_k) P-registers	Mod 3	Swap	Triple	Clear Copy	Clear Flags
main-phase	0	0	0	0	>	>	>	>
run-time	0							
loop$_P$		>	0	0				
r_k			>	0				
mod-three				>				
swap-phase					0	>	>	>
output$_P$						>	0	0
copy							>	0

Execution of Q, i.e. iterated execution of B_Q until decrement of loop$_Q$, is split into two macro phases, as signalled by the flag register main-phase, The first macro phase, when main-phase has value 0 (lines 2–22), is dedicated to executing the original program P while keeping track of the total iteration count in a dedicated register run-time. At the end of this phase, after P has exited with result $o \in \mathbb{N}$ in register output$_P$, we want the total iteration count to equal $3m + 1$ for some arbitrary $m \in \mathbb{N}$. The second macro phase, when main-phase has value 1 (lines 23–44) performs the computations necessary to encode the result of P in the number of iterations: we must triple the total iteration count o times, plus an additional tripling to reset run-time, so that the total iteration count becomes $(3m+1) \times 3^{o+1}$.

Every phase is now described in detail, referring to the program in Listing 5.1, the line numbers corresponding to each phase are given next to the name.

The first macro phase (lines 2–22) consists of:

Init. 3–4

Executed only at the beginning of the first iteration to initialise the loop register of P and mod-three (see next phase). Note that the input register of P does not need to be initialised as $\mathsf{input_P} = \mathsf{input_Q} = i$

Update Iteration Count. 6–12
Keep track not only of the current iteration count by incrementing run-time once per iteration, but also of how many iterations modulo 3 we are afar from meeting the $(3m + 1)$-condition in a dedicated register mod-three.

Run P. 13–14
Execute the body of P (textually inserted) once per iteration until termination is signalled by loop_P being set to zero.

We leave the phase when $\mathsf{loop}_P = 0$. At this point, the result of executing P on input i is stored in register 'output_P', let o be this value.

Clear P-Registers. 15–19
All the registers used in P except its output register are reset to zero by successive decrements.

Mod 3. 20–21
Additional iterations, up to two, might be necessary to get a total iteration count equal to one modulo three (tested in line 20). The register 'mod-three' go through the values $0, 2, 1, 0, 2, 1, 0, 2, 1, 0, \cdots$. it is zero exactly when run-time $\equiv 1 \bmod 3$. we then set main-phase and proceed to the second macro phase (line 21).

After the final iteration of this phase, the iteration count is $3m + 1$ for some $m \in \mathbb{N}$ and the only possibly non-zero registers are main-phase and swap-phase of value 1 and output_P of value o. We duly note that if P does not terminate, then neither does Q.

The second macro phase (lines 23–44) consists of two alternately executed sub-phases, *swap* and *triple* responsible for shifting the iteration count back and forth between the registers run-time and copy. The current sub-phase is indicated by the flag register swap-phase:

Swap. 24–28
Entered with register values $[\mathsf{swap\text{-}phase} : 0, \mathsf{run\text{-}time} : x \geq 1, \mathsf{copy} : 0]$.

Left after x iterations, with values $[\mathsf{swap\text{-}phase} : 1, \mathsf{run\text{-}time} : 0, \mathsf{copy} : x]$.

Triple. 30–43
Entered with $[\mathsf{swap\text{-}phase} : 1, \mathsf{run\text{-}time} : 0, \mathsf{copy} : x \geq 1, \mathsf{output}_P : y \geq 1]$.

Left after x iterations, with $[\mathsf{swap\text{-}phase} : 0, \mathsf{run\text{-}time} : 3x, \mathsf{copy} : 0, \mathsf{output}_P : y - 1]$.

Considering *swap* and *triple* together, we establish that starting (the first sub-phase) with $[\mathsf{swap\text{-}phase} : 0, \mathsf{run\text{-}time} : x \geq 1, \mathsf{copy} : 0]$ and output_P non-zero, after $2x$ iterations, the effective changes will be tripling of run-time and decrement of output_P. In particular, if x and hence run-time denoted the iteration count before these iterations, run-time will again denote the iteration count after these iterations. After following this reasoning o times, the iteration count and value of run-time will be $(3m + 1) \times 3^o$ when output_P reaches zero. One last instance of the swap phase, necessary to reset run-time, costs $(3m+1) \times 3^o$ iterations, ends with register copy $= (3m + 1) \times 3^o$ and with a total iteration count of $2 \times (3m + 1) \times 3^o$.

Clear Copy. 30
Clearing the register copy by successive decrements bring total iteration count to $(3m + 1) \times 3^{o+1}$. The last iteration of Q has both a *clear copy* and a *clear flags* phase.

Clear Flags. 39–42
This phase is executed during the last iteration, just after the last decrement of copy, and is necessary to clear the remaining flags main-phase and swap-phase, so that all registers are null without exception. In particular, clearing loop_Q signals the end of the program Q. □

5.2.4. Proof of Unary Decidability

Theorem 5.2.10. *A unary polymorphic stream function is definable by a non-mutually recursive unary system if and only if its indexing function is computable.*

Proof. We need only consider the reverse implication. Let a computable function $\phi : \mathbb{N} \to \mathbb{N}_\perp$ be represented as an ITERATION-program P, i.e. $\phi = \phi_P$, and let Q be the modified ITERATION-program as defined in Lemma 5.2.9. By Lemma 5.2.8, there is a non-mutually recursive unary system defining a stream function w such that for all $i \in \mathbb{N}$ there exists $m \in \mathbb{N}$,

$$\overline{w}(i) = \text{count}_Q(i) = (3m + 1) \times 3^{\phi_P(i)+1} \quad .$$

As stated at the beginning, the latter expression is taken to mean $\perp$ if $\phi_P(i) = \perp$.

Our strategy for extracting the final output value $\phi_P(i)$ from this expression is by iterating a second program, adding a tail for each time the stream index is divisible by 3. In particular, from Lemma 5.2.3, it is clear how to give a non-mutually recursive unary system defining u such that

$$\overline{u}(k) = \begin{cases} k/3 & \text{if } 3 \mid k, \\ 0 & \text{else} \end{cases}$$

since this is a Collatz function. But note that we can alternatively directly define

$$\begin{aligned} u(s) = {} & \text{head}(s) :: \text{head}(s) :: \text{head}(s) :: \\ & \text{head}(\text{tail}(s)) :: \text{head}(s) :: \text{head}(s) :: \text{tail}^3(u(\text{head}(s) :: \text{tail}^2(s))) \end{aligned}$$

using only a single equation. For either choice, we define

$$v(s) = \text{head}(s) :: \text{tail}(u(v(\text{tail}(s))))$$

A second application of Lemma 5.2.2 shows that for $i, m \in \mathbb{N}$,

$$\overline{v}((3m + 1) \times 3^{i+1}) = i + 1 \ .$$

This function is almost as critically important as w as it repeats each natural number output infinitely many times in a controlled way and reverses the iteration count result encoding of program Q. We now have all the parts necessary for concluding our venture. Defining $f(s) = w(v(\text{head}(s) :: s))$, we see that

$$\begin{aligned} \overline{f}(i) &= \max(\overline{v}(\overline{w}(i)) - 1, 0) \\ &= \max(\overline{v}((3m + 1) \times 3^{\phi_P(i)+1}) - 1, 0) \\ &= \max((\phi_P(i) + 1) - 1, 0) = \phi_P(i) \end{aligned}$$

with our usual convention regarding $\perp$, proving $\overline{f} = \phi_P$. □

5.3. Further Results

Christian Sattler must be credited for the yet unpublished results whose proofs we omit here and which characterise with even tighter constraints the expressivity of pure stream equations. A thorough investigation of the proofs in the previous section shows that, in total, ten equations were defined for proving the main result, two of which can be inlined. It is natural to ask: what is the minimum number of unary equations required to define an arbitrary computable indexing function? It can be established with a rather involved proof that *four* equations free of mutual recursion suffice. Furthermore, it can be shown that recognising productivity is undecidable with complexity Π^0_2 even for unary systems with only *two* non-mutually recursive equations. Interestingly, Sattler proved that productivity is decidable for a *single* unary equation.

Altogether, this work amounts to an exhaustive classification of definability and complexity of recognising productivity based on unary system size.

Part II.

Circular Traversals Compositionally

Chapter 6. Circular Traversals Using Algebras

In this chapter, we present circular traversals. They allow to combine many traversals into a single one and rely on lazy evaluation. There is a systematic technique to transform many traversals into a circular one, but such a technique is not modular. We propose a new abstraction that captures circular traversals and allow to compose them.

Overview §6.1 explains circular traversals through an example. §6.1.1 presents the technique in general terms. §6.1.2 raises the issues with the technique and motivates our research. §6.2 defines the abstract interface for our combinator library. §6.2.1 shows that the environment `Arrow` is suitable to define non-circular traversals compositionally. §6.2.2 extends the `Arrow` interface with a method for constructing primitive traversals as iteration of an algebra. §6.3 gives an implementation of the interface using circular traversals. §6.3.1 defines the recursive pattern capturing circular traversals. §6.3.2 gives the actual implementation. §6.3.3 proves that this circular implementation is semantically equivalent to the non-circular implementation. §6.3.4 shows that our implementation has the underlying structure of an indexed monad. §6.4 is an example of using the library.

6.1. Introduction

Richard Bird [Bir84] popularised the technique of combining traversals using circularity. His motivating example is a function `repMin` that computes a binary tree with the same shape as the input tree, but where all the leaves are replaced with the minimum value found in the input tree. A straightforward implementation requires two traversals of the tree: the first `minTree` to compute the minimum, and the second `replace` to create the new tree, replacing leaves of the input tree with the minimum while preserving the structure.

```
data Tree = Leaf Int | Fork Tree Tree

minTree (Leaf x) = x
minTree (Fork l r) = min (minTree l) (minTree r)

replace y (Leaf x) = Leaf y
replace y (Fork l r) = Fork (replace y l) (replace y r)

repMin tree = replace (minTree tree) tree
```

We can rewrite `repMin` by binding the result of each traversal:

```
repMin' tree =
  let min = minTree tree
      rep = replace min tree
  in rep
```

The bindings are combined in a product, making the definition circular:

```
repMin'' tree =
  let (min, rep) = (minTree tree, replace min tree)
  in rep
```

The right hand side is abstracted:

```
repMinCirc tree =
  let (min, rep) = min_and_replace min tree
  in rep
```

It is critical that `min_and_replace` is non-strict in its first argument, otherwise the circular definition `repMinCirc` would diverge.

`min_and_replace` takes a replacement value and an input tree, and computes a pair of the minimum of the input tree and a new tree with the same structure but with the leaves replaced with the replacement value. We can implement `min_and_replace` in a single recursion. Bird's starts with the product of `minTree` and `replace` (two traversals) and applies *unfold* and *fold* steps to eliminate the calls to `minTree` and `replace`.

```
min_and_replace y (Leaf x) = (x, Leaf y)
min_and_replace y (Fork l r) = (min ml mr, Fork tl tr)
  where (ml, tl) = min_and_replace y l
        (mr, tr) = min_and_replace y r
```

It is indeed the case that `min_and_replace` is non-strict in its second argument, thus `repMinCirc` is well-defined.

The circular programming technique generalises the previous example and applies to similar cases. The technique answers the question: *given an implementation doing many traversals, can we write a semantically equivalent program with only one traversal?*

6.1.1. Circular Traversals in General

A circular program is a form of recursive definition in which a structured value is defined as a function of itself.

All circular programs studied here are concerned with the computation of a tuple obtained by flattening mutually recursive bindings. Consider some terms `A1`,$\cdots$`An`, `E` with free variables `x1`,$\cdots$ `xn` bound by a let expression:

```
let { x1 = A1 ; x2 = A2 ; ··· ; xn = An } in E
```

By making a tuple of all the bound variables, and all the right hand sides, we can bind all the variables at the same time in a single circular definition:

```
let (x1, ···, xn) = (A1,···,An) in E
```

Now the right hand side is expressed as a function of the bound variables:

```
f x1 ··· xn = (A1, ···, An)
```

The circular program is therefore equivalent to

```
let (x1, ···, xn) = f x1 ··· xn in E
```

The circular binding we obtain is operationally equivalent to the mutual bindings. The advantage of writing a single circular definition, rather than a group of mutually recursive definitions is that, having regrouped all the individual computations in one function `f`, we can then try to optimise it using program reasoning. The subject of this chapter is the optimisation of programs performing many traversals of the same datastructure by combining them in a circular definition which is then optimised to perform only one traversal. Imagine in the previous program that all $\mathtt{A}_1,\cdots,\mathtt{A}_n$ do a traversal over some datastructure. The goal is to find a function `g` equivalent to `f` but computing the product of those different traversals by traversing the input only once.

6.1.2. Issues With the Transformation

This programming pattern has some weaknesses:

- It is sometimes difficult to ensure totality and semantic equivalence of the optimised program.
- The optimised program is more difficult to understand than the original.
- Modularity is lost: we're no longer composing independent traversals, we must rewrite them and merge them in the optimised program.

In this chapter we address each of those issues by defining a library for defining circular programs modularly, preserving the clarity and semantics of the multi-traversal implementation.

Diverging programs When applying the technique, one must be careful that the optimised program doesn't diverge.

If `E` is strict in one of $\mathtt{x}_1,\cdots,\mathtt{x}_n$, then the circular program `let` $(\mathtt{x}_1,\cdots,\mathtt{x}_n)$ `= C in E` terminates *only if* `C` is non-strict in all the parameters $\mathtt{x}_1,\cdots,\mathtt{x}_n$. That is because to force the evaluation of a $\mathtt{x}_i$, `C` must first be evaluated and the computation would loop if `C` were strict in one of $\mathtt{x}_1,\cdots,\mathtt{x}_n$. In the particular case that `C =` $(\mathtt{A}_1,\cdots\mathtt{A}_n)$, `C` is trivially non-strict in $\mathtt{x}_1,\cdots,\mathtt{x}_n$. However, optimised `C` are usually not of this simple form.

Sometimes (cf §7.1) it is difficult to keep `C` non-strict whilst optimising it.

Complex and non-modular programs When optimising `(`$\texttt{A}_1$`,`$\cdots$$\texttt{A}_n$`)` into a `C`, the simple computations `A_i` are merged into a complex unmodular program `C` where all aspects are intertwined and thus difficult to understand. Additionally, we lose the advantage of defining each aspect separately, the work is repeated again in `C`. Ultimately if some corrections were to be made to some $\texttt{A}_i$, all the circular programs defined in terms of it would need to reflect the changes.

Unfortunately the programs obtained by the circular transformation always appear more complex than the original non-circular ones because they merge all the aspects of the computation together. Bird acknowledges this [Bir84, §4]:

> Though incomprehensible taken by itself, the final version has been derived systematically from the original specification using cyclic programming in conjunction with other transformations.

In addition, modularity is lost: whereas `repMin` was the composition of `replace` and `minTree`, they cannot be reused in `repMinCirc`: instead, their bodies are fused in `min_and_replace`. This has many unfortunate consequences: The functionality of each traversal, like `replace` and `minTree` are duplicated, and it is difficult and error-prone to keep a coherent state during the development of the program as corrections and improvements must be reflected on each duplicate. Additionally, errors can be introduced whilst fusing the functions, And the newly fused function is more complex, bigger, harder to analyse, thus harder to maintain. Finally, the transformation imposes some constraints on the traversals in order to ensure termination (the non-strictness of `replace` in our example).

Attribute Grammars The drawbacks can be somewhat mitigated. One approach is to automatise the transformation [CGK99]. This allow us to keep programming modularly as before, with many traversals, and run a tool to obtain the circular single traversal equivalent. Another approach is to use the attribute grammar formalism to program each traversals and rely on a tool to translate the attribute grammar into a circular functional program. Alternatively, if the attribute grammar formalism is implemented as a domain specific language (DSL) in the host functional programming language, as in [dMBS00] or [VSS09], then each traversal can be implemented as a grammar *aspect* and combined into a circular computation using the combinators of the DSL.

Relying on external tools for program transformation does alleviate the modularity loss but adds to the complexity of the development process, which may deter programmers from using it. The attribute grammar formalism is also somewhat limited in modularity as evidenced by the number of articles proposing diverse extensions to the paradigm: [VSK89, DC90, KW92, FMY92, Hed99, MLAZ99, VS12]. DSL implementations of attribute grammars may be weakly typed [dMBS00], or they may not allow the definition of different aspects, which is necessary if we want to reuse them [MFS13, UV05]. Reviews of attribute grammar systems for Haskell are given in the introduction, §1.5.2.

It has been suggested that the previous issues can be somewhat mitigated by expressing the traversals as attribute grammars and compiling them using a special preprocessor, or using an embedded language for them. But those solutions are often more complex and less modular than the original formulation and require the trouble of using new language syntax, an external tool (preprocessor) or a macro processor (template Haskell) together with various language extensions.

Our Solution for Compositionality In this chapter, we recover the lost compositionality without resorting to changing the compiler or the language. We propose a set of combinators that encapsulate a recursive pattern and hide away the circularity from the programmer's concern. For instance, with our combinators, the user will be able to build `repMinCirc` by composing a `MinCirc` and a `ReplaceCirc` much like `repMin` was built by composing `min` and `replace`. The composition of circular definitions is based on the `Arrow` type-class, which generalises function composition. The programs written using the combinators are structurally very similar to the program doing many traversals, the differences are that the datastructure being traversed is implicit and that we use the arrow notation [Pat01] to express the programs. We show below, side by side, the `repMin` program in the arrow notation and the original implementation.

```
repMinA = proc () → do
  m ← minA ≺ ()
  replaceA ≺ m
```

```
repMin tree =
  let m = minTree tree
   in replace m tree
```

Additionally, it is possible to choose whether many traversals or only one should be done according to which instance we choose. Furthermore, in contrast with the use of automatic program transformation tools and attribute grammar compiler, not only the source code is modular, but compilation as well. Additionally, using the combinators prevents errors that can occur when doing the transformation manually: in particular, certain strictness properties of the traversals could make the circular program non-terminating [Bir84, §3]. The library interface is given as a Haskell type-class, for which we provide two implementations: a circular implementation with one traversal and a strict implementation with many traversals. The user code can be executed with either approach depending on his needs.

We developed the library in two stages. The first version relies on algebras and is too limited to cover Bird's palindrome example [Bir84, §3]. To remedy this limitation, we found a novel traversal abstraction, that generalises algebras and has a direct explanation with attribute grammars.

6.2. Abstract Programming Interface for Computations over a Data Structure

A library should always provide an interface that hides details about the concrete implementation. In our particular case, the library should give us the means to compose traversals on the same datastructure, without being concerned by the operational semantics. In fact, it should be possible to give two implementations: one that traverses the datastructure many times, and one that traverses it only once. Therefore we must abstract over the notion of function and composition. The `Arrow` type class [Hug98] is such an abstraction: it captures function like objects that can be composed and paired. Functions can be lifted to *pure* `Arrows`.

The `Category` type class generalises the category of Haskell types and functions where objects are types, but where the arrows are more general. We use a type operator variable `(⇝)` for hom-sets to make the type signatures nicer.

```
class Category (⇝) where
  id  :: a ⇝ a
  (.) :: b ⇝ c  →  a ⇝ b  →  a ⇝ c
```

The name `id` and `(.)` clash with the `Prelude` export but generalise them: the `Category` instance for Haskell functions correspond to the `Prelude` definitions. Thus we hide the `Prelude` definition and use their `Category` generalisation:

```
import Prelude hiding (id, (.))
import qualified Prelude
instance Category (→) where
 id = Prelude.id
 (.) = (Prelude..)
```

`Arrow` is a subclass of `Category` with more structure which implies the existence of products. `arr` is the action of a functor from the category `(→)` to the category `(⇝)` and `first` is a family of endofunctors' actions.

```
class Category (⇝) ⇒ Arrow (⇝) where
  arr   :: (a → b)  →   a     ⇝  b
  first ::  a ⇝ b   →  (a,c)  ⇝  (b,c)
```

`arr` and `first` imply the universal existence of products objects `(a,b)` for any types `a` and `b`.

```
&&& :: c ⇝ a    →  c ⇝ b  →  c ⇝ (a,b)
f &&& g = arr swap . first g . arr swap . first f . arr dup
  where dup x = (x,x)
        swap (x,y) = (y,x)
arr fst :: (a,b) ⇝ a
arr snd :: (a,b) ⇝ b
```

Functions are `Arrows`: with `arr = id` and `first f (x,y) = (f x,y)`

The arrow notation [Pat01] is an extension to Haskell implemented in GHC, it provides an applicative style for defining arrows instead of the point-free style that is unavoidable when using the arrow primitives `id`, `(.)`, `arr` and `first`. The notation is translated directly into those primitives. The syntax is described in the GHC manual [Tea, §7.17].

To give an intuition of the arrow notation, here is a simple arrow program. In the particular case when the arrow type (⇝) is instantiated with Haskell functions

(→), the arrow program on the left is equivalent to the functional program on the right.

```
example’ f g h j =
  proc (x,y) → do
    a ← f ≺ x
    b ← g ≺ h a y
    returnA ≺ j b
```

```
example’’ f g h j =
 λ(x,y) →
   let a = f x
       b = g (h a y)
    in j b
```

The arrow notation is translated to a functional program that uses the arrow primitives:

```
example :: (Arrow (⇝)) ⇒
  (a ⇝ b) → (c ⇝ d) → (b → p → c) → (d → q) → (a,p) ⇝ q
example f g h j  =  arr j . g . arr (uncurry h) . first f
```

If `(⇝)` is a `Arrow` instance, `f : a ⇝ b`, `x : a`, and `y` is a variable, then `y ← f ≺ x` is a arrow *command* that binds the variable `y` to a value of type `b`. Only an intuitive understanding of the notation is necessary to read this chapter. The GHC manual gives a precise syntax and semantics.

6.2.1. The Environment Arrow

Our problem is to combine traversals over the same datastructure. We make it implicit using the environment arrow.

The type `Env e` captures functions that share the same global argument of type `e`. The `Arrow` abstraction makes this environment implicit: it is automatically passed around when composing and pairing functions.

```
newtype Env e a b = Env {runEnv :: e → a → b}
```

We define `Category` and `Arrow` instances for `Env e` so that the domain and codomain of `Env e a b` are `a` and `b`. We can only compose arrows that have the same environment type. Note that for an `Arrow` it is always possible to define `id` as `arr id`.

```
instance Category (Env e) where
  id = arr id
  Env g . Env f = Env (λe → g e . f e)

instance Arrow (Env e) where
  arr f = Env $ const f
  first (Env f) = Env $ λ e (x,y) → (f e x, y)
```

Let’s see how our running example can be written in the environment arrow. Since the traversal that computes the minimum has no other argument than the tree, we use the unit type for the domain.

```
minEnv :: Env Tree () Int
minEnv = Env $ λt () → minTree t
```

The domain of `replace` is `Int` for the replacement value that goes on every leaf.

```
replaceEnv :: Env Tree Int Tree
replaceEnv = Env $ λt y → replace y t
```

Now the `repMin` function can be defined in the arrow notation.

```
repMinEnv = proc () → do
  m ← minEnv ≺ ()
  replaceEnv ≺ m
```

`runEnv` runs the environment arrow with given global environment. We run the `repMinEnv` program with `runEnv repMinEnv t ()` which is operationally equivalent to the original `repMin` with multiple traversals.

6.2.2. A primitive to define traversals

Although we successfully wrote `repMin` with the arrow notation, the definition is not polymorphic in the `Arrow`: the instance is constrained to `Env Tree` because the two traversal primitives `minEnv` and `replaceEnv` are of this arrow type. We want to make it polymorphic so that we can design a new arrow instance implementing the circular traversal combinators.

Algebras and Catamorphisms

The first step is to abstract over the recursive pattern used to define traversals. Traversals of inductive datastructures are catamorphisms: the iteration of an algebra. We give some generic definitions to work with algebras. These technique are quite familiar to Haskell programmers [MFP91, Ven00, BdM97, BH12]. We work in the category of Haskell types and functions. An inductive datastructure is the least fixed-point of a base functor. We provide a generic view for fixed-points with a type class `Fix f t`, that says `f` is the *base functor* of `t`. It has two methods which are isomorphisms. The class is suitable both for inductive and coinductive type. In the former case, `inn` is the initial algebra and `out` its inverse. in the later case, `out` is the final coalgebra and `inn` its inverse.

```
class Functor f ⇒ Fix f t | t → f where
  out :: t → f t
  inn :: f t → t
```

The base functor of `Tree` is:

```
data TreeF r = LeafF Int | ForkF r r deriving Functor
instance Fix TreeF Tree where
  out (Leaf x)    = LeafF x
  out (Fork l r)  = ForkF l r
  inn (LeafF x)   = Leaf x
  inn (ForkF l r) = Fork l r
```

A `f`-algebra captures the body of a catamorphism, it is an inductive step: `f t` contains the results of recursive calls on the immediate subterms.

```
type Alg f t = f t → t
```

Catamorphisms are the iteration of f-algebras: their existence and unicity is ensured for the least fixed-points of f.

```
cata :: (Fix f t) ⇒ Alg f r → t → r
cata alg = phi
  where  phi = alg . fmap phi . out
```

The traversals of our running example can be expressed as iterated algebras:

```
minAlg (LeafF x)   = x
minAlg (ForkF l r) = min l r

mapTreeAlg f (LeafF x)   = Leaf (f x)
mapTreeAlg f (ForkF l r) = Fork l r

replaceAlg y = mapTreeAlg (const y)
```

Extending the Abstract Interface

As motivated at the beginning of §6.2 we want the library interface to be abstract, so we provide a type-class with all the necessary combinators as methods.

The solution is to extend the Arrow abstraction with a primitive to build single traversals. Keeping in mind that Env must be an instance of this new class, it follows that the datastructure on which the traversal is performed must be implicit. The class is parameterised with the base functor of the datastructure and the arrow type.

```
class (Arrow (⇝), Functor f) ⇒ ArrowCata f (⇝) | (⇝) → f where
  cataA :: Alg f x ⇝ x
```

The environment instance defines cataA directly with cata.

```
instance (Fix f t) ⇒ ArrowCata f (Env t) where
  cataA = Env $ flip cata
```

Now our running example can be defined solely with the ArrowCata abstraction:

```
minA :: ArrowCata TreeF (⇝) ⇒ () ⇝ Int
minA = proc () → cataA ≺ minAlg

replaceA :: ArrowCata TreeF (⇝) ⇒ Int ⇝ Tree
replaceA = proc y → cataA ≺ replaceAlg y

repMinA :: ArrowCata TreeF (⇝) ⇒ () ⇝ Tree
repMinA = proc () → do
  m ← minA ≺ ()
  replaceA ≺ m
```

The multi-traversal definition of repMin is operationally equivalent to repMinA when we run it with the environment arrow: repMin t == runEnv repMinA t ().

Polymorphic encoding of inductive datatypes The class `ArrowCata` generalises an encoding of inductive datatypes variously called Church, Girard, system-F, or polymorphic encoding [BB85, GTL89, Wra89]. The least fixed-point of functor `f` is encoded as the polymorphic type: `∀ x . Alg f x → x`. The bijection is given in one direction by partially applying the `cata` function to the datastructure, and in the other by applying the polymorphic function to the initial algebra.

`ArrowCata` generalises the encoding by making the function type an arbitrary arrow. We can always retrieve the implicit datastructure of an `ArrowCata` by applying `cataA` to the initial algebra.

6.3. Circular Implementation

In the previous section, we defined the type-class `ArrowCata f (⇝)` which is an abstract interface whose primitives can define complex traversals over the same datastructure, implicit in the arrow (⇝), and whose base-functor is `f`. We also gave an implementation of this interface: the instance `ArrowCata f (Env t)`, where `t` is the fixpoint of `f`, given by the instance `Fix f t`. When executing an `ArrowCata` program with this instance, every use of `cataA` corresponds to a distinct traversal of the datastructure.

Now we give a second implementation of the interface which is semantically equivalent to `Env`, but operationally different: it computes the result in a single traversal, using the circular technique behind the scene to combine the uses of `cataA`.

6.3.1. Recursive Pattern for Circular Traversals

Before we define a new instance for `ArrowCata`, we must step back and think about how circular traversals are formed, generalising the examples of Bird's article.

In the introduction, we defined a combined traversal `min_and_replace` by fusing the bodies of the `minTree` and `replace`. There is a generic operation on algebras that allow us to do this modularly. We can form the product of any two `f`-algebras.

```
pairAlg :: Functor f ⇒ Alg f i → Alg f j → Alg f (i,j)
pairAlg alpha beta = λy → (alpha (fmap fst y) , beta (fmap snd y))
```

Thus rather than iterating each algebra in turn (two traversals), we will iterate their product (one traversal). This is justified by the universality of the product and the initiality of algebras, from which follows the so-called *banana-split* law [BdM97]

```
(cata a, cata b) ≡ cata (pairAlg a b)
```

`min_and_replace` can be defined modularly using our generic combinators:

```
min_and_replace' :: Int → Tree → (Int, Tree)
minReplaceAlg :: Int → Alg TreeF (Int, Tree)

min_and_replace' y t = cata (minReplaceAlg y) t
minReplaceAlg y = minAlg `pairAlg` replaceAlg y
```

We capture the recursive pattern of circular traversals by abstracting over the body of the circular traversal. Looking at our introductory example:

```
repMinCirc tree = newtree
  where (minimum, newTree) = min_and_replace minimum tree
```

The body is a catamorphism which can be parameterised with its own result. The final result is extracted from the result of the catamorphism.

```
type CircBody f s r = s → (Alg f s , r)
```

In the example:

```
repMinBody :: CircBody TreeF (Int,Tree) Tree
repMinBody (minimum, newTree) = (minReplaceAlg minimum, newTree)
```

`f` is the base functor of the datastructure being traversed.

`s` is the carrier of the algebra; when many traversals are composed, we use `pairAlg` to combine them and `s` is the product of all their carriers. It appears as an argument because the algebra can circularly depend on its own result.

`r` is the result of the whole computation.

To run the circular traversal, we obtain the algebra and the result by calling `body` with the value returned by iterating the very same algebra with `cata`. Note that this circularity could be a cause of non-termination. In particular, `body` *must* be non-strict.

```
circFix :: Fix f t ⇒ CircBody f s r → t → r
circFix body structure = result
  where (alg, result) = body (cata alg structure)
```

Thus we obtain the circular version of `repMin`, which was modularly built using the algebras of two independent traversals (`minAlg` and `traverseAlg`) and the generic combinators `pairAlg` and `circFix`.

```
repMinCirc1 = circFix repMinBody
```

Unfolding the previous definition yields:

```
repMinCirc2 inputTree = resultTree
  where (alg, resultTree) = repMinBody (minimum, newTree)
        (minimum, newTree) = cata alg inputTree
```

Further expansion of `repMinBody` then `minReplaceAlg` yields:

```
repMinCirc3 inputTree = resultTree
  where (alg, resultTree) = (minAlg ‘pairAlg‘ replaceAlg minimum, newTree)
        (minimum, newTree) = cata alg inputTree
```

And since `cata (minAlg ‘pairAlg‘ replaceAlg minimum)` is equivalent to `min_-and_replace`, we obtain after simplification:

```
repMinCirc4 inputTree = newTree
  where (minimum, newTree) = min_and_replace minimum inputTree
```

which is exactly the circular program we gave in the introduction.

6.3.2. ArrowCata Instance

In the previous section, we defined `circFix` that already allow us to define circular traversals modularly by combining algebras with the ‘pairAlg‘ operator. However, using this `circFix` directly could lead to non-termination if we’re not careful when defining the `circBody` argument: there shouldn’t be a loop between the dependencies of the algebra constituents. Additionally, the carrier of the algebra becomes a possibly complicated nested tuple type after pairing many algebras which makes the definition of the `circBody` values cumbersome because we must keep track of the order in which the algebras are paired, and find the correct paths in the nested tuples to extract the wanted components.

To address the previous deficiencies, we define a new type around `CircBody` in order to implement the type-class `ArrowCata`. Not only the algebra carrier (complex nested tuple type) will be hidden by an existential quantification, but we also get the property that every circular traversal built with the `ArrowCata` primitives is well-behaved.

```
data Circ f a b = ∀ s . C {runC :: a → CircBody f s b}
runCirc :: Fix f t ⇒ Circ f a b → t → a → b
runCirc (C circ) t x = circFix (circ x) t
```

Confusingly existentials are defined using the keyword ∀ in GHC-Haskell. The previous definition corresponds to a data constructor of type:

```
C :: (∃ s . a → CircBody f s b) → Circ f a b
```

Type Class Homomorphisms

Note that `runCirc` almost computes an environment arrow, we just need to package its result in the newtype constructor `Env`.

```
circEnv :: Fix f t ⇒ Circ f a b → Env t a b
circEnv = Env . runCirc
```

`circEnv` gives the semantics of a circular traversals in terms of the modular traversals written as `Env` arrows. This is equivalent to say that `circEnv` is a `ArrowCata`-homomorphism, hence it must preserve each of the `arrowCata` methods:

```
circEnv id ≡ id
circEnv (g . f) ≡ circEnv g . circEnv f
circEnv (arr f) ≡ arr f
circEnv (first f) ≡ first (circEnv f)
circEnv cataA ≡ cataA
```

Those properties can be useful to derive the actual implementations of the methods. This approach was popularised by Elliot [Ell09]. He calls this approach *denotational design* and explains type class homomorphisms in the following words: *The instance’s meaning follows the meaning’s instance.*

Composition

The composition of two circular traversals builds the product of their respective algebras. Given:

```
f :: a → s → (alg fun s, b)
g :: b → t → (alg fun t, c)
```

we define: `compCirc f g :: a → (s,t) → (alg fun (s,t), c)`.

```
compCirc g f = λx ~(s,t) →
  let (f_alg, b) = f x s
      (g_alg, c) = g b t
   in (pairAlg f_alg g_alg, c)
```

The lazy pattern `~(s,t)` makes the function non-strict in that argument. This is necessary for `circFix` to converge.

Lifting non-traversals

`arr` lifts a function to a circular traversals that doesn't actually need to traverse anything to return a result, so we use the trivial (terminal) algebra. Given `f :: a → b`, we define: `arrCirc f :: a → () → (alg fun (), b)` as

```
arrCirc f = λx _ → (const (), f x)
```

Pairing

The method `first` should copy value from the argument to the result.

Given `f :: a → s → (alg fun s, b)`
we define `firstCirc f :: (a,c) → s → (alg fun s, (b,c))`

```
firstCirc f = λ (a, c) s →
  let (alg, b) = f a s
   in (alg, (b, c))
```

Catamorphisms

The `cataA` combinator is simply the pair constructor: since it maps an algebra and the result of iterating that algebra, to their pair:

```
(,) :: Alg fun s → s → (Alg fun s, s)
```

Class Instances

```
instance Functor f ⇒ Category (Circ f) where
  id = arr id
  C g . C f  = C $ compCirc g f

instance Functor f ⇒ Arrow (Circ f) where
  arr f = C $ arrCirc f
  first (C f) = C $ firstCirc f

instance Functor f ⇒ ArrowCata f (Circ f) where
  cataA = C (,)
```

Example

The previous definition of `repMinA` (p. 81) being polymorphic in the `ArrowCata` can be run as a single circular traversal with `runCirc` since `Circ` is now an instance of `ArrowCata`.

```
repMinCirc t ≡ runCirc repMinA t ()
```

6.3.3. Proving the Homomorphism Properties

The type-class homomorphism properties given above can all be verified. They entail an important result: The interpretation of `ArrowCata` programs with the `Env` and `Circ` arrows are semantically equivalent. A corollary is that circular programs written using `ArrowCata` primitives won't be non-terminating unless the non-circular program is non-terminating as well: the circularity isn't the cause of non-termination, but rather one of the algebras.

In all the proofs we use the following equality:

```
circEnv $ C f
  ≡ Env $ runCirc $ C f
  ≡ Env $ λt x → circFix (f x) t
  ≡ Env $ λt x → let (a,r) = f x (cata a t) in r
```

We prove: `circEnv (arr f) == arr f`

```
circEnv (arr f)
  ≡ circEnv (C (arrCirc f))
  ≡ Env $ λt x → let (a,r) = arrCirc f x (cata a t) in r
  ≡ Env $ λt x → let (a,r) = (const (const (), f x)) (cata a t) in r
  ≡ Env $ λt x → let (a,r) = (const (), f x) in r
  ≡ Env $ λt x → f x
  ≡ Env $ const f
  ≡ arr f
```

Using the previous result, we prove: `circEnv id == id`

```
circEnv id
  ≡ circEnv (arr id)
  ≡ arr id
  ≡ id
```

We prove: `circEnv (C g . C f) == circEnv g . circEnv f`

```
circEnv (C g . C f)
  ≡ circEnv $ C $ compCirc g f
  ≡ Env $ λt x → let (a,r) = compCirc g f x (cata a t) in r
  ≡ Env $ λt x → let (f_a, y) = f x (fst p)
                     (g_a, z) = g y (snd p)
                     p        = cata (pairAlg f_a g_a) t
                 in z
  { cata (pairAlg f_a g_a) t ≡ (cata f_a t, cata g_a t) }        -- banana-split

  ≡ Env $ λt x → let (f_a, y) = f x (cata f_a t)
                     (g_a, z) = g y (cata g_a t)
                 in z
  ≡ Env $ λt x → circFix (g (circFix (f x) t)) t
  ≡ Env $ λt → runCirc (C g) t . runCirc (C f) t
  ≡ Env (runCirc (C g)) . Env (runCirc (C f))
  ≡ circEnv g . circEnv f
```

We prove: `circEnv (first f) == first (circEnv f)`

```
circEnv $ first $ C f
  ≡ circEnv $ C $ firstCirc f
  ≡ Env $ λt x → let (a,r) = firstCirc f x (cata a t) in r
  ≡ Env $ λt (y,z) → let (a,r) = let (a',b) = f y (cata a t)
                                  in (a', (b,z))
                     in r
  ≡ Env $ λt (y,z) → let (a, b) = f y (cata a t) in (b,z)
  ≡ Env $ λt (y,z) → (let (a, b) = f y (cata a t) in b, z)
  ≡ Env $ λt (y,z) → (circFix (f y) t, z)
  ≡ Env $ λt (y,z) → (runCirc (C f) t y, z)
  ≡ first $ Env $ runCirc $ C f
  ≡ first $ circEnv $ C f
```

We prove: `circEnv cataA == cataA`

```
circEnv cataA
  ≡ circEnv (C (,))
  ≡ Env $ λt x → let (a,r) = (,) x (cata a t) in r
  ≡ Env $ λt x → cata x t
  ≡ cataA
```

6.3.4. Kleisli Arrow for an Indexed Monad

If it weren't for the existential in the definition of `Circ f a b`, it would have the same expression as a Kleisli arrow, prompting us to ask whether `CircBody f s b` has a monadic structure. In fact it does: it is an indexed monad, with `s` being the index.

Definition 6.3.1 (Indexed Monad). *An indexed monad M on a category C, is given by a monoid S with unit ε and multiplication $\star$, a S-indexed family of endofunctors $M : S \to C^C$, a natural transformation from the identity functor $\eta : I \to M_\varepsilon$ and a S^2 indexed family of natural transformations: $\mu_{i,j} : M_i \circ M_j \to M_{i\star j}$ verifying the following commutative diagrams for all i, j and k.*

$$\begin{tikzcd}
M_i \arrow[r, "M_i\eta"] \arrow[d, "\eta M_i"'] \arrow[dr, "id"] & M_i M_\varepsilon \arrow[d, "\mu_{i,\varepsilon}"] \\
M_\varepsilon M_i \arrow[r, "\mu_{\varepsilon,i}"'] & M_i
\end{tikzcd}
\qquad
\begin{tikzcd}
M_i M_j M_k \arrow[r, "M_i\mu_{j,k}"] \arrow[d, "\mu_{i,j} M_k"'] & M_i M_{j\star k} \arrow[d, "\mu_{i,j\star k}"] \\
M_{i\star j} M_k \arrow[r, "\mu_{i\star j,k}"'] & M_{i\star j\star k}
\end{tikzcd}$$

To properly implement indexed monads we need dependent types. In the indexed monad `CircBody`, the monoid is the free monoid that we can approximate in Haskell with nested tuple types: `()` :: * is the unit, `(,)` :: * → * → * is the multiplication, and we must consider isomorphic types `(a,(b,c))` and `((a,b),c)` as equal, as well as `((),a)`, `(a,())` and `a`.

Arrows are Preferable over Indexed Monads for Circular Programs
Although we could think of writing circular programs monadically, it would have the disadvantage of making the index visible. We'd rather hide it because the semantics of the indexed monad `CircBody` doesn't depend on the index, thus we can write two semantically equivalent `CircBody` with different indices, but we cannot substitute one for the other since they have different indices. On the other hand the arrow `Circ` hide the indices, so the substitution is possible.

6.4. Example: List of Deviations

To conclude the section we give another example of using our programming interface. Given a finite list of real numbers, we compute their deviations which is their difference with the mean. This is the motivating example of "Why attribute grammars matter" [Swi05]. We can write this function by composing three traversals to compute the sum, the length, and the deviations.

```
deviations xs =
 let mean = sum xs / fromIntegral (length xs)
  in map (λx → x - mean) xs
```

Before we can write this program as a `ArrowCata`, we must first define the algebras of each traversal. Therefore we define the base functor of finite lists, and its `Fix` instance:

```
data ListF a x = Nil | Cons a x deriving Functor
instance Fix (ListF a) [a] where
  out []         = Nil
  out (h : t)    = Cons h t
  inn Nil        = []
  inn (Cons h t) = h : t
```

Since defining the algebra requires always the same case analysis, we define a combinator for that:

```
listAlg :: b → (a → b → b) → Alg (ListF a) b
listAlg n c Nil = n
listAlg n c (Cons h t) = c h t
```

The algebras are straightforward.

```
sumAlg    :: Num a ⇒ Alg (ListF a) a
lengthAlg :: Num b ⇒ Alg (ListF a) b
mapAlg    :: (a → b) → Alg (ListF a) [b]
```

```
sumAlg    = listAlg 0 (+)
lengthAlg = listAlg 0 ((+) . const 1)
mapAlg f  = listAlg [] ((:) . f)
```

Each algebra gives raise to a `ArrowCata` primitive traversal.

```
sumA    :: (ArrowCata (ListF a) t, Num a) ⇒ t () a
lengthA :: (ArrowCata (ListF a) t, Num a) ⇒ t () a
mapA    :: ArrowCata (ListF a) t          ⇒ t (a → b) [b]

sumA    = proc () → cataA ≺ sumAlg
lengthA = proc () → cataA ≺ lengthAlg
mapA    = proc f  → cataA ≺ mapAlg f
```

Finally we compute the deviations with the help of the previous combinators and the arrow notation.

```
deviationsA :: (ArrowCata (ListF b) t, Fractional b) ⇒ t () [b]
deviationsA = proc () → do
  sum ← sumA ≺ ()
  len ← lengthA ≺ ()
  let mean = sum / len
  mapA ≺ (λx → x - mean)
```

Avoiding Intermediate Bindings The bindings for `sum` and `len` might seem notationally heavy. To avoid them, we can compute the mean directly from `sumA` and `lengthA` if we define a new arrow combinator `liftArr2`. Noting that the `Applicative` [MP08] instance of `(→ x)` can be generalised to any arrow:

```
constArr :: Arrow t ⇒ b → t x b
appArr :: Arrow t ⇒ t x (a → b) → t x a → t x b

constArr = arr . const
appArr f g = proc x → do
  y ← f ≺ x
  z ← g ≺ x
  returnA ≺ y z
```

Using the two applicative primitive `constArr` and `appArr` we are able to lift functions of any arities to act upon the codomain of arrows having the same domain:

```
liftArr :: Arrow t ⇒ (a → b) → t x a → t x b
liftArr2 :: Arrow t ⇒ (a → b → c) → t x a → t x b → t x c
liftArr3 :: Arrow t ⇒
  (a → b → c → d) → t x a → t x b → t x c → t x d

liftArr  f a     = constArr f ‘appArr‘ a
liftArr2 f a b   = liftArr f a ‘appArr‘ b
liftArr3 f a b c = liftArr2 f a b ‘appArr‘ c
```

We can now inline the computation of the mean to simplify `deviationsA`.

```
deviationsA' = proc () → do
  mean ← liftArr2 (/) sumA lengthA ≺ ()
  mapA ≺ (λx → x - mean)
```

This program can be run either within `Env` or within `Circ` arrows, depending whether we want the computation in multiple independent traversal or in a single but circular traversal.

Chapter 7.

Circular Traversals Using Attribute Grammars

The ArrowCata interface relies on explicit algebra iteration to define primitive traversals. The Circ instance combine the algebras and most of the time iterating the resulting algebra produces a single traversal. Unfortunately, in some cases it does not. This happens when the carrier of the algebra is a function type. We address this problem by considering another way of defining traversals.

Overview The chapter proceeds as follows: §7.1 introduces an example for which the Circ implementation fails to produce a single traversal, we give a modular solution based an alternative traversal primitive related to attribute grammars; §7.3 defines generic attribute grammar traversals, with a strongly typed implementation using containers; §7.4 extends ArrowCata using the new traversal representation, and gives a circular implementation which solves the previous problem modularly; §7.5 concludes the section with some examples.

7.1. Palindrome

7.1.1. The Problem

This example shows both the limits of Bird's approach and of ArrowCata. It motivates a new representation of traversals instead of algebras. After exposing the problem, we'll give Bird's solution. His solution isn't easily generalisable because of an ad-hoc step he took and we'll fix that issue. Then we'll study the problem using ArrowCata and the Circ interpretation. We'll see that it fails its purpose of evaluating in a single traversal. We'll seek to understand why and then propose a new traversal primitive on which to base our circular combinators.

A palindrome is a word that is invariant by reversing the order of the letters: it reads the same left to right and right to left. Accordingly we define a function that computes whether a list is equal to its reversal.

```
palindrome :: (Eq a) ⇒ [a] → Bool
palindrome x = x ≡ reverse x
```

This function does two traversals: one to compute whether two lists are equal (== is redefined as eqlist below), and one to compute the reversal of the list.

```
eqlist [] [] = True
eqlist (x:xs) (y:ys) = x ≡ y && eqlist xs ys
eqlist _ _ = False

reverse xs = revcat xs []
revcat [] ys = ys
revcat (x:xs) ys = revcat xs (x:ys)
```

We want to implement `palindrome` as a single traversal using Bird's circular technique.

The palindrome example was used by Bird [Bir84, §3] to illustrate that termination of circular programs is a delicate issue. In fact the circular solution that results from the systematic transformation doesn't terminate and Bird shows how to fix it. Unfortunately his solution is very specific to this example. Looking at his approach there doesn't seem to be a general way to design a single-traversal correctly yet alone to define it modularly. It should therefore be instructive to study the `ArrowCata` implementation of `palindrome`: does it actually capture the single-traversal circular solution? Does it terminate?

Note that by testing equality up to the middle of the list we would avoid many redundant tests. It would also require to compute the length of the list which is an additional traversal. However Bird didn't study this case, and we chose to reproduce his exact approach. The less redundant version is given in §7.5.2.

7.1.2. Bird's Solution

Following Bird's approach, we first take the product of `eqlist` and `revcat`:

```
eqrev x y z = (eqlist x y, revcat x z)
```

so that palindrome can be written circularly (but still with two traversals):

```
palindrome1 x = eq where (eq, rev) = eqrev x rev []
```

The next step in Bird's approach is to express `eqrev` as a single traversal. This is done by repeatedly inlining and abstracting steps:

```
eqrev1 []     []     zs = (True, zs)
eqrev1 (x:xs) (y:ys) zs = (x ≡ y && t, r)
  where (t,r) = eqrev1 xs ys (x:zs)
```

Note the definition is partial, the missing cases are `eqrev1 [] (y:ys) zs` and `eqrev1 (x:xs) [] zs`. Bird dismissed them on account that `eqrev1` is only ever used when the first two arguments have the same length. We will come back to this point after exposing Bird's method.

First let it be noted that `eqrev1` traverses its first argument once, which is what we wanted to achieve. Second, it is totally defined when the first two arguments have the same length. Third, it is not semantically equal to `eqrev` since it is now partial. Fourth, the function is strict in its second argument whereas `eqrev` wasn't, which is another reason for semantic difference, this is in fact a fundamental difference because we can only define palindrome circularly if eqrev1 is non-strict in its second argument. Thus we merged two traversals, but we lost the ability to use the function circularly. The solution is to avoid pattern matching on the second argument:

```
eqrev2 []       ys zs  =  (True, zs)
eqrev2 (x : xs) ys zs  =  (x ≡ head ys && t, r)
  where (t,r) = eqrev2 xs (tail ys) (x : zs)
```

Now the circular palindrome can be defined with the single traversal eqrev2:

```
palindrome2 x = eq  where (eq,rev) = eqrev2 x rev []
```

Note that eqrev2 still isn't semantically equal to eqrev, since eqrev2 diverges whenever the second argument is longer than the first. However, it doesn't matter when used to define palindrome. This aspect of the transformation is ad-hoc and against modularisation since some of the functionality of eqlist was lost when fusing it with revcat. What if we complete eqrev1 with the missing equations?

```
eqrev1 []     (y:ys) zs = (False, zs)
eqrev1 (x:xs) []     zs = (False, r)
  where (t,r) = eqrev1 xs [] (x:zs)
```

Then we cannot even make eqrev1 non-strict in its second argument anymore. Thus it is rather a coincidence that we can define a partial eqrev that works for us. The solution is ad-hoc and it is not at all obvious how to generalise this example.

7.1.3. Fixing Bird's Solution

Bird's eqrev2 relied on the assumption that both lists compared for equality had the same length. In order to devise a general programming pattern, we cannot make such assumptions. Can we define eqrev4 as a single traversal that is semantically equal to eqrev under the constraint that eqrev4 must be non-strict in its second argument? This is the constraint that pushed Bird to depart from the semantics of eqlist.

eqrev can in fact be defined as a single traversal, non-strict in the second argument by postponing the case analysis (this remark was made previously [CGK99]).

```
eqrev3 [] ys zs = (null ys, zs)
eqrev3 (x:xs) ys zs = (eq_xxs, rev)
  where (eq_xs, rev) = eqrev3 xs (safe_tail ys) (x:zs)
        eq_xxs = case ys of
          [] → False
          (y : ys') → x ≡ y && eq_xs


safe_tail [] = []
safe_tail (h:t) = t
```

Notice however that ys is pattern matched twice, meaning the second argument list as a whole is consumed twice. We can again use circularity to share the pattern matching to compute both safe_tail and eq_xxs at the same time. The circularity between eq_xs and tail_ys doesn't create a loop during evaluation because eqrev4 is non-strict in its second argument and the case expression is non-strict in eq_xs.

```
eqrev4 [] ys zs = (null ys, zs)
eqrev4 (x:xs) ys zs = (eq_xxs, rev)
  where (eq_xs, rev) = eqrev4 xs tail_ys (x:zs)
        (eq_xxs, tail_ys) = case ys of
          [] → (False, [])
          (y : ys') → (x ≡ y && eq_xs, ys')
```

Note that we have two levels of circularity now: one in the definition of eqrev4 and one in the definition of palindrome4:

```
palindrome4 x = eq where (eq,rev) = eqrev4 x rev []
```

Let's unfold the recursion in palindrome4 [a,b,c] to see how it works.

```
palindrome4 [a,b,c] = q1
 where
   (q1, r1) = case r of                               -- eqrev4 [a,b,c] r []
              [] → (False, [])
              (y : ys') → (a ≡ y && q2, ys')
   (q2, r2) = case r1 of                              -- eqrev4 [b,c] r1 [a]
              [] → (False, [])
              (y : ys') → (b ≡ y && q3, ys')
   (q3, r3) = case r2 of                              -- eqrev4 [c] r2 [b,a]
              [] → (False, [])
              (y : ys') → (c ≡ y && q4, ys')
   (q4, r) = (null r3, [c,b,a])                      -- eqrev4 [] r3 [c,b,a]
```

Notice how in each definition (q_k, r_k) depends on (q_{k+1}, r_{k-1}), as if when we traverse the input list, each q_k is computed bottom up and r_k top down. This is in fact what would be called synthesized and inherited attributes in the attribute grammar terminology. The connection with attribute grammars will become clear later in this section and in §7.4.

For each input element a new thunk is created in which the case expression cannot be reduced until the reverse list is computed. The whole input list is read before we can reduce the first case expression. Then each subsequent case expression becomes reducible.

```
palindrome4 [a,b,c] = q1
 where
   (q1, r1) = (a ≡ c && q2, [b,a])
   (q2, r2) = (b ≡ b && q3, [a])
   (q3, r3) = (c ≡ a && q4, [])
   (q4, r)  = (True, [c,b,a])
```

Equivalently without the superfluous bindings.

```
palindrome4 [a,b,c] = a ≡ c  &&  b ≡ b  &&  c ≡ a  &&  True
```

To avoid the redundant tests we would need to compare only half of the list. See §7.5.2 for an implementation.

7.1.4. Palindrome with ArrowCata

When combining multiple traversals with `ArrowCata`, we must express the simple traversals as algebras. Both algebras are higher order: their carriers are function types. Thus the iteration of the algebra computes a partial application.

The algebra for `revcat` traverses the list left to right and builds a Hughes list [GH09] right to left.

```
revcatAlg :: Alg (ListF a) ([a] → [a])
revcatAlg Nil zs  =  zs
revcatAlg (Cons x revcat_xs) zs  =  revcat_xs (x:zs)
```

Although the definition of `eqlist` by pattern matching was symmetrical in the way both arguments where recursed upon, when we define the underlying algebra, only one of the arguments drives the recursion and is the focus of the inductive definition. The catamorphism computes the partial application of equality: `eqlist xs`, thus a list predicate `[a] → Bool` that computes whether the list is equal to `xs`.

```
eqlistAlg :: Eq a ⇒ Alg (ListF a) ([a] → Bool)
eqlistAlg Nil [] = True
eqlistAlg (Cons x eqlist_xs) (y:ys) = x ≡ y && eqlist_xs ys
eqlistAlg _ _ = False
```

Now that we have expressed the traversals as algebras, we may combine them using our primitives to define `palindrome`:

```
palindromeA = proc () → do
  rev ← cataA ≺ revcatAlg
  eq ← cataA ≺ eqlistAlg
  returnA ≺ eq (rev [])
```

As expected we can run the program in the `Env` and `Circ` arrows:

```
runEnv  palindromeA [1,2,3] ()  ⟹  False
runCirc palindromeA [1,2,1] ()  ⟹  True
```

Unlike when using Bird's technique, we do not have to worry about termination since we proved in §6.3.3 that `Circ` and `Env` are semantically equivalent interpretations of a `ArrowCata` program. And since the `Env` interpretation is operationally equivalent to the original multiple traversal `palindrome` which terminates, so does the `Circ` interpretation.

Unfortunately, even the `Circ` interpretation runs two traversals. That's because when we evaluate `eq (rev [])` both `rev` and `eq` have the list stored in their closure and traverse it to compute their result. The program isn't even circular: if we inline the definitions, `runCirc palindromeA` is operationally equivalent to:

```
palindrome3 x =
  let (eq,rev) = cata (pairAlg eqlistAlg revcatAlg) x
  in eq (rev [])
```

7.1.5. Pairing Higher-Order Algebras

Since our approach for modular circular programs is to combine basic traversals, we now wonder if algebras are suitable: can we combine `eqlistAlg`, `revcatAlg` and obtain an algebra operationally equivalent to the one underlying the single traversal `eqrev4` defined earlier?

The algebra of `eqrev4` has for carrier `[a] → [a] → (Bool, [a])` whereas the algebra underlying `runCirc palindromeA`, being the product of `eqlistAlg` and `revcatAlg` has for carrier the product of their carriers, namely `([a] → Bool, [a] → [a])`. That product algebra computes a pair of partial applications that keep the whole input datastructure in their closure environment to re-traverse it when applied to the remaining argument. We shall seek to merge them, to obtain an algebra equivalent to `eqrev4`. For this we would need a new operation to combine higher-order algebras:

```
hpair :: Functor f ⇒
  Alg f (a → b) → Alg f (c → d) → Alg f ((a, c) → (b,d))
```

It would need to verify that:

```
(1) cata (hpair g h) x (a,c) ≡ (cata g x a, cata h x c)
```

We can define such a function:

```
hpair u v x (a,c) = (u (fmap (pr1 c) x) a, v (fmap (pr2 a) x) c)

pr1 :: c → ((a, c) → (b,d)) → a → b
pr2 :: a → ((a, c) → (b,d)) → c → d
pr1 c f a = fst $ f (a,c)
pr2 a f c = snd $ f (a,c)
```

The reader can verify (1). It is a consequence that `pr1 c` and `pr2 a` are algebra morphisms. Unfortunately even `cata (hpair eqlistAlg revcatAlg)` does two traversals. This is because the recursive calls are computed independently for each component of the pair in `fmap (pr1 c) x` and `fmap (pr2 a) x` as can be seen when we simplify `eqrev5 = cata (hpair eqlistAlg revcatAlg)`:

```
eqrev5 Nil (y, z)   ⟹  (null y, z)
eqrev5 (Cons h t) (y, z)
  ⟹  case y of
        [] → (False, snd (eqrev5 t ([], h : z)))
        (h':t') → ( h ≡ h' && fst (eqrev5 t (t', z))       -- one recursive call
                  , snd (eqrev5 t (y, h : z)))           -- another recursive call
```

7.1.6. A New Traversal Primitive

In the previous section, we failed to combine the two higher-order algebras `eqlistAlg` and `revcatAlg` to obtain an algebra for `eqrev4` which does a single traversal. The operator that we designed has the right type but duplicates the recursive calls. I conjecture that *it is not possible to solve this problem with algebras.* In order to express **eqrev4** modularly, we must have a control over the way higher-order algebras use the function they recursively compute. A higher-order list-algebra is isomorphic to

```
data HOListAlg a i s = HOListAlg (i → s) (a → (i → s) → (i → s))
```

The first component is the function associated with an empty list and the second component computes the function associated with a non-empty list from its head and the function associated with its tail. Let's consider this second case: we're defining:

```
algCons :: a → (i → s) → (i → s)
algCons a f i = ...
```

We're interested in characterising how `algCons` will use the function `f`. That can only be by applying `f` to some `x :: i` and using `f x :: s` to compute the final `s` result of `algCons a f i`. This can be specified with a function of this type:

```
algConsSpec :: a → i → (i, s → s)
```

We define an algebra from that specification:

```
algCons a f i = result (f x)
  where (x, result) = algConsSpec a f i
```

Without loss of generality, we can "pull" the `s` argument out of the pair, this will allow the `i` of the result to depend circularly on it.

```
algConsSpec2 :: a → s → i → (s,i)

algCons x f i = s
  where (s,i') = algConsSpec2 x (f i') i
```

Now we can add the case for the empty list, this gives us a full specification of a higher-order algebra. Since that specification also provides a direct explanation of the computations of an attribute grammar, we call the type `ListAG`.

```
data ListAG a i s = ListAG (i → s) (a → s → i → (s, i))
```

We turn a list attribute grammar to a list algebra just as above:

```
algListAG :: ListAG a i s → Alg (ListF a) (i → s)
algListAG (ListAG nil cons) Nil = nil
algListAG (ListAG nil cons) (Cons h t) = λ i →
  let (s, ti) = cons h (t ti) i in s
```

The two algebras of palindrome are easily specified as attribute grammars. We spend some time explaining the first one.

```
eqlistLAG :: Eq a ⇒ ListAG a [a] Bool
eqlistLAG = ListAG null eqlistConsAG
```

Only the case for non-empty list is interesting:

```
eqlistConsAG x eq_xs [] = (False, [])
eqlistConsAG x eq_xs (y:ys) = (x ≡ y && eq_xs, ys)
```

Let us verify that running that specification yields the algebra that we expect:

```
algListAG eqlistLAG Nil ys ≡ null ys
algListAG eqlistLAG (Cons x eq_xs) [] ≡
  let (s, ti) = eqlistConsAG x (eq_xs ti) [] in s
≡
  let (s, ti) = (False, []) in s
≡
  False

algListAG eqlistLAG (Cons x eq_xs) (y:ys) ≡
  let (s, ti) = eqlistConsAG x (eq_xs ti) (y:ys) in s
≡
  let (s, ti) = (x ≡ y && eq_xs ti, ys) in s
≡
  let (s, ti) = (x ≡ y && eq_xs ys, ys) in s
≡
  x ≡ y && eq_xs ys
```

The AG implementation for `revcat` is not more difficult, we let the reader verify for himself that `algListAG revAG == revcatAlg`

```
revcatLAG :: ListAG a [a] [a]
revcatLAG = ListAG id (λx rev_xs zs → (rev_xs, x:zs))
```

Now thanks to the direct insight on the use of higher-order recursive arguments we can pair two attribute grammars in a single traversal, it's almost a product, if not for the transposition of the results.

```
pairListAG :: ListAG a i s → ListAG a i' s' → ListAG a (i,i') (s,s')
pairListAG (ListAG n1 c1) (ListAG n2 c2) = ListAG n c
 where
   n ~(i1,i2) = (n1 i1, n2 i2)
   c x (s1,s2) ~(i1,i2) = transpose (c1 x s1 i1, c2 x s2 i2)

transpose (~(a,b),~(c,d)) = ((a,c),(b,d))
```

Remark: lazy patterns on inherited attribute are essential since we want `eqrev` to be lazy on its second argument. Our final implementation of `palindrome` is modularly defined by pairing two independent traversals specified as attribute grammars.

```
eqrev5 = cata $ algListAG $ pairListAG eqlistLAG revcatLAG
palindrome5 x = fst p  where p = eqrev5 x (snd p, [])
```

7.2. Attribute Grammars

Algebras characterise a tree traversal by defining the inductive step: how to compute the value for a whole tree from the values recursively computed for each subtree. We say the value is synthesized, or computed bottom-up.

Attribute grammars (AG) complement this by allowing some parameters to be propagated to the subtrees from the root, thus computed top-down. They are said *inherited* in AG terminology. As we illustrated in the previous section, the algebras associated with AG are higher-order and compute a function from inherited to synthesized attributes. What makes them interesting for circular programming is that we can merge independent AG to run both at the same time in a single traversal. This was not possible with higher-order algebras. Since they offer more modularity we will use AG instead of algebras as the basis of our circular traversal abstraction. Prior to implement a new `Circ` datatype based on AG in §7.4, we shall first study the generalisation of `ListAG` given in the previous section, and define AG generically for any recursive datatype.

7.2.1. Attribute Grammar Implementations

AG systems (compiler compilers, and other tools) are defined around a rich AG language where attributes and rules to compute them are named objects. However, the core computation of an AG can be characterised abstractly in a functional language. Such an abstraction may in turn form the base of a full-featured AG embedded language. Such implementations are called *first-class* because the attribute grammars are data objects that can be combined, and eventually evaluated as a tree traversal. One major advantage of this approach is that new combinators can be defined as simple functions.

First Class AG

There are currently three implementations of first-class AGs to my knowledge. The first [MBS00] which we will explain shortly, doesn't capture important structural properties, and non-termination can result from careless usage from the programmer. The second [VSS09] is robust but relies on type families and many extensions to the type-class system. Furthermore, the genericity, the structural invariants and simplicity of the code all depend on the all important use of template Haskell. We discuss those systems in more detail in §8.5. The third is the one we give now. It may be the first time that a strongly typed characterisation of AG is given.

Zipper Based AG

There are also some embedding of AG in functional languages that are not first class: the zipper based approach for instance [UV05, MFS13]. Such embeddings define the computation of attributes as mutually recursive functions over a zipper [Hue97] view of the tree focusing on a particular node. The recursion is intrinsic to the definition of the attributes, unlike in first-class implementations in which the recursion is extrinsic and confined in the evaluator.

7.2.2. Generic AG Rules

Independently of the concrete language used to define an AG, we can characterise the computation of attributes in the following way. Each node of the tree is decorated with inherited and synthesized attributes by applying rules which are functions of attributes inherited from the parent and attribute synthesized by the children, to the inherited attributes of the children and the synthesized attribute of the parent.

Weakly Typed AG Implementation

The previous description of attribute rules is illustrated by the following type very similar to the one given in [MBS00]:

```
type SimpleAG c i s = c → (i, [s]) → (s, [i])
```

`c` standing for constructor, is a label identifying the particular node on which the rule is applied. The lists `[s]` and `[i]` are the collections of children's synthesized and inherited attributes.

Unfortunately, this representation relies on the assumption that the AG rules always compute a list of the same length as it was given. The generic view used here is that of a general tree type. Concrete datatypes must be converted to this tree in order to run an attribute grammar on them. `c` is a type of node labels:

```
data TreeG c = Node c [TreeG c]
```

For instance, a generic view for lists would be:

```
data ListC a = ConsC a | NilC                                    -- list constructors
listToTree :: [a] → TreeG (ListC a)
listToTree [] = Node NilC []
listToTree (x : xs) = Node (ConsC x) [listToTree xs]
```

Note how the type `ListC` parallels the definition of `List`: it has the same constructors, except that the children (recursive) are removed, all other fields (non-recursive) are kept. We call such a type, the type of shapes from the container terminology.

`listToTree` is a total injective function, however its inverse is partial since nothing ensures that the list of children has the right size.

The execution of the attribute grammar is a circular catamorphism.

```
runSimpleAG :: SimpleAG c i s → TreeG c → i → s
runSimpleAG g (Node c tt) i = s
  where (s, ii) = g c (i, ss)
        ss = zipWith (runSimpleAG g) tt ii
```

`tt` and `ii` are expected to have the same size but this invariant is not ensured by the type system.

Bijective Generic Tree Representation

We refine the generic representation above and make it bijective. We must ensure that the children collections have the required arity which depends on each constructor. This is achieved using GADTs by having the constructors indexed by their arity, as type level natural numbers. The `TreeW` type uses existential quantification (confusingly ∀ in GHC syntax).

```
data TreeW c = ∀ p . NodeW (c p) (p (TreeW c))
```

The constructors for lists are now indexed by the type of their children collections, one for `Cons` and none for `Nil`. `V0` and `V1` are vector types of zero and one element.

```
data V0 x = V0
data V1 x = V1 x
data ListS a p where
  ConsS :: a → ListS a V1
  NilS  :: ListS a V0
```

Lists are now in bijection with their generic tree representation.

```
listToTreeW :: [a] → TreeW (ListS a)
listToTreeW [] = NodeW NilS V0
listToTreeW (x : xs) = NodeW (ConsS x) (V1 $ listToTreeW xs)
```

Now converting from the generic representation is also a total function, since the exact number of children is known.

```
treeWToList :: TreeW (ListS a) → [a]
treeWToList (NodeW NilS V0) = []
treeWToList (NodeW (ConsS x) (V1 xs)) = x : treeWToList xs
```

Strongly Typed AG Implementation

With this generic implementation of trees, we can more precisely capture what an attribute grammar is:

```
type AG c i s = ∀ p . c p → (i, p s) → (s, p i)
```

Running an AG is exactly as before with `runSimpleAG`, it is simply a matter of generalising `zipWith` to `liftA2`:

```
runAG :: Shape c ⇒  AG c i s  →  TreeW c (i → s)
runAG g (NodeW c tt) i = s
  where (s , ii) = g c (i, ss)
        ss = liftA2 (runAG g) tt ii
```

`liftA2` is an operation on applicative functors. Just like `zipWith`, it applies `runAG g` component-wise to the vectors.

ListAG

We can show that `AG (ListS a) i s` is isomorphic to `ListAG a i s` of the previous section. Recall the definition:

```
data ListAG a i s = ListAG (i → s) (a → s → i → (s, i))

fromListAG :: ListAG a i s → AG (ListS a) i s
fromListAG (ListAG nil cons) NilS (i, V0) = (nil i, V0)
fromListAG (ListAG nil cons) (ConsS x) (i, V1 s) = (s', V1 i')
  where (s', i') = cons x s i

toListAG :: AG (ListS a) i s → ListAG a i s
toListAG g = ListAG (nil g) (cons g)
 where nil g i = fst $ g NilS (i, V0)
       cons g x s i = (s',i')
         where (s', V1 i') = g (ConsS x) (i, V1 s)
```

7.2.3. Attribute Grammar Systems

Attribute grammars (AG) have been suggested as a way to write circular programs modularly before [SAAS99, Swi05, MFS13, MBS00, VSS09]. Our approach is different. They use full featured attribute grammar language which is programming paradigm, whereas we work in the functional paradigm by composing `AG` values using arrows. Ours is a novel way to compose attribute grammars and is closer to the applicative style of functional programming.

Although our type `AG` characterises attribute grammars, there is a lot more to an attribute grammar system: named attributes, rule definitions, and various combinators that allow to define attribute grammars modularly. Those features are very desirable when writing compilers or other big project. In this chapter we do not study such a full AG system, however we give an overview of our work in this respect in §8.4. Rather, this chapter focuses on the composition of traversals expressed using the `AG` primitive.

7.3. Containers and W Types

It may not appear at first sight that our new generic view is based on W types, the dependent type of well-orderings, and closely related to the notion of *container*.

The type `TreeW` given previously corresponds to a well known type-theoretical construct: the `W` type, capturing well-orders. This section explains the type theoretical implementation of containers and well-orders, and relates them to their surprising implementation in Haskell with a novel insight that type families are the display maps of GADTs.

Containers [Abb03] give a semantic description of a data-structure that contains some elements. They have been used for generic programming [Mor07], as an alternative to the algebraic (syntactic) description of datatypes. Another use of containers, and the one that interests us in this chapter, is as a representation of strictly positive functors (see Definition 2.1.2).

7.3.1. Type-Theoretical Implementation

The important observation is that we can separate two concepts: the shape of the data-structure, corresponding to the bare structure with all the elements removed; and a mapping that can put back the elements in their original position in the structure. We formalise this idea in type-theory: We call container the couple (S, P) usually written $S \lhd P$ of a set of shapes S and a shape indexed family of positions $(P_s)_{s \in S}$. Using Agda syntax [Nor07],

```
Container : Set₁ where
  _◁_ : (S : Set) → (P : S → Set) → Container
```

The container `S ◁ P` is a code representing a parameterised type. Its semantics is given by its extension, written `⟦S ◁ P⟧ : Set → Set`, denoting a parametric type of data-structures. The elements of `⟦S ◁ P⟧ X` denote the data-structures which, as explained above are separated into a bare structure given by a shape `s : S`, and a payload function `f : P s → X` that restores the elements in their original positions, for each position in `P s`.

```
⟦S ◁ P⟧ X = Σ(s:S)(P s → X)
```

We will call the functors `⟦S ◁ P⟧`, *container functors*. Elements of the dependent sum `Σ(s:S)(P s → X)` are pairs `(s,f)` with `s : S` and `f : P s → X`. The extension is functorial in the element type `X`. Given a function `g : X → Y`, we can apply it to every element in the structure, the resulting structure has the same shape, we compose `g` with the payload function to get a new payload function.

```
⟦S ◁ P⟧ g = λ (s , f) → (s , g ∘ f)
```

Container functors capture exactly strictly positive functors (SPF). When writing generic programs over functors, we can either choose the syntactic representation of a SPF, or the semantic representation as a container [Mor07].

Well-Orders

The fixed-point of a container functor is a W type, it is a generic representation of an inductive datatype. This is what we use in our generic implementation of attribute grammars.

7.3.2. Haskell Implementation

Containers are dependent types. It is surprising that we can implement them in Haskell. The respective definitions are very different, how are they related?

Haskell with GHC extensions can emulate some dependent type constructs. There are a few different approaches, they rely on representing values as types using type constructors instead of value constructors, on type level functions, either with the logical paradigm offered by type classes [Hal00], or with open type families. With those extensions, GHC-Haskell is very similar to the language Omega [SL07].

The approach we will use here is different and novel. Using only GADTs, and the isomorphism between a display map and an indexed set, we can define a dependent type without representing the values as types.

Display Maps

Display maps are used in categorical semantics of type-theory to represent indexed sets [Tay99]. Sets like `A : Set` are object in the category and families `B : A → Set` are represented by a display map, also called a fibration: `f : E → A`. The idea is that `E` represents the union of all `B a`: `Σ(a : A)(B a)` and `f` is the first projection. An element `b : B a` is represented by an element `x : E` such that `f x = a`. Reciprocally, given a display map `f : E → A`, we associate a family `B : A → Set` defined by `B a = Σ(x : E)(f x = a)`.

Type Families are the Display Maps of GADTS

A GADT is a datatype family indexed by types. This is not a usual construction in type theories, where indices are normally values of closed types. This is in contrast with GADTs for which the set of indices is open, as we can always extend the set of types. We will best understand what a GADT is in type-theoretical terms by considering its display maps, although the index is a type rather than a value, so is an abuse of language to call this a display map.

Take a GADT with one type parameter `G : Set → Set`, Its display map would be `F : I → Set`, where `I = Σ(X:Set)(G X)`. So a GADT can in fact be viewed as a type family indexed by values: the set of indices is the union of all `G X`, and the type family `F` returns the `G` subset of the union in which its argument is an element: `∀ i ∈ I, i ∈ G (F i)`.

Reciprocally, given a type of indices `I`, any type family `F : I → Set` can be represented with a GADT `G : Set → Set`, such that for any set `X`, `G X` is the inverse image of `X` by `F`: `G X = { i : I | F i = X }`. We used the set comprehension because in GHC-Haskell, the proof term corresponding to the type equality is implicit, we cannot manipulate it.

Note that the other approaches using GADTs define indexed types directly with type indices representing values either by using singleton types [McB01a] or with the GHC extension `DataKind` which promotes every datatype to a kind and its constructors to types. On the other hand, by representing the display map as a GADT, the indices are implicit, but correspond to Haskell values.

Now that we have a representation of type families, we can represent dependent sums and products. `Σ (a : A)(B a)` is represented by:

```
data Dsum a where
  Dpair :: a b → b → Dsum a
```

`Π(a:A)(B a)` is represented by:

```
newtype Dprod a = Dprod (∀ b . a b → b)
```

Now we have everything we need to implement containers in Haskell: a container is defined by a GADT of shapes indexed by their positions.

The type of container extension takes a container `s` (GADT indexed by the positions) and an element type `x`, it is a dependent sum

```
infix 0 :<
data Ext s x where
  (:<) :: s p → (p → x) → Ext s x
```

Note that with efficiency in mind, we will be generalising the applicative functor $\Lambda x.p \rightarrow x$ to any applicative functor f. Since most of our examples will have finite positions, this allow us to use tuple types directly. Accessing the fields of a tuple are arguably faster than calling a function, with the advantages of data sharing and memoisation. Now the shapes are indexed by that applicative functor instead of the positions:

```
data Ext s x where
  (:<) :: s f → f x → Ext s x
```

Alternatively, we could have chosen to use type level natural numbers as shape indices and fix the applicative functor to a vector type:

```
data Ext s x where
  (:<) :: s n → Vec n x → Ext s x
```

Where `Vec` is a GADT indexed by type level natural numbers:

```
data Vec n x where
  VNil :: Vec Zero x
  VCons :: x → Vec n x → Vec (Succ n) x

data Zero
data Succ n
```

Such containers with finite positions are called *small container* [Mor07], or *decidable* containers [Abb03].

7.4. Circular Programs as Compositions of AGs

In §7.1 we saw that higher-order algebras couldn't be combined in a single traversal, and we gave a solution as attribute grammars (AG), but only for the special case of lists. We now give a generic implementation of AG and extend `ArrowCata` to use AG as primitive traversals. We also modify the type `Circ` to use AG instead of algebras.

Overview

We first define a generic view of recursive datatypes as fixed-point of a container functor (W type) §7.4.1, on which the generic AG implementation is based. AG datatype and operations are given next §7.4.2. Then the type-class `ArrowAG` extending `ArrowCata` is defined §7.4.3, it constitutes the programming interface for composing attribute grammars using the arrow notation. Finally, the circular implementation of `ArrowAG` is given §7.4.5.

7.4.1. Generic View as a W Type

For our first circular abstraction based on algebras in §6.2 and §6.3 the generic view for recursive datatypes was as fixed-point of a functor, and witnessed by the type-class `Fix f t` §6.2.2. For the new circular abstraction based on attribute grammars, we use an alternative generic view as a fixed-point of a container functor, also known as a W type.

W Types

W types are parameterised by a container which in Haskell are implemented as a position indexed type of shapes §7.3.2.

```
data W s where
  Sup :: s p → p (W s) → W s
```

Container Functor

The base functor of a W type is also known as the extension of the container:

```
data WF s x where
  WF :: s p → p x → WF s x
```

Payload Functors

The parameter `s` in `W` and `WF` is in fact a generalisation of a container and to be useful the indices `p` known as *payload functors* need to be applicative functors. This characterises the fact that if we have two trees of the same shape, although we don't know their children positions as in normal containers, we can nonetheless make the pairs of children in the same position.

We capture this with a type-class `Shape`:

```
class Shape s where
  pureS :: s p → x → p x
  appS  :: s p → p (x → y) → p x → p y
```

Applicative functors must also obey some laws [MP08].

For example, the list payloads are applicative, respectively the empty vector for `NilS` and the one element vector for `ConsS`.

```
instance Shape (ListS a) where
  pureS NilS = λcase{}
  appS (ConsS h) (V1 f) (V1 x) = V1 (f x)
```

Functoriality of the payload functor follows from any applicative instance:

```
mapS :: Shape s ⇒ s p → (x → y) → p x → p y
mapS sp f px = appS sp (pureS sp f) px
```

The functoriality of a container functor follows from the functoriality of its payload functors.

```
instance Shape s ⇒ Functor (WF s) where
  fmap f (WF s p) = WF s (mapS s f p)
```

Generic View as W Type

We view recursive types as fixed-points of a container functor. This is a refinement of the `Fix` generic view where we require the base functor to be a container functor.

```
class (Shape s, Fix (WF s) t) ⇒ FixW s t | t → s where      -- no new method
```

The view for lists:

```
instance Fix (WF (ListS a)) [a] where
  out [] = WF NilS V0
  out (h:t) = WF (ConsS h) (V1 t)
  inn (WF NilS V0) = []
  inn (WF (ConsS h) (V1 t)) = h:t

instance FixW (ListS a) [a] where
```

Note that because of the type dependency, only one base functor is allowed (one instance of `Fix f t` for any type `t`). Therefore, all the previous examples using the instance `Fix (ListF a) [a]` cannot be compiled together with the new container view of lists: `FixW (ListS a) [a]`. It is in fact possible to design a type-class hierarchy where more than one base functor is used but we chose to omit those complications.

Container Algebras

Existential types like `WF` are eliminated with a polymorphic function. `AlgW s x` corresponds to the uncurried version of `Alg (WF s) x`.

```
type AlgW s x = ∀ p . s p → p x → x

uncurryW :: AlgW s x → Alg (WF s) x
curryW :: Alg (WF s) x → AlgW s x

uncurryW f (WF s p) = f s p
curryW f s p = f (WF s p)
```

A generic iteration on container algebras:

```
cataW :: (FixW s t) ⇒ AlgW s r → t → r
cataW = cata . uncurryW
```

7.4.2. Generic Attribute Grammar Traversals

The computation of an attribute grammar can be described by the type `AG f i s` where `f` is the type of node shapes of the tree, indexed by the payload functors of children collections.

```
newtype AG f i s = AG {appAG :: ∀ p . f p → (i , p s) → (s , p i)}
```

Let us breakdown the type:

```
newtype AG f i s = AG {appAG ::
∀ p .   – collection type, for the children of the current node
f p →   – node shape (grammar production)
          also containing the non-recursive data (the terminals)
(i,     – inherited attribute from the parent
 p s) → children synthesized attributes
(s,     – synthesized attribute for this node
 p i)}  – children inherited attribute
```

`AG f i s` is the description of the `AG`. Its semantics is a a tree traversal that computes the synthesized attribute of the root given an inherited attribute for the root. We call this function `i → s` the semantics of the tree given by the attribute grammar. We express the tree traversal as a container algebra.

```
algAG :: Shape f ⇒ AG f i s → AlgW f (i → s)
algAG (AG ag) f p i = s
  where  (s , pi) = ag f (i, appS f p pi)
```

Remarks

`p` is a collection of the semantics of the subtrees (children), `pi` is the collection of inherited attributes for the children, `appS f p pi` applies the semantic function of each child to its inherited attribute to compute the child's synthesised attribute. It is best if the operation `appS f p pi` is lazy in `pi` to ensure the circular definition is not diverging, but it is enough that `ag` itself is lazy in the collection of children synthesized attributes.

We must iterate the algebra to traverse the tree.

```
runAG :: FixW f t ⇒ AG f i s → t → i → s
runAG ag t i = cataW (algAG ag) t i
```

For the rest of the section we define some useful operations on AG.

Product of Attribute Grammars

The product of two AG takes the product of inherited attributes and return the product of synthesized attributes. It satisfies:

```
runAG (pairAG g1 g2) t (i1,i2) ≡ (runAG g1 t i1, runAG g t i2)
```

In terms of algebras, it verifies:

```
algAG (pairAG g1 g2) ≡ hpair (algAG g1) (algAG g2)
```

However, the left hand side does one traversal whereas the right hand side does two, see §7.1.5 p. 96 for the definition of `hpair`.

The implementation of `pairAG` uses the fact that the children collection is an applicative functor to separate the component of the pairs of synthesized attributes and to make the pairs of inherited attributes for each child.

```
pairAG :: Shape f ⇒ AG f a b → AG f c d → AG f (a,c) (b,d)
pairAG g1 g2 = AG $ λ f ~((a,c), pbd) →
  let (b, pa) = appAG g1 f (a, pb)
      (d, pc) = appAG g2 f (c, pd)
      pb      = mapS f fst pbd                    -- b attributes from (a,b)
      pd      = mapS f snd pbd                    -- d attributes from (b,d)
      pac     = mapS f (,) pa ⊛ pc               -- pairs of a and c attributes
      (⊛)     = appS f
  in ((b,d), pac)
```

Constant Inherited Attribute

If an attribute grammar depends on a global parameter, we can make it an inherited attribute that gets duplicated to every non-terminal of the grammar.

```
inherit :: Shape f ⇒ (a → AG f b c) → AG f (a,b) c
inherit pag = AG $ λf ~((a,b), pc) →
  let (c, pb) = appAG (pag a) f (b, pc)
  in (c, mapS f (λb → (a,b)) pb)
```

Attribute Grammar from an Algebra

An algebra is an attribute grammar with only synthesized attributes.

```
synth :: Shape f ⇒ AlgW f s → AG f () s
synth alg = AG $ λf ~((),s) → (alg f s, pureS f ())
```

It verifies

```
algAG (synth (curryW alg)) ≡ curryW (cons . alg . fmap ($ ()))
```

Note that we cannot compute the inverse of `algAG`, i.e. a function of type:

```
AlgW f (i → s) → AG f i s
```

Thus AGs contain strictly more information on the traversal than algebras. Just like algebras contain strictly more information than a catamorphism.

Identity Attribute Grammar

`idAG` copies its inherited attribute everywhere: inherited and synthesized attribute all are the same. The semantics is the identity function for any tree `t`:

```
runAG idAG t ≡ id
```

```
idAG :: Shape f ⇒ AG f x x
idAG = AG $ λf ~(x, ps) → (x, pureS f x)
```

7.4.3. ArrowAG

We extend ArrowCata with a new primitive to run an attribute grammar. The arrow primitive takes the AG and the inherited root attribute as input and returns the synthesized attribute of the root. As for ArrowCata, the tree on which the traversal is being made is implicit.

```
class (ArrowCata (WF f) (⇝)) ⇒ ArrowAG f (⇝) | (⇝) → f where
  apply_ag :: (AG f i s, i) ⇝ s
```

The primitive apply_ag allow to compose higher-order AG: an AG which computes another AG [VSK89]. On the other hand, if the AG isn't computed by another AG, we may prefer to use the following function:

```
ag :: ArrowAG f (⇝) ⇒ AG f i s → i ⇝ s
ag g = proc i → apply_ag ≺ (g,i)
```

We may also initialise the inherited attribute from outside the arrow:

```
ag_init :: ArrowAG f t ⇒ i → t (AG f i s) s
ag_init i = proc g → apply_ag ≺ (g,i)
```

Once an AG is made into a ArrowAG, it can be composed with other AG computations simply by using the composition operator (.) redefined in Category, the super-class of Arrow. Composing AGs g and f is as simple as: ag g . ag f.

It is the first time to my knowledge that attribute grammars are composed using arrows. This promotes an applicative style which is closer to that of the functional paradigm.

7.4.4. Multiple Traversals

Env is an instance of ArrowAG. The method apply_ag immediately runs the attribute grammar, therefore when composing attribute grammars, each does its own traversal (a pass in compiler terminology).

```
instance (FixW f t, Shape f) ⇒ ArrowAG f (Env t) where
  apply_ag = Env $ λt (g,i) → runAG g t i
```

Env is again the semantics for the circular implementation.

7.4.5. Circular Implementation

We revise the type of circular programs of §6.3.2.

```
data CircAG f a b  =  ∀ i s . CAG (a → s → (AG f i s, i, b))
```

Let us breakdown the role of each type components:

```
data CircAG f a b = ∀ i s . CAG (
a →            input of the computation: domain of the arrow
s →            synthesized attribute of the root after computation
(AG f i s,     attribute grammar
i,             inherited attribute for the root
b)             result of the computation: codomain of the arrow
```

The types of inherited and synthesized attributes are hidden by the existential quantification. The attribute grammar, the inherited attributes for the root and the result of the computation all depend on the input and the synthesized attributes of the root element, obtained by circularity after running the attribute grammar.

Running the Circular Computation

Just as in Chapter 6, the semantics of a circular program is given in the environment arrow.

```
circAGtoEnv :: FixW f t ⇒ CircAG f a b → Env t a b
circAGtoEnv = Env . runCircAG
runCircAG :: FixW f t ⇒ CircAG f a b → t → a → b
runCircAG (CAG body) struct param = result
  where (ag, inherited, result) = body param (runAG ag struct inherited)
```

Instances Definitions

The implementation of `ArrowAG` combinators is straightforward and is similar to the instances for `Circ` given in §6.3.2 for which we gave detailed explanations.

Composition is the most interesting case. We make the pair of the two underlying AGs, we must wire the pairs of synthesized and inherited attributes to the corresponding CAG body, and the thread the arrow computation parameter `x` to `b1`, whose result `r1` is then passed to `b2` producing the final result `r2`. One must be careful to use lazy patterns on the second argument of a `CircAG` so that the circularity in `runCircAG` doesn't cause it to diverge.

```
instance Shape f ⇒ Category (CircAG f) where
  id = arr id
  CAG b2 . CAG b1 =  CAG $ λ x ~(s1 , s2) →          -- the lazy pattern is crucial
    let (g1, i1, r1) = b1 x s1
        (g2, i2, r2) = b2 r1 s2
    in (pairAG g1 g2, (i1, i2), r2)

instance Shape f ⇒ Arrow (CircAG f) where
  arr f = CAG $ λx ~() → (idAG, (), f x)              -- arr :: (a → b) → (a ⇝ b)
  first (CAG b) = CAG $ λ(x,y) s →            -- first :: (a ⇝ b) → ((a,c) ⇝ (b,c))
    let (g, i, r) = b x s
     in (g, i, (r,y))

instance Shape f ⇒ ArrowCata (WF f) (CircAG f) where
  cataA = ag_init () <<< arr (λx → synth (curryW x))                          --
cataA :: Alg (WF f) x ⇝ x
```

```
instance Shape f ⇒ ArrowAG f (CircAG f) where
  apply_ag = CAG $ λ ~(g,i) s → (g, i, s)        -- apply_ag :: (AG f i s, i) ⇝ s

instance Shape f ⇒ ArrowLoop (CircAG f) where
  loop (CAG b) = CAG $ λ x s →                -- loop :: ((a, r) ⇝ (b, r)) → (a ⇝ b)
    let (g, i, ~(y, r)) = b (x, r) s
     in (g, i, y)
```

7.5. Examples

7.5.1. Palindrome

The palindrome function studied in §7.1 can be implemented modularly using `ArrowAG`. The two traversals are defined as attribute grammars.

List equality.

```
eqlistAG :: Eq a ⇒ AG (ListS a) [a] Bool
eqlistAG = AG $ λcase
  NilS    → λ(y, V0)   → (null y, V0)
  ConsS a → λ(y, V1 t) → case y of
    [] → (False, V1 [])
    (x:y') → (a ≡ x && t, V1 y')

eqlistA :: (Eq a, ArrowAG (ListS a) (⇝)) ⇒ [a] ⇝ Bool
eqlistA = ag eqlistAG
```

Reversing a list. The inherited attribute is used as a stack to compute the result: when the end of the list is reached (`NilS` case), the inherited attribute is returned.

```
revcatAG :: AG (ListS a) [a] [a]
revcatAG = AG $ λcase
  NilS    → λ(z, V0)   → (z, V0)
  ConsS a → λ(z, V1 r) → (r, V1 (a:z))
```

To compute the reverse of the list we must initialise the inherited attribute with the empty list:

```
reverseA :: ArrowAG (ListS a) (⇝) ⇒ () ⇝ [a]
reverseA = proc () → ag revcatAG ≺ []
```

The palindrome function simply composes reversal and equality (remember that the list being traversed is implicit).

```
palindromeA2 :: (Eq a, ArrowAG (ListS a) (⇝)) ⇒ () ⇝ Bool
palindromeA2 = eqlistA . reverseA
```

We can evaluate the program as a single traversal:

```
palindrome6 :: (Eq a) ⇒ [a] → Bool
palindrome6 x = runCircAG palindromeA2 x ()
```

7.5.2. Removing the Redundant Tests

In the previous implementation of palindrome half of the equality tests are redundant since it is enough to test that the first half of the list is equal to the reverse of the second half (which is equal to the first half of the reverse of the whole list). Therefore we implement a new list equality predicate `equalprefix` which takes an additional integer parameter to interrupt the recursion early.

```
palindrome7 x = equalprefix (n + q) x (reverse x)
  where (n, q) = length x ‘quotRem‘ 2

equalprefix n x y
  | n ≤ 0 = True
  | null x || null y = False
  | otherwise = head x ≡ head y
              && equalprefix (n - 1) (tail x) (tail y)
```

We must provide AG primitives for `length` and `equalprefix`: `length` is best implemented using an algebra.

```
listAlgW :: b → (a → b → b) → AlgW (ListS a) b
listAlgW n c NilS V0 = n
listAlgW n c (ConsS x) (V1 xs) = c x xs

lengthAlgW :: AlgW (ListS a) Int
lengthAlgW = listAlgW 0 ((+) . const 1)

lengthA2 :: ArrowAG (ListS a) (⇝) ⇒ () ⇝ Int
lengthA2 = ag (synth lengthAlgW)
```

The AG for `equalprefix` inherits the prefix length and the second list. Above, the guards allowed for an concise implementation, instead we are forced to pattern match first on the list. Which means the case `n ≤ 0` is repeated.

```
equalprefixAG :: Eq a ⇒ AG (ListS a) (Int, [a]) Bool
equalprefixAG = AG $ λcase
  NilS → λ ((n,y), V0) → (n ≤ 0, V0)
  ConsS a → λ ((n,y), V1 t) →
    if n ≤ 0 then (True, V1 (n,y))
    else case y of [] → (False, V1 (n,y))
                   (b:y’) → (a ≡ b && t, V1 (n-1, y’))

equalprefixA :: (Eq a, ArrowAG (ListS a) (⇝)) ⇒ (Int, [a]) ⇝ Bool
equalprefixA = ag equalprefixAG
```

A modular version of the palindrome with no redundant tests.

```
palindrome8 = proc () → do
  rev ← reverseA ≺ ()
  len ← lengthA2 ≺ ()
  let (n, q) = len ‘quotRem‘ 2
  equalprefixA ≺ (n + q, rev)
```

Chapter 8.

Perspectives

In this chapter we put our work on circular programming in perspective. We discuss the performance of the library. We compare it to other optimisation techniques. We discuss other uses of containers in particular their application to implement attribute grammars. We conclude with a survey of related work.

Overview First (§8.1) there is the important aspect of performance. Does the circular programming technique offer an improvement over multiple traversals? What is the additional cost of our implementation over manually written circular programs? In §8.2 we discuss other transformation based program optimisations, in particular optimisations where the program must conform to certain recursive patterns. In §8.3 we explain further developments on containers and the potential they offer for generic programming. In §8.4 we give a review of our strongly typed embedding of a full AG system in Haskell and compare it with other existing first class AG implementations. Finally in §8.5 we give credit to related works which influenced us and we compare our work with theirs.

8.1. Performance

Originally, circular traversals were used as an optimisation. However recent advances in compilers for lazy languages optimise multiple strict traversals often much better than circular ones. Circular programs, by combining the consumers of a datastructure maximise the possibilities of fusions, particularly of the foldr/build variety [GLJ93].

An improved circular transformation algorithm [CGK99] incorporates a strictness analysis to ensures the circular program is as strict as possible. Their experimental results show that their stricter circular programs are an improvement over the multi-traversals, whilst they also show that the plain circular programs have worse performance. Unfortunately, we haven't been able to reproduce their results. They stress that it is essentials that the strictness analysis be done prior to introducing circularity.

Although our research on circular traversals wasn't focused on optimisation but rather in finding a good abstraction, it is important to compare the performance of programs using our library, with the corresponding multiple and circular traversals written directly (without the library). Remember that a program using the library can be run both as multiple traversals or as circular traversals.

Our conclusions should be taken with caution as we only compared the performance of a single example: *palindrome* which is a very small program. Nonetheless, that example is disappointing from an optimisation perspective: manually written implementations both normal and circular are faster than the ones using the library and multiple traversals solutions are always much faster than the circular ones.

8.1.1. Experimental Results

We used GHC version 7.8.3 and compiled our program with option `-O` for the normal level of optimisation. We ran different implementations of the function `palindrome` with arguments `[1..n]++[n,n-1..1]` with `n = 10000`.

Description	Program	Heap Alloc.	GC Copy	GC colls.
Original				
Multi Traversal	`palindrome`	2,754,472	2,043,936	6
Bird	`palindrome2`	8,034,512	5,150,008	16
Bird Fixed	`palindrome4`	9,474,488	9,065,392	19
ArrowAG:Env	`palindromeA2`	26,908,888	4,224,720	52
ArrowAG:Circ	`palindrome6`	294,923,656	127,285,536	582
Equal Prefix				
Multi Traversal	`palindrome7`	2,072,016	1,099,992	4
ArrowAG:Env	`palindrome8`	35,674,736	6,103,816	68
ArrowAG:Circ	`palindrome8`	414,818,048	226,515,240	828

The last two columns correspond to the memory copied during garbage collection and the number of garbage collections. Sizes are given in bytes.

It is striking that our modular approach comes with a huge overhead, which is difficult to explain unless more is known about the particular compiler. `ArrowAG:Env` programs should be operationally equivalent to the multi traversals yet they perform with a significant penalty. The `ArrowAG:Circ` program should be equivalent to `Bird Fixed`, yet it is inconceivably memory hungry. Such a huge cost in memory and garbage collection overshadows any possible advantage that could have been gained in traversing the datastructure only once or of deforesting it.

The tests were done with the short-cut fusion enabled to optimise lists operations and we might think that it bias the comparison, but when deactivating them the figures are similar.

In summary, our comparison showed bad performance for our system on a single and extremely simple example. This motivates further research to improve our solution, as well as investigate the difficulties of the compiler to optimise it. More experiments with more complex examples are needed before any strong conclusions should be formed regarding the practical potential of our approach.

8.2. Optimisation and Recursion Schemes

As we explained in the previous section, the circular transformation was originally intended as an optimisation. We now discuss the general problem of program optimisation based on identifying recursion schemes and discuss the advantages of using the newly found attribute grammar recursion scheme for optimisation purposes. We conclude that strong languages by limiting (co)-recursion facilitate the task of automatic optimisation.

Many optimisation techniques involve using specific recursion schemes to define functions. If the programmer is to manually optimise his code with such a technique, he almost always trades program clarity or modularity for efficiency.

Ideally, we would write programs only caring about their semantics and rely on a clever compiler to optimise it. That task is greatly facilitated when programs are structured to reflect some underlying principles. For instance, writing traversals as algebras rather than general recursive functions gives opportunities for the compiler to combine them. One major success story is the foldr/build fusion rule implemented in GHC which expects consumers (traversals) as algebras and producers to use a polymorphic encoding rather than constructors [GLJ93].

If algebras are useful, it is natural that attribute grammars will be even more so, since they provide more insight on the recursion (one can compute an algebra from an attribute grammar, but not the converse). By attribute grammar, we mean the recursive scheme in its embodiment with containers, rather than a high level attribute grammars language. An AG recursive scheme would provide even more clues to a compiler. In particular it would allow to choose the most efficient way to schedule traversals, making use of circularity only when it's advantageous, combining strict traversals as much as possible.

However this disciplined programming is in practice too much to ask. There are two reasons: 1) there may be numerous underlying schemes in which to cast a given function. For instance `map` may be viewed both as a catamorphism and an anamorphism, both as a producer and a consumer. In fact, simply considering shortcut fusion and its dual stream-fusion[CLS07], there would be already four ways to define `map` so that it is fusible. 2) The specific encodings that are required usually add some complexity. For instance producers in short-cut fusion and especially consumers in stream-fusion may be repelling for the extra effort required to implement them.

If we cannot rely on the programmer to write his programs using given abstractions, one approach to optimisation is to transform it so that it does. To make the recursive schemes explicit, the compiler must first find the algebras corresponding to catamorphisms [LS95, JL97, Nem00, JV00, YHT] or the coalgebras corresponding to anamorphisms [CLS07, Har11, HHJ11] or both algebras and coalgebras underlying hylomorphisms [DP11], or any other possible scheme (paramorphisms, histomorphisms, etc).

In the presence of general recursion, such a transformation is never complete in the sense that there will always be some catamorphisms that the compiler doesn't recognise as such, and cannot calculate their algebra. This is a consequence of Rice's theorem which states that any non-trivial property of a computable function is undecidable.

A clear advantage of strong languages against weak ones, with respect to optimisation, is that by forbidding general recursion it is always possible to derive the algebras and coalgebras. It also seems likely that attribute grammars could be automatically derived.

8.3. Generic Programming with Containers

Containers were used in Chapter 7 to precisely capture the recursive scheme underlying the evaluation of attribute grammars. Containers are a recent discovery with many useful applications to generic programming. This section discusses containers in this broader context.

Overview In §8.3.1 we start by discussing existing work on generic programming with containers and parametric genericity. In §8.3.2, we list some important implementations of generic functions using containers, noting their suitability for defining generic types. In §8.3.3 we discuss different possibilities for the payload functor when implementing containers in Haskell. In §8.3.4 we explain the generalisation of containers to *indexed containers* which provides a generic view for mutually recursive datatypes, and thus allows us to implement a full attribute grammar system, since we must support context free grammars, naturally implemented as mutually recursive datatypes. In §8.3.5 we show some other use of containers: implementing container based modular datatypes (à la carte) and generic operations on them involving paths.

8.3.1. Related Works

Generic programming with containers is the subject of two chapters of Morris [Mor07], there is unfortunately too few examples of generic programming: the major part of the chapters are concerned with implementing containers and the bijection between the container and strictly positive generic view. Only one section (5.4) gives a simple example of a generic program: the generic `Map`, a one liner when using containers, to be contrasted with the equivalent definition with a syntactic view which would need to consider many cases. Morris mainly used this example to illustrate that containers offer an alternative view of types where their syntactic structure doesn't play a role. Let me stress again that very importantly this means that *parametric* properties of generic functions with containers are automatic. A feature that Gibbons [GP09] values particularly. The containers are therefore a welcome addition to the functorial based generic-view that Gibbons studied. In his paper, he only considered (co-)iteration over (co-)algebra, with quite a limited scope for generic programming. Containers broaden the applications of parametric genericity.

8.3.2. Generic Functions and Generic Types

In many cases, generic programs not only work on the generic representation of a type, but also on types which are constructed generically from another type. We saw an example with our AG implementation: the type `AG c i s` depends on the container representing the base functor of a recursive type and the generic function `runAG :: AG c i s → W c → i → s` traverses the structure whose generic representation is `W c`.

We believe that containers are particularly suited to define generic types. Generic types constructed using the syntactic view are often hard to use because they usually consist of sums of products and fixed-points: the constructor names from the original type are lost. On the other hand, the semantic view offered by containers allow to retain the connection with the original constructors via the shapes.

Apart from their application to attribute grammars, we know of two other elegant generic types using containers.

Generic differentiation [Mcb01b] is used in the definition of the zipper [Hue97]. The derivative of a containers is given [AAGM05]. It should be interesting to see if we can implement this in Haskell using our encoding of containers with GADTs.

From a recursive type, we can define a type of its path which uniquely identify subtrees. Operations on path include membership, extracting a subtree, substituting another subtree. In this thesis we used containers as representation of the base functors of recursive datatypes and with this representation paths are lists of dependent pairs of shapes and positions. It becomes more interesting when we want to define the paths of mutually recursive datastructures. We give more details about this in a section below. Interestingly, containers can also be seen as representation of datastructures with the positions being in fact paths to the leaves.

8.3.3. Design Choices for the Payload Functor

The basic characteristic of containers is to split a functor into its non-recursive part (the shape) and its payload depending on the shape. We can choose to implement the payload as a function indexed by the recursive positions. This is the usual definition of containers.

```
data WF1 s where
  WF1 :: s p → (p → x) → WF1 s
```

Or we can represent the payload as an applicative functor. It is a generalisation of the previous case, as `p → x` is an applicative functor. This is the approach followed in this chapter. Note that the applicative instance must be given for every index `f` of a shape type `s`.

```
data WF2 s where
  WF2 :: s f → f x → WF2 s
```

A third possibility, when the children are finite is to index the shape with the number of children and use a vector type for the payload. Such containers are called *decidable* because equality on positions is decidable [Abb03].

```
data WF3 s where
  WF3 :: s n → Vec n x → WF3 s
```

Where `n` is a type level natural number and `Vec n x` is a dependent type of lists of length `n`.

With that last approach we may define uncurried functions from vectors: instead of `Vec n x → y` we may define a `Uncurry n x y` where `Uncurry` is a type function defined as a type family in GHC, such that:

```
Uncurry 0 x y = y
Uncurry (n+1) x y = x → Uncurry n x y
```

This makes programming with containers very non-obtrusive. For instance a list algebra using a this approach would be written:

```
length NilS = 0
length (Cons x) n = n + 1
```

This is the approach used in our implementation of an AG System on top of the AG type.

8.3.4. Mutually Recursive Data Structures

To program generically with mutually recursive data structures, and to implement context free attribute grammars, we need to view them as indexed W types, and represent their indexed base-functor as an indexed container. Although a complex dependent type, indexed containers can be implemented in Haskell using the same display-map technique we used for simple containers.

Indexed containers are defined in type theory as follows [AM09]:

```
data ICont (I : Set)
           (S : I → Set)
           (P : (i : I) → S i → I → Set)
           (X : I → Set)
             : I → Set where
  icont : (i : I) →
          (s : S i) →
          (f : (j : I) → P i s j → X j)
          → ICont S P i
```

In the Haskell encoding `P i` is the display map of `a i`.

```
data ICont (a :: * → (* → *) → *)
           (x :: * → *)
           (i :: *) where
  icont :: a i b → (∀ j . b j → x j) → ICont a x i
```

The following correspondence between the type-theoretical and Haskell implementation may be helpful.

Type theory	Haskell
`Set`	Kind *
`I : Set`	Kind *
`i, j : I`	`i, j : *`
`{ s : S i | P i s = b }`	`a i b`
`P i s : I → Set`	`b : * → *`
`X : I → Set`	`x : * → *`

8.3.5. Modular Datatypes and Paths

As part of our AG System described in the next section, we have implemented a modular generic representation of mutually recursive datatypes, inspired by [Swi08] and [BH11]. Basically, each constructor is defined as an independent container functors, and we can gather any collection of them in a sum before taking the fixed-point (W-type), or more usefully to define a free monad. Indexed containers offer the right abstractions to implement strongly typed paths, in term of which we may define some useful generic functions: we may for instance check whether a path is contained in a certain datastructure, and if so extract the subtree identified by that path, or substitute another subtree. The notation we found is extremely clear: it is a dependently typed list of alternating shapes and positions: each shape must be tested against the nodes of the tree, and each position is used to select one of the children for each node. For instance, a path through a binary tree, starting at the root, expecting a node (not a leaf) and following its left child, and so on.

```
a_path = TOP ▷ node ⟶ left
             ▷ node ⟶ right
             ▷ node ⟶ right
```

Note that the actual shape value is not important, only its type matters, for that reason we chose to use type proxies instead. `node` is a proxy for `Node`. Thus the data associate with each `Node` shape doesn't appear in the path.

8.4. Embedding a Strongly Typed AG System

Using our Haskell encoding of indexed-container functors, we implemented a strongly typed embedded AG system for Haskell. Our library, using many type system extensions, is essentially a dependently typed implementation of [MBS00]. It retains the advantage of type inference, and a simplicity of the types that the user can manipulate. The authors of the aforementioned article doubted that such a simplicity would be possible whilst enforcing strong type invariants. Moreover, the syntax of the embedding is pure haskell yet is that of a high level AG language unlike that used in [MBS00] and [VSS09] where a pre-processor or macro-processor was needed in order to present a nicer syntax. Such a two-level approach has a disadvantage in that the user who wants to define new combinators has to manipulate the underlying implementation which is very different from the surface language.

It has a number of novel characteristics. Notably its high degree of modularity. It implements all the modular features of [MBS00] and more, whilst implementing a stronger type safety. The features in common with [MBS00] are:

- Attribute grammars are composed of aspects which are merged together. Also provided in [VSS09].
- Aspects defines the computation rules of a single attribute for a set of production. They may depend on other attributes, but the computation of those attributes is defined independently as another aspect.
- Rules define the computation of a single attribute for a single production.
- Rule combinators are functions that compute some rule according to a general pattern: copying, collecting the values of a monoid, chaining attributes.

We were also able to implement AG macros as in [VS12] which allow to define the semantics of a constructor in terms of others and delegate all attribute computation to them, save for a few special cases. As example they define a constructor `Square` in terms of `Multiply`, and inherit all the semantics of computing a value but override the semantics of representing the expression as a string.

We must stress the new level of modularity that we have by defining rules on single productions: this allows us to define attribute grammars over modular datatypes as was done in [Swi08] but with algebras. We also work with mutually recursive datatypes as in [BH11].

Thus there are three independent dimensions of modularity. Two of them correspond to the expression problem [Coo91].

- Productions (constructors) are defined independently and combined;
- Attributes are defined independently and combined;
- Rules are defined independently and combined.

Some other features supporting modularity:

- Reusing an attribute grammar, renaming some attributes;
- Reusing an attribute grammar, changing the semantic rules for some attributes, for some productions;
- Reusing rules on other productions to define a rule;
- Generic rules which can be used as default, and implement some common pattern: copy, chain, collect, macros.

We find worth mentioning that all the generic rules are implemented on top of the library, as a user could program them. In particular the macro system described in the example below relies on the possibility to run the AG itself that is being defined: this is a true example of higher-order attribute grammar where an attribute is an attribute grammar and is executed during the computation of another attribute grammar. For the macro system, we make sure that the two are actually the same using a circular binding.

Finally we made a lot of effort in order to simplify the types that the user of the library must manipulate, but in some cases the error messages can leak some information about the library implementation and are actually quite remote to what we would want to tell the user if we had a control on GHC error handling.

The whole library is about 3250 lines of Haskell, including comments. It went through 27 revisions. It is now ready for being made public.

8.4.1. Example

Without giving any details about the actual library implementation, we will give an example of its use, so as to illustrate some of the above claims. The whole program is given at the end of the section, we now proceed to explain it step by step.

We give some snippets of a attribute grammar for a primitive desk calculator, the main example of Paakki's AG survey [Paa95], also used in [MFS13].

A desk program is of the form `PRINT e WHERE x = c, ... x = c` where `e` stands for an expression, `x` stands for variables, and `c` for integer constants. Expressions are sums of variables and constants. For instance:

```
PRINT x + y + 11 WHERE x = 22, y = 33
```

A Haskell representation of this language uses mutually recursive datatypes to precisely capture the context free grammar.

```
data Prog = Print Exp [Def]
data Exp  = Exp :+ Fact | Fact Fact
data Fact = Var String | Cst Int
data Def  = String := Int
```

Our previous example:

```
deskExample =
  Print (Fact (Var "x") :+ Var "y" :+ Cst 11) ["x" := 22, "y" := 33]
```

We will write an attribute grammar to compile Desk programs. The target language has four instructions:

```
data Instr = LOAD Int | ADD Int | PRINT | HALT
```

A compiled program is simply a list of instruction. The previous example compiles to:

```
runDeskG deskExample
   ⟹  [LOAD 22, ADD 33, ADD 11, PRINT, HALT]
```

Grammar Definition We will implement the same language in a modular fashion: meaning that we will be able to extend it later, and choose the constructors a la carte, as in [BH11] (which generalises [Swi08] to mutually recursive datatypes).

Each non-terminal is defined as an empty type, serving as a label

```
data PROG
data EXP
data DEF
data DEFS
data FACT
```

Each production, is given a name (corresponding to the datatype constructor). This time, the datatype should have only one constructor and should contain any non-recursive data. We only give two cases, the other ones are similar. Note that we reuse the same constructor names as in `Prog`, this only means that the two definitions are in different modules. As a convention, we write non-terminals in upper case.

```
data Print = Print
instance Constructor Print where
  type Production Print = PROG ⟹ '[EXP, DEFS]          -- list of non-terminals
```

Notice that how we specify the recursive part of the production almost as BNF production, thanks to the type level list of GHC. $\Longrightarrow$ is simply a more illustrative way to write a product type.

The `Var` production has only one terminal, the `String` name of the variable and no non-terminals are on the right hand side.

```
data Var = Var String                          -- terminals are part of the shape
instance Constructor Var where
  type Production Var = FACT ⟹ '[]      -- no non-terminals for that production
```

Next some smart constructors are defined. A process that may seem tedious although could easily be automated with template Haskell. Smart constructors are immensely useful when using datatypes à la carte, and our implementation is similar in spirit to that of [Swi08]. We omit the details.

```
iprint   = injC proxies Print
ivar     = injC proxies . Var
```

We obtain an implementation à-la-carte of `Desk` as the fixpoint `Expr` of the sum `CSum` of its constructors functors given as a type level list, once again very useful.

```
type DeskExpr = Expr (CSum '[Print, Add, Fact, Var, Cst, Defnil, Defcons, Def])
```

We can now convert from `Desk` to `DeskExpr` using the smart constructors. Each datatype (non-terminal) needs its conversion function:

```
progE  :: Prog  → DeskExpr PROG
expE   :: Exp   → DeskExpr EXP
factE  :: Fact  → DeskExpr FACT
defsE  :: [Def] → DeskExpr DEFS
defE   :: Def   → DeskExpr DEF
```

We give two cases to illustrate how straightforward the definition is.

```
progE (Print e ds) = iprint (expE e) (defsE ds)
factE (Var v)      = ivar v
```

We may also do without using `Desk` and using `DeskExpr` with the smart constructor instead.

Attributes definition The attribute grammar has the following attributes.

code synthesized target code, list of instructions, defined for `PROG` and `EXP`;
name synthesized name, string, defined for `DEF`;
value synthesized value, integer, defined for `DEF`, `FACT`;
ok synthesized attribute, boolean that indicates correctness, defined for `DEFS`, `FACT`;
envs synthesized environment, symbol table, defined for `DEFS`;
envi inherited environment, symbol table, defined for `EXP`, `FACT`.

The textual description translates immediately so transparently to Haskell that there is very no need for explanation. We give only a few cases.

```
data Code = Code
instance Attribute Code where
  type Mode Code = Synthesized
  type Type Code a c n = [Instr]
  type Domain Code = Over '[PROG, EXP]
```

```
data Ok = Ok
instance Attribute Ok where
  type Mode Ok = Synthesized
  type Type Ok a c n = Bool
  type Domain Ok = Over '[DEFS, FACT]
```

```
data EnvI = EnvI
instance Attribute EnvI where
  type Mode EnvI = Inherited
  type Type EnvI a c n = SymbolTable
  type Domain EnvI = Over '[EXP, FACT]
```

Remark: the parameters `a`, `c`, `n` of `Type` allow to define attributes depending on (`a`) the attribute record, (`c`) the container and (`n`) the non-terminal to which it is associated. Those parameters are not used in this example but are very useful in other cases.

Aspects and Namespace Before defining rules, we must define a namespace: attributes may be given alternative computation rules in different namespaces. For our simple example we use one namespace:

```
data Desk = Desk
```

Namespaces contain a set of rules. There are some operations for importing rules from other namespaces. The importing mechanism is very flexible but we lack the space to describe it in detail.

An aspect is a set of concrete choices of rules picked from different namespaces. Rules are uniquely identified by their namespace and the attribute they implement. Here we define a single aspect which takes all of the rules from the same namespace:

```
type DeskAspect =
  IA Desk '[Code, EnvS, Ok, Value, Name, EnvI]
```

Semantic Rules We use two classes to define rules: `SRule` for synthesized attributes and `IRule` for inherited attributes.

The context of the instance declaration `UseS a '[Ok]` says that we will use the synthesized attribute `Ok`. Then the parameters of the class are: the namespace `Desk`, the attribute being defined `Code` and the production `Print`. The type system only allow correct rules: for instance this one is possible because `Code` is defined for the non-terminal `PROG` of which `Print` is indeed a production.

The actual implementation is given by the method `srule` of which the first argument is a type proxy necessary to identify the namespace (`Desk` here), the second is the inherited attribute `i`, the third is the production shape `Print` (along with non-recursive data) and the rest are the eventual recursive arguments. Note that the method has a variable number of argument using the dependently typed technique described in §8.3.3: observe how the definition feels almost as if we used the original constructors. There is absolutely no syntactic overhead and we didn't even resort to template Haskell.

In the definition, any attribute may be referred to using his label and the operator `!` (left associative and with the highest priority so that attributes can be used as functions). The validity is typechecked: here it is possible to access the `Ok` attribute of `d` because `Ok` is defined for `DEFS` and `d` is indeed a `DEFS` since it is the second child of `Print`. Likewise, `Code` is defined for `e` of type `FACT`. Note that `Code` is being defined so we can use in the definition without listing it in the context. The instance wouldn't be valid if we'd use an attribute but didn't specify it in the context unless it is being defined.

```
instance (UseS a '[Ok]) ⇒ SRule c a Desk Code Print where
  srule _ i Print e d  =  if Ok! d then Code! e ++ [PRINT, HALT] else [HALT]
```

The other two productions for which a rule for `Code` should be defined are `Add` and `Fact` for the `EXP` non-terminal. Both of them need to access the attributes `Ok` and `Value`.

```
instance (UseS a '[Ok, Value]) ⇒ SRule c a Desk Code Add where
  srule _ i Add e f  =  if Ok! f then Code! e ++ [ADD $ Value! f] else [HALT]
```

`Fact` has only one child.

```
instance (UseS a '[Ok, Value]) ⇒ SRule c a Desk Code Fact where
  srule _ i Fact f  =  if Ok! f then [LOAD $ Value! f] else [HALT]
```

Let us also see the rules for `Ok`, recall that this attribute is defined over `FACT` with productions `Var` and `Cst`, and `DEFS` with productions `Defnil` and `Defcons`.

The case for `Var` uses the inherited attribute `EnvI`: we must project it from the inherited record `i`. Observe as well that `Var` doesn't have children, and its field `name` corresponds to a non-terminal of type `String`.

```
instance (UseI a '[EnvI]) ⇒ SRule c a Desk Ok Var where
  srule _ i (Var name)  =  Map.member name (EnvI! i)
```

The cases for `Cst` and `Defnil` show that all the information really necessary to identify the rule is given in the instance head.

```
instance SRule c a Desk Ok Cst where
  srule _ _ _  =  True
instance SRule c a Desk Ok Defnil where
  srule _ _ _  =  True
instance (UseS a '[Name, EnvS]) ⇒ SRule c a Desk Ok Defcons where
  srule _ i Defcons h t  =  Ok! t && not (Map.member (Name! h) (EnvS! t))
```

We now give a rule for inherited attributes. By default inherited attributes are simply passed down the tree, this is achieved by saying that in the default case, the copy rules should be used in the namespace `Desk`.

```
type instance Default Desk = Copy
```

Copy rule will cover the cases for `Add` and `Fact`. The only case we need to implement is for `Print`. The context is the same as in `SRule`, and `IRule` has the same parameters as `SRule` plus one added at the end which is the name of the child for which we define the inherited attribute. Recall that each child gets an inherited attribute from his parent, since we may want to pass different values to different children, each child gets its own `IRule` instance (only if necessary: the default mechanism and copy rule take care of all the other cases).

```
instance (UseS a '[EnvS]) ⇒ IRule c a Desk EnvI Print PrintEXP where
  irule _ i Print e d  =  EnvS ! d
```

Children names are simply type level natural numbers identifying their position. We will define `EnvI` for the `EXP` child of `Print`. It is its first child so we define

```
type PrintEXP  = N0                                    -- N0 is the type level zero
```

Running the AG We define an AG by combining different attributes. The actual rules chosen to compute them are given independently by choosing of a particular aspect. Only inherited attributes like `EnvI` are initialised.

```
deskF envi = envi ‘asAttr‘ EnvI  &  Code  &  EnvS  &  Ok  &  Name  &  Value
```

We run the AG by providing a namespace in which to choose the attribution rules (here `Desk`) and by providing the AG specification `deskF`. `attrTrivial` is the initial value for the inherited attribute `EnvI`: it is used in this case because the non-terminal `PROG` doesn't inherit a `EnvI`. Finally, we extract the `Code` attribute which contains the list of instructions corresponding to the `Desk` program.

```
runDeskG :: Prog → [Instr]
runDeskG prog = Code ! runAG’ Desk (deskF attrTrivial) exp
  where exp = progE prog
```

AG Macros We extend the grammar with a new production `Twice` for the `EXP` non-terminal. It has a single child, a `FACT` non-terminal.

```
data Twice = Twice
instance Constructor Twice where
  type Production Twice = EXP  ⟹  '[FACT]
```

We define a new namespace for the extension:

```
data DeskExt = DeskExt
```

Just like we used the copy rule as default for the `Desk` namespace, we can also import rules from different namespaces. Here we import the macro rules and copy rules for `Twice`, as well as all the rules of `DeskAspect` and `Utils` contains some rules necessary for the macro mechanism.

```
type instance Import DeskExt =
  '[ IA Copy '[Macro Twice]
   , IC Macros '[Twice]
   , DeskAspect
   , UtilsAspect]
```

A macro definition has two components. The first builds an macro expression on which to delegate the computation of synthesized attributes. Here we convert `Twice e` to the macro expression `Add (Fact e) e` and run the AG on this tree. The synthesized attribute at the root of the macro tree will be taken for the attribute of the original node (`Twice`). Note in passing that we implemented the macro system as a true higher order attribute grammar: each macro computation runs the final attribute grammar on the macro trees. The final AG is circularly bound to an inherited attribute passed down the main tree.

The second component of a macro must specify for each child of the original node where to find the corresponding inherited attribute in the macro expression. Note that the macro expression may not have the same number of children: this is the case here: `Twice e` has one child, `Add e e` has two. We must choose one of them to retrieve the inherited attribute for the child of `Twice`. The specification is given as paths: one for each child. Using containers, we can represent strongly typed path generically. Here there is only one child, and we give the path to the second child of `Add`.

```
Top ▷ Add ⟶ add_right
```

Macros are implemented on top of the AG system: the macro specification is given as an inherited attribute, and the generic rules of macro attribute computations will take care of the rest.

We define a AG fragment that only contains the macro attribute which we initialise with our specification

```
twice_macro = frag $ Macro Twice' `with`
   MacroDef (λ_ e → iadd (ifact e) e)
            (Top ▷ Add ⟶ add_right :V)
```

We can run the extended AG. This time we use `runHoagWithUtils` which takes care of passing the circularly bound AG as an inherited attribute, and adding the necessary attributes for computing macros. We reused the previous `deskF` AG fragment, and extended it with `twice_macro`.

```
runDeskExt e =
  Code ! runHoagWithUtils DeskExt (λenvi → deskF envi & twice_macro)
          (getAttr EnvI) attrTrivial e
```

Types Amazingly we where able to simplify the types that a user manipulates. This was not likely according to [MBS00]. The use of type level lists brings a lot of clarity.

The type of the previous attribute grammar fragment as inferred by the system:

```
deskF :: Container c ⇒ Attr EnvI r c 'I n →
  PFrag '[EnvI]                                   -- used inherited attributes
        '[Code, EnvS, Ok, Name, Value]          -- used synthesized attributes
        c                                                       -- container
        r                                 -- attribute record type with renaming
        r                                                 -- attribute record type
        '[EnvI, Code, EnvS, Ok, Name, Value]               -- produced attributes
        n                                                     -- non-terminal
```

`Attr EnvI r c 'I n` is the inherited attribute initial value that we must provide to actually get the AG fragment.

Whereas a proper AG cannot be extended, a fragment retain that possibility. In fact for that very reason, we made sure the user of the library cannot ever access the AG type. So that we can combine a fragment, we must keep track of 1) the attributes it uses and that may be provided by other fragments 2) the attributes it computes. When combining fragments, the union of each set of attribute is made. An AG is simply a fragment which computes all the attributes that it needs. This is the case of `deskF` above.

The `PFrag` parameters are in order: the list of the inherited attributes (only `EnvI` here) that are needed, the list of synthesized attributes (`[Code, EnvS, Ok, Name, Value]` that are needed. The type variable `c` is the container on which the AG will be run. `deskF` is generic and can be run on any container for which all the. The type variable `r` is the attribute record. The polymorphic quantification allows future extension, the concrete type will be inferred when running the AG. The second occurrence of `r` reflects the fact that no attribute was renamed in this fragment, otherwise we would have some other type. The next parameter is the list of attributes which the fragment computes. The last parameter is the non-terminal for which the attribute grammar is defined: it will determine the type of inherited attributes that must be given to the root when computing the AG.

8.4.2. Full Listing

```
-- Instruction: Target Language is [Instr]
data Instr
  = LOAD Int | ADD Int | PRINT | HALT

-- Non-Terminals
data PROG
data EXP
data DEF
data DEFS
data FACT


-- Productions
data Print = Print
instance Constructor Print where
  type Production Print =
                   PROG ⟹ '[EXP, DEFS]

data Add = Add
instance Constructor Add where
  type Production Add =
                   EXP ⟹ '[EXP, FACT]

data Fact = Fact
instance Constructor Fact where
  type Production Fact = EXP ⟹ '[FACT]

data Var = Var String
instance Constructor Var where
  type Production Var = FACT ⟹ '[]

data Cst = Cst Int
instance Constructor Cst where
  type Production Cst = FACT ⟹ '[]

data Defnil = Defnil
instance Constructor Defnil where
  type Production Defnil = DEFS ⟹ '[]

data Defcons = Defcons
instance Constructor Defcons where
  type Production Defcons =
                   DEFS ⟹ '[DEF, DEFS]

data Def = Def String Int
instance Constructor Def where
  type Production Def = DEF ⟹ '[]
```

```
-- Smart constructors
iprint   = injC proxies Print
iadd     = injC proxies Add
ifact    = injC proxies Fact
ivar x   = injC proxies (Var x)
icst x   = injC proxies (Cst x)
idefnil  = injC proxies Defnil
idefcons = injC proxies Defcons
idef x y = injC proxies (Def x y)


-- Children Names
type PrintEXP  = N0  -- First child of Print


-- Type of the Abstract Syntax Tree.
type DeskExpr = Expr (CSum
  '[Print, Add, Fact, Var, Cst,
    Defnil, Defcons, Def])


-- Attributes Declaration
data Code = Code
instance Attribute Code where
  type Mode Code = Synthesized
  type Type Code a c n = [Instr]
  type Domain Code = Over '[PROG, EXP]

data Name = Name
instance Attribute Name where
  type Mode Name = Synthesized
  type Type Name a c n = String
  type Domain Name = Over '[DEF]

data Value = Value
instance Attribute Value where
  type Mode Value = Synthesized
  type Type Value a c n = Int
  type Domain Value = Over '[DEF, FACT]

data Ok = Ok
instance Attribute Ok where
  type Mode Ok = Synthesized
  type Type Ok a c n = Bool
  type Domain Ok = Over '[DEFS, FACT]

type SymbolTable = Map String Int
```

```
data EnvS = EnvS
instance Attribute EnvS where
  type Mode EnvS = Synthesized
  type Type EnvS a c n = SymbolTable
  type Domain EnvS = Over '[DEFS]

data EnvI = EnvI
instance Attribute EnvI where
  type Mode EnvI = Inherited
  type Type EnvI a c n = SymbolTable
  type Domain EnvI = Over '[EXP, FACT]

-- Namespace and Aspects
data Desk = Desk
type DeskAspect = IA Desk
     '[Code, EnvS, Ok, Value, Name, EnvI]

-- Semantic Rules for Code
instance (UseS a '[Ok]) ⇒
         SRule c a Desk Code Print where
  srule _ i Print e d  =
    if Ok! d then Code! e ++ [PRINT, HALT]
             else [HALT]

instance (UseS a '[Ok, Value]) ⇒
         SRule c a Desk Code Add where
  srule _ i Add e f =
    if Ok! f
    then Code! e ++ [ADD $ Value! f]
    else [HALT]

instance (UseS a '[Ok, Value]) ⇒
         SRule c a Desk Code Fact where
  srule _ i Fact f =
    if Ok! f then [LOAD $ Value! f]
             else [HALT]

-- Semantic Rules for Value
instance (UseI a '[EnvI]) ⇒
         SRule c a Desk Value Var where
  srule _ i (Var name) = fromJust $
    lookup name (EnvI! i)

instance SRule c a Desk Value Cst where
  srule _ i (Cst x) = x

instance SRule c a Desk Value Def where
  srule _ i (Def name value) = value
```

```
-- Semantic Rules for Ok
instance (UseI a '[EnvI]) ⇒
         SRule c a Desk Ok Var where
  srule _ i (Var name) =
    member name (EnvI! i)

instance SRule c a Desk Ok Cst where
  srule _ i _ = True

instance SRule c a Desk Ok Defnil where
  srule _ i _ = True

instance (UseS a '[Name, EnvS]) ⇒
         SRule c a Desk Ok Defcons where
  srule _ i Defcons h t = Ok! t
    && not (member (Name! h) (EnvS! t))

-- Semantic Rules for Name
instance SRule c a Desk Name Def where
  srule _ i (Def name value) = name

-- Semantic Rules for EnvS
instance SRule c a Desk EnvS Defnil where
  srule _ i Defnil = empty

instance (UseS a '[Name, Value]) ⇒
         SRule c a Desk EnvS Defcons where
  srule _ i Defcons h t =
    insert (Name! h) (Value! h) (EnvS! t)

-- Semantic Rules for EnvI
instance (UseS a '[EnvS]) ⇒ IRule c a
         Desk EnvI Print' PrintEXP where
  irule _ i Print' e d = EnvS ! d

-- Copy Rule for Inherited Attributes
type instance Default Desk = Copy


-- Grammar Fragment
deskF envi = envi `asAttr` EnvI  &
  Code  &  EnvS  &  Ok  &  Name  &  Value


-- Running the Attribute Grammar
runDeskG :: DeskExpr → [Instr]
runDeskG x =
  Code ! runAG' Desk (deskF attrTrivial) x
```

8.5. Related Works

A whole body of work influenced our research. In addition to putting our work in a broader context, this section will also serve to give credit where it is due.

8.5.1. Circular Traversals without Attribute Grammars

Foremost, Bird's article [Bir84] was the catalyst that started me thinking about circular programs and why couldn't we possibly capture the recursive pattern within the language rather than resort to a meta transformation. Bird explains his method through examples without giving an algorithm for a systematic transformation. For one of his example, the palindrome, Bird takes a creative step to obtain a terminating circular program. It is not obvious from his article whether a systematic transformation could be formulated.

Takeichi [Tak87] claims to offer a systematic transformation corresponding to Bird's technique, but what he does is actually very different. Essentially he expresses each traversal as an algebra, specialises a catamorphism combinator by partial application to the datastructure, which by the way corresponds to the polymorphic (a.k.a. Church) encoding of the datastructure, and applies this function to all the algebras. Not only his programs are not circular, but arguably they still perform many traversals, since each algebra is being iterated independently.

Pardo, Fernandes and Saraiva [PFS09] present a variation of foldr/build shortcut fusion rule [GLJ93], which may produce circular programs when it is applied. In particular, GHC is suitable for this technique as it supports the application of user supplied transformation rules. In order for the rule to be triggered, the program must be written in a special way. In particular we must choose whether a function is a producer or a consumer. Surprisingly for the `repmin` example, we need to define `min` as a producer of both the minimum value and a copy of the tree and define `replace` as a consumer. There lies a lack of modularity: in other circumstances `min` would be a consumer.

8.5.2. Circular Traversals with Attribute Grammars

The implementation of AG in functional languages was first studied in 1987 [Joh87, KS87].

Both articles revisit Bird's circular traversals using AG. Johnsson's approach [Joh87] is to formulate the whole program in the AG paradigm and then translate it in a lazy functional program. He writes the recursive function directly (as Bird did), as one monolithic definition. He realises that the encoding of an AG lacks the clarity of its formulation in the AG paradigm and proposes a language construct to write AGs, and gives an algorithm to translate it into the base language.

Kuiper and Swierstra [KS87] give a systematic transformation from multiple traversals (MT) to single circular traversals (CT). Their approach is to use attribute grammars as an intermediate step. They give two formal translations of AG as functional programs: one corresponds to MT, the other to CT. Thus, to transform a MT program to CT, one must identify the MT program with the non-circular implementation of an AG; and from that AG, derive the circular implementation, which results in Bird's CT. Their transformation-based approach isn't modular.

Swiestra, Alcocer and Saraiva's work [SAAS99] was a primary influence for us. They propose a modular approach to circular programming. Traversals are written as algebras and combined using the product operation. It is in fact the underlying principle for our `Circ` implementation of `ArrowCata`. We went one step further than them: whereas they explicitly take the products of the algebras and binding them circularly which adds a lot of complexity to the program compared to the multi-traversal version, we on the other hand, devised combinators around a type that encapsulates circular algebras: this allows us to hide the underlying complexity and implementation details. Consequently, `ArrowCata` programs implemented in the arrow notation are as simple and clear to read and write as are their multi-traversal equivalent written compositionally in functional notation. They also define AG using algebras, following a systematic principle to define complex algebras over mutually recursive datatypes and their iteration. They acknowledge that it is cumbersome to go through the systematic process of deriving the algebra types and combinators, and they suggest to leverage genericity using PolyP. However, they argue that much of the complexity lies in programming directly with algebras and in the second part of the article, they introduce an attribute grammar preprocessor for Haskell. They discuss optimising the generated program. They avoid the circularity by analysing the dependencies between the attributes and generate the minimum of tree traversals as needed: thus each traversal still computes as many attributes as possible.

AG as an Embedded Language Other approaches to circular traversals focus on embedding an AG language in Haskell.

Zipper Based AG A zipper [Hue97] consists of the pair of a subtree and its context. One can "plug" back the subtree into its context and obtain the whole tree. Attributes are translated as mutually recursive functions over the zipper. It should be noted that this approach to AG doesn't lead to circular programs in fact the zipper is traversed many times. Moreover traversing the zipper involved many operations of plugging a subtree into its context, an operation which involves copying the spine of the context (proportional to the depth of the subtree). Alternatively, one can use a cyclic zipper [BFT11], which involves copying the whole tree to start with, but which doesn't need subsequent allocations. Attributes may be recomputed many times but that can be avoided by using memoization [Hin00]. Lastly, this approach to AG offers less modularity than first class embeddings.

Zippers were first used to define AG by Uustalu and Vene [UV05]. They realised that the comonadic structure of trees and zippers has a natural interpretation for AG: the counit extracts the attribute of the root and the cobind extends a function that computes an attribute to a function that computes a decorated tree. They give an abstract interface for defining the functions computing the attributes of an AG. It consists of three typeclasses: `Comonad` of course with methods `counit` and `cobind` already described, `Synth` which gives access to only children's attributes, and `Inh` which gives access to parent's and sibling's attributes. To compute one of those relative attributes it is necessary to use the cobind function first.

Another AG library [MFS13] is built on top of a generic zipper library [Ada10]. An AG translates directly into mutually recursive functions over the zipper of the tree. To access an inherited attribute, one simply calls the corresponding function on the parent node from the zipper's context. The style they advocate risks throwing runtime errors as they access children by number without static checks. This is in fact similar to the partiality we can get when using Moor's first class implementation of AG [MBS00].

First Class AG A survey of first class embeddings is given in §8.4. We will only mention here what differentiate them from direct translations. A first class embedding comes with the possibility to define AG combinators. Which means we can split up an AG into its constituent parts, which we call aspects. It is then possible to replace an aspect with another one to alter a local behaviour. To obtain that sort of modularity in Haskell, one would need parameterised modules: one could affect globally for the whole module which implementation of a function they use, simply by changing the parameter we use to instantiate the module. Another use of AG combinators is to abstract over common programming patterns, and thus promote code reuse. Examples from Kastens and Waite [KW92] include automatic propagation of inherited attributes, computation of chained attributes which capture inorder traversals, etc.

8.5.3. Optimisation of Circular Programs

Our research wasn't actually focused on optimisation, and there is evidently much room for refining our implementation.

Recent compilers optimise much more efficiently the multiple traversals than the circular ones. And whilst the circular transformation used to be viewed as an optimisation in the eighties, it is now the other way around: lazy programs are now optimised by making them stricter, and thus circularity must be removed.

An intermediate approach [CGK99] is to retain circularity whilst maximising strictification when possible. Their algorithm transform MT into efficient CT. A strictness analysis allow to optimise the circular program so that as little laziness is used. They claim better performance over both lazier circular programs and non-circular ones, but it may be that progress in compilers has either reduced or reversed this advantage.

Chapter 9.

Conclusion

9.1. Productivity of Pure Stream Equations

We studied productivity of stream equations. We chose to focus on streams as they are a simple but very common coinductive type. We furthermore restricted our object of study to polymorphic functions defined using only the terminal coalgebra and its inverse, and mutual recursion.

We proved interesting and novel results in this limited setting, showing that those simple means allow the definition of all polymorphic functions, the productivity of which is undecidable. It was crucial to realise that as a consequence of the Yoneda lemma, polymorphic functions are in bijection with their indexing functions: endofunctions on natural numbers, and that productivity of the former is equivalent to totality of the latter, this is the focus of §3.1.3.

We gave three results each improving on the previous one, each characterising more closely the stream equations. The first states that productivity of pure stream equation systems (PSES) is undecidable. The proof is by reduction from a generalised Collatz problem, given in Chapter 4.

The other two results, given in §5.1 and §5.2, showed that PSES and a unary PSES (a further restriction) can define all the polymorphic functions. The proofs are by reduction from a Turing complete language to a PSES, whose indexing function is given by that program. In the last reduction, the pure stream equations considered are limited to only one parameter; however their expressivity is retained and all the polymorphic functions are captured. Further reducing the number of equations to four, retains the expressivity. And further reducing them to two, productivity is still undecidable yet not all polymorphic functions definable. Finally, the productivity of a single equation is decidable. This work showed that unrestricted recursion, and in particular nested recursion, in the otherwise most restricted setting is the main cause for the undecidability of productivity.

Further Research Our research showed that unrestricted recursion makes productivity undecidable. Thus we may want to find useful recursion schemes for coinductive definitions. One such interesting scheme is given by lambda-coiteration [Bar03]. I have developped a Haskell library for writing lambda-coiterative definitions. Type-checking ensures productivity of the definitions. It would be very useful to formalise a general syntactic criterion based on lambda-coiteration. It would find practical applications for the productivity check of theorem-provers and dependent type theories.

9.2. Circular Traversals Compositionally

Summary By investigating the recursive pattern behind the technique of circular traversals, we developed two abstractions, each relying on a different representation of primitive (single) traversals. The first abstraction used algebras. We found out that it was not possible to combine two higher-order algebras (whose carrier is a function) in a single traversal. This prompted us to find a finer representation of higher-order traversals in which the function being computed can only be used in a controlled manner: the representation doesn't have direct access to that function and must instead provide arguments for it and expect results from it. This new representation of traversal coincides with the computation rules of an attribute grammar (AG). We went on to define the circular traversal combinator using this representation.

Main Results In §7.1, we drew the conclusion that higher-order algebras cannot be combined in a single traversal whereas attribute grammars can.

By designing an "Arrow" interface around algebras and attribute grammars, this allows them to be composed in an applicative style which is not usual in the AG paradigm. This approach fits naturally in a functional setting and is to be contrasted with the use of a full blown AG language involving attribute names and many syntactic hurdles.

In addition, we provided two semantically equivalent implementations of the "Arrow" interface allow to execute the same program with different operational costs: one circularly with a single traversal, the other with multiple traversals. In particular, this could be of interest when programming with AG when it is normally necessary to analyse attribute dependencies to avoid a too costly circular definition.

In Chapter 7, we represented primitive traversals as attribute grammars. That representation is novel: we showed that AG semantic rules are best described using a container representation of the base-functor of the tree-type denoted by the grammar.

Containers are dependent types, but we have been able to implement them as well as indexed containers in Haskell. This is explained in §7.3. We discuss in §8.3 the new possibilities offered by containers for parametric generic programming.

The container implementation relies on a novel insight on GADTs, namely the realisation that dependent type families can be viewed as the display maps of GADTs, as we explain in §7.3.2.

In §8.4 we discuss our implementation of a full featured AG system on top of the generic representation of AG semantic rules. This is an embedded language that offers a lot of modularity in the spirit of datatypes à la carte [Swi08].

Further Research There is a wealth of ideas that stems from our work and need to be studied.

We didn't focus on performance. Both the circular library and the attribute grammar DSL need to be tested with more examples, closer to real world cases. Understanding the performance costs associated with circular programs is crucial since laziness plays an important role in the functional paradigm. We could work on compiler optimisation in the presence of circularity and laziness.

Attribute grammars have many advantages for writing modular code, but they are ignored by many functional programmers. Working to better integrate them to functional languages facilitate their adoption. This may involve the design of a new language seamlessly integrating principles from both paradigms, or making better embedding languages.

AG embedded languages are inherently slower than external AG systems like a preprocessor. Could we use introspection so that an AG embedded language could optimise the number of traversals to avoid circularity when necessary?

Containers seem have a great potential for generic programming, it would be interesting to design more algorithms with containers in Haskell. A good choice would be the zipper.

Finally, to make the connection between the two parts of my thesis: circular programs and productivity, we could implement an attribute grammar system in type-theory, where the corecursion and the proof of productivity would be explicit using the partiality monad. This work would have the benefit of giving more flexibility to define corecursive programs than the usual guardedness criteria.

References

[AAG05] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[AAGM05] M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. δ for data: derivatives of data structures. *Fundamenta Informaticae*, 65:1–128, March 2005.

[Abb03] Michael Abbott. *Categories of Containers.* PhD thesis, University of Leicester, October 2003.

[Abe10] Andreas Abel. Miniagda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Workshop on Partiality And Recursion in Interative Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010*, 2010.

[Ada10] Michael D. Adams. Scrap your zippers: A generic zipper for heterogeneous types. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 13–24, New York, NY, USA, 2010. ACM.

[AJ94] Samson Abramsky and Achim Jung. Domain Theory. In Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, Oxford, 1994.

[AM09] Thorsten Altenkirch and Peter Morris. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009. to appear.

[Bar03] Falk Bartels. Generalised coinduction. *Mathematical Structures in Computer Science*, 13(2):321–348, 2003.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BdM97] Richard S. Bird and Oege de Moor. *Algebra of programming.* Prentice Hall International series in computer science. Prentice Hall, 1997.

[Ber05] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005*, pages 102–115. Springer-Verlag, 2005.

[BFT11] Eric Badouel, Bernard Fotsing, and Rodrigue Tchougong. Attribute grammars as recursion schemes over cyclic representations of zippers. *Electronic Notes in Theoretical Computer Science*, 229(5):39 – 56, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

[BH11] Patrick Bahr and Tom Hvitved. Compositional data types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, pages 83–94, New York, NY, USA, 2011. ACM.

[BH12] Patrick Bahr and Tom Hvitved. Parametric compositional data types. In James Chapman and Paul Blain Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–24. Open Publishing Association, feb 2012.

[Bir84] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf.*, 21:239–250, 1984.

[BJ66] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

[Buc05] Wilfried Buchholz. A term calculus for (co-)recursive definitions on streamlike data structures. *Ann. Pure Appl. Logic*, 136(1-2):75–90, 2005.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[Cap10] Venanzio Capretta. Bisimulations generated from corecursive equations. *Electronic Notes in Theoretical Computer Science*, 265:245 – 258, 2010. Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2010).

[Cap11] Venanzio Capretta. Coalgebras in functional programming and type theory. *Theoretical Computer Science*, 412(38):5006–5024, 2011. CMCS Tenth Anniversary Meeting.

[CD82] Robert Cartwright and James E. Donahue. The semantics of lazy (and industrious) evaluation. In *Symposium on LISP and Functional Programming*, pages 253–, 1982.

[CGK99] Wei-Ngan Chin, Aik-Hui Goh, and Siau-Cheng Khoo. Effective optimization of multiple traversals in lazy languages. In *PEPM*, pages 119–130, 1999.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM*

SIGPLAN International Conference on Functional Programming, ICFP 2007, April 2007. To appear.

[Con72] John Horton Conway. Unpredictable iterations. In *Number Theory Conference*, pages 49–52. University of Colorado, 1972.

[Con87] John H. Conway. Fractran: A simple universal programming language for arithmetic. In T. M. Cover and B. Gopinath, editors, *Open Problems in Communication and Computation*, chapter 2, pages 4–26. Springer, 1987.

[Coo91] William R. Cook. Object-Oriented Programming Versus Abstract Data Types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, 1991. Springer-Verlag.

[Coo03] S. Barry Cooper. *Computability Theory.* Chapman & Hall / CRC mathematics, 2003.

[Dan10] Nils Anders Danielsson. Beating the productivity checker using embedded languages. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers*, volume 43 of *EPTCS*, pages 29–48, 2010.

[DC90] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Comput. J.*, 33(2):164–172, April 1990.

[DGM03] Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 618–618. Springer Berlin / Heidelberg, 2003.

[Dij80] Edgar W. Dijkstra. On the producitivity of recursive definitions. EWD749, 1980.

[dMBS00] Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.

[DP11] Facundo Domínguez and Alberto Pardo. Exploiting algebra/coalgebra duality for program fusion extensions. In *Proceedings of the 11th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2011).* To be published by ACM, 2011.

[Dum77] Michael Dummett. *Elements of Intuitionism.* Clarendon Press, 1977.

[EGH$^+$07] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. In *Proceedings of FCT 2007*, number 4639 in LNCS, pages 274–287. Springer, 2007.

[EGH08] Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Data-oblivious stream productivity. In *LPAR*, pages 79–96, 2008.

[EGH09a] Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Complexity of fractran and productivity. *CoRR*, abs/0903.4366, 2009.

[EGH09b] Jrg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Complexity of Fractran and productivity. In *CADE*, pages 371–387, 2009.

[EGSZ11] Jörg Endrullis, Herman Geuvers, Jacob G. Simonses, and Hans Zantema. Levels of undecidability in rewriting. *Inf. Comput.*, 209(2):227–245, 2011.

[Ell09] C. Elliott. Denotational design with type class morphisms. *extended version), LambdaPix*, 2009.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[FSSV11] João Paulo Fernandes, João Saraiva, Daniel Seidel, and Janis Voigtländer. Strictification of circular programs. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 131–140, New York, NY, USA, 2011. ACM.

[GH09] Andy Gill and Graham Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[GHK$^+$03] Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael Mislove, and Dana S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2003.

[GJ98] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *ICFP*, pages 273–279, 1998.

[GLJ93] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, 1993*, 1993.

[GP09] Jeremy Gibbons and Ross Paterson. Parametric datatype-genericity. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, WGP '09, pages 85–93, New York, NY, USA, 2009. ACM.

[Gru06] Dominik Gruntz. Infinite streams in java. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, PPPJ '06, pages 182–187, New York, NY, USA, 2006. ACM.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[Hal00] Thomas Hallgren. Fun with functional dependencies. In *Proc. of the Joint CS/CE Winter Meeting*, 2000.

[Har80] David Harel. On folk theorems. *SIGACT News*, 12:68–80, September 1980.

[Har11] Thomas Harper. A library writer's guide to shortcut fusion. In *Haskell Symposium 2011*, September 2011.

[Hed99] Gorel Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 153–172, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[HHJ11] Ralf Hinze, Thomas Harper, and Daniel W.H. James. Theory and practice of fusion. Technical Report CS-RR-2011-01, Department of Computer Science, University of Oxford, 2011.

[Hin89] Andreas M. Hinz. The Tower of Hanoi. *Enseign. Math.*, 35(2):289–321, 1989.

[Hin00] Ralf Hinze. Memo functions, polytypically! In *Proceedings of the 2nd Workshop on Generic Programming, Ponte de*, pages 17–32, 2000.

[Hin11] Ralf Hinze. Concrete stream calculus—an extended study. *JFP*, 20(5-6):463–535, 2011.

[Hue97] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[Hug98] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[Jac05] Bart Jacobs. Draft: Introduction to coalgebra. towards mathematics of states and observations, 2005.

[JL97] Patricia Johann and John Launchbury. Warm fusion for the masses: Detailing virtual data structure elimination in fully recursive languages. Available on the Internet, 1997.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, volume 274 of LNCS*, pages 154–173. Springer-Verlag, 1987.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[JV00] Patricia Johann and Eelco Visser. Warm fusion in stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29:1–34, 2000.

[Kle43] Stephen C. Kleene. Recursive predicates and quantifiers. *Trans. AMS*, 53(1):41–73, 1943.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[KS87] M. F. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. In *IN COMPUTING SCIENCE IN THE NETHERLANDS CSN'87*, 1987.

[KS07] Stuart A. Kurtz and Janos Simon. The undecidability of the generalized Collatz problem. In *TAMS*, volume 4484 of *LNCS*, pages 542–553. Springer, 2007.

[KW92] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *ACTA INFORMATICA*, 31:601–627, 1992.

[Lag06] J.C. Lagarias. The 3x+1 problem: An annotated bibliography (1963-2000). Technical report, ArXiv math (NT0608208), 2006.

[Lam68] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *FPCA*, pages 314–323, 1995.

[Mar10] Simon Marlow. Haskell 2010 language report. `http://www.haskell.org/definition/haskell2010.pdf`, 2010.

[Mat99] John Matthews. Recursive function definition over coinductive types. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 839–839. Springer Berlin / Heidelberg, 1999.

[MBS00] Oege De Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica*, 24:2000, 2000.

[McB01a] Connor McBride. Faking it: Simulating dependent types in haskell, 2001.

[Mcb01b] Conor Mcbride. The derivative of a regular type is its type of one-hole contexts (extended abstract). 2001.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, pages 124–144, 1991.

[MFS13] Pedro Martins, João Paulo Fernandes, and João Saraiva. Zipper-based attribute grammars and their extensions. In *Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3 - 4, 2013. Proceedings*, pages 135–149, 2013.

[Min67] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[ML84] P. Martin-Lof. Constructive mathematics and computer programming. *Royal Society of London Philosophical Transactions Series A*, 312:501–518, October 1984.

[MLAZ99] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Multiple Attribute Grammar Inheritance. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 57–76, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[Mor07] Peter Morris. *Constructing universes for generic programming*. PhD thesis, Department of Computer Science, University of Nottingham, 2007.

[MP08] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[Nem00] L. Nemeth. *Catamorphism-based program transformations for non-strict functional languages*. University of Glasgow, 2000.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[NW00] Dung Nguyen and Stephen B. Wong. Design patterns for lazy evaluation. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 21–25, New York, NY, USA, 2000. ACM.

[Paa95] Jukka Paakki. Attribute grammar paradigms&mdash;a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995.

[Pat01] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[PE98] Dusko Pavlovic and Martín Hötzel Escardó. Calculus in coinductive form. In *LICS*, pages 408–417, 1998.

[Per74] Eleonora Perkowska. Theorem on the normal form of a program. *Bull. Acad. Pol. Sci., Ser. Sci. Math. Astr. Phys.*, 22(4):439–442, 1974.

[PFS09] Alberto Pardo, João Paulo Fernandes, and João Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 81–90, New York, NY, USA, 2009. ACM.

[Roş06] Grigore Roşu. Equality of streams is a Π_2^0-complete problem. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06).* ACM, 2006.

[Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[Rut03a] Jan J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theor. Comput. Sci.*, 308(1-3):1–53, 2003.

[Rut03b] Jan M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theor. Comp. Sci.*, 308(1-3):1–53, 2003.

[SAAS99] S. D. Swierstra, P. R. Azero Alocer, and J. Saraiava. Designing and implementing combinator languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

[SB12] Christian Sattler and Florent Balestrieri. Turing-Completeness of Polymorphic Stream Equation Systems. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 256–271, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[SBMG07] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483. Springer Berlin Heidelberg, 2007.

[Sch86] David A. Schmidt. *Denotational semantics: a methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[Sch00] Sibylle Schupp. Lazy lists in c++. *SIGPLAN Not.*, 35(6):47–54, June 2000.

[Sij89] Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11:633–649, October 1989.

[Sim09a] Jakob Grue Simonsen. The Π_2^0-completeness of most of the properties of rewriting systems you care about (and productivity). In *Proc. 20th Int. Conf. on RTA*, RTA '09, pages 335–349. Springer, 2009.

[Sim09b] Jakob Grue Simonsen. The pi-2-0-completeness of most of the properties of rewriting systems you care about (and productivity). In *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 335–349. Springer, 2009.

[SL07] Tim Sheard and Nathan Linger. Programming in omega. In *CEFP*, pages 158–227, 2007.

[Soa87] Robert I. Soare. *Recursively Enumerable Sets and Degrees.* Perspectives in Mathematical Logic. Springer, 1987.

[SS03] João Saraiva and Sérgio Schneider. Embedding domain specific languages in the attribute grammar formalism. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9*, HICSS '03, pages 324.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13:11–49, April 2000.

[Swi05] Wouter Swierstra. Why Attribute Grammars Matter. *The Monad.Reader*, 4, July 2005.

[Swi08] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[Tak87] Masato Takeichi. Partial parametrization eliminates multiple traversals ofdata structures. *Acta Informatica*, pages 42–57, 1987.

[Tay99] Paul Taylor. *Practical Foundations of Mathematics*, volume 59 of *Cambridge studies in advanced mathematics.* Cambridge University Press, 1999.

[Tea] The GHC Team. Ghc user guide.

[TT97a] Alastair Telford and David Turner. Ensuring streams flow. In *Proc. 6th AMAST*, pages 509–523. Springer, 1997.

[TT97b] Alastair Telford and David Turner. Ensuring the Productivity of Infinite Structures. Technical Report 14-97, The Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK, September 1997. This technical report has been revised (March 1998). A shorter version of this paper was presented at AMAST '97.

[UV99] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica, Lith. Acad. Sci.*, 10(1):5–26, 1999.

[UV05] Tarmo Uustalu and Varmo Vene. Comonadic functional attribute evaluation. In M. van Eekelen, editor, *Proceedings of 6th Symposium on Trends in Functional Programming*, Trends in Functional Programming, pages 33–43, Tallinn, september 2005. Intellect.

[Ven00] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis (Diss. Math. Univ. Tartuensis 23), Dept. of Computer Science, Univ. of Tartu, August 2000.

[VS12] Marcos Viera and S. Doaitse Swierstra. Attribute grammar macros. In *SBLP*, pages 150–164, 2012.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Not.*, 24(7):131–145, June 1989.

[VSS09] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. *SIGPLAN Not.*, 44:245–256, August 2009.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[Win93] Glynn Winskel. *Formal Semantics of Programming Languages: an Introduction, The.* MIT Press, 1993.

[Wra89] G. C. Wraith. A note on categorical datatypes. In *Category Theory and Computer Science*, pages 118–127, London, UK, 1989. Springer-Verlag.

[YHT] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Calculation rules for warming-up in fusion transformation.

[Zan10] Hans Zantema. Well-definedness of streams by transformation and termination. *LMCS*, 6(3), 2010. paper 21.