Università di Genova
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

Enhancing Regular Corecursion

Pietro Barbieri

Theses Series DIBRIS-TH-2023-42

Ph.D. Thesis

Università di Genova
Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi
Ph.D. Thesis in
Computer Science and System Engineering
Computer Science Curriculum

Enhancing Regular Corecursion

by

Pietro Barbieri

December 2023

Ph.D. Thesis in Computer Science and System Engineering (S.S.D. INF/01)
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova

Candidate
Pietro Barbieri
`pietro.barbieri@edu.unige.it`

Title
Enhancing Regular Corecursion

Advisors
Davide Ancona
DIBRIS, Università di Genova
`davide.ancona@unige.it`

Elena Zucca
DIBRIS, Università di Genova
`elena.zucca@unige.it`

External Reviewers
Paola Giannini,
Università del Piemonte Orientale, Vercelli (IT),
`paola.giannini@uniupo.it`

Jurriaan Rot,
Radboud University, Nijmegen (NL),
`jrot@cs.ru.nl`

Alexandra Silva,
Cornell University Ithaca, New York,
`alexandra.silva@cornell.edu`

Location
DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy

Submitted On
December 2023

Abstract

Nowadays, data structures which are conceptually infinite, such as streams or infinite trees, are very common in computer science. When it comes to their manipulation, one major problem to face is how to finitely represent and deal with them without incurring in non-terminating behaviours. *Regular corecursion* is a solution relying on finite representation of regular data structures, and detection of cyclic calls. The topics in the thesis revolve around two enhancements of regular corecursion in different directions. In the first part, we present Corecursive Featherweight Java (coFJ), an object-oriented calculus which supports *flexible* regular corecursion, that is, allows the programmer to specify the behaviour when a cyclic call is found. In the second part, instead, we extend regular corecursion *beyond regular terms*, focusing on the significant case of stream definitions.

Acknowledgements

I would like to express my heartfelt thanks to my supervisors Elena Zucca and Davide Ancona for their unwavering support over the years, as their help and guidance have been indispensable in pursuing my research and reaching this important milestone. During my time as a student, they thought me how to do research and encouraged me to improve day after day, never letting me down in difficult moments.

I would also like to extend my deepest gratitude to Fracesco Dagnino for his valuable insights and suggestions over the years. Fracesco is the most talented young researcher I have ever met and, besides feeling very fortunate to have worked with him, I owe him a great deal for several of the results of this thesis.

Special thanks to Paola Giannini, Jurriaan Rot and Alexandra Silva for their careful review of my thesis. Their comments were very helpful in improving my work and broadening my perspective on the work related to my thesis.

This achievement would not have been possible without the support and nurturing of my family. My parents Ida and Carlo always supported my decisions and never doubted me, while my brothers Daniele and Luca always thought I was the best, even at times when my behaviour did not make me deserve it.

Among my family members, I would like to thank Davide, my cousin, for being one of the best friends one could wish for, especially in those moments when I knew I could share my thoughts and feelings with him knowing that he would understand perfectly.

Special thanks also goes to my grandparents Lino, Aurelia and Assunta, and to my uncles Nicoletta, Giulio and Enrico, who were very important for my growth as a man and to whom I owe a great deal.

At this moment in my life I am undoubtedly happy, not only for my achievement or my wonderful family, but above all for Lara, my life partner, who is the best thing that has happened to me in a long time. We have shared some beautiful moments together and I am sure we will have many more as life goes on.

I have many friends to thank and had great difficulty in putting together a complete list for my thanks. I must certainly start with my lifelong friends from Montebruno, my home town, because we have known each other for ages and have spent countless moments together. Among them, a special thanks goes to Federico, Filippo, Giacomo and Mattia, with whom I always enjoy spending time with, whether it be to talk, or for a pizza, or to defeat some evil creature in the lands of Azeroth.

My special thanks also go to my friends Andrea, Luca and Tomaso with whom I shared so many fun moments at the G.D. Cassini scientific high school in Genova. We have kept in touch ever since and, despite being scattered all over Italy and Europe, I am always pleased to meet them and share our

adventures.

Lastly, I would like to thank all the friends I met during my eight years at the University of Genova. I will carry with me fantastic memories of that period, and I will intentionally not make a list here because I would be very sorry to forget even one person. Of all the people I met at DIBRIS, I owe particularly much to Luca Ciccone, who is probably one of the best friends I have ever had. Whether talking about ordinary stuff, or 70s prog rock, or even serious things, Luca always has interesting insights and has never let me down as a friend.

Contents

Introduction 1

1 Introduction 5

I Flexible corecursion in coFJ 13

2 Preliminary notions 15
2.1 Generalized inference systems 15
2.2 Syntax and semantics of FJ 17
2.3 Towards infinite objects and codefinitions 19

3 coFJ and its abstract semantics 23
3.1 Formal definition of coFJ 23
3.2 Examples 24
3.3 Non-determinism and conservativity 25

4 coFJ operational semantics 29
4.1 Formal definition 29
4.2 Determinism and conservativity 35

5 coFJ intermediate semantics 37
5.1 Formal definition 38
5.2 Examples 40
5.3 Discussion 43

6 Advanced examples 45

7 Soundness 51
7.1 From capsules to infinite objects 51
7.2 Statement and proof 53

8 Related and future work 61

II Beyond regular terms 65

9 Stream calculus 67
9.1 Formal definition 67
9.2 Examples 72

10 Well-definedness check 75

11 Expressive power 83
11.1 Streams and polynomials 83
11.2 Other binary operators on streams 84

12 Extended calculus 93

13 Extended well-definedness check 97
13.1 Extended definition 97
13.2 Examples 100
13.3 Soundness and completeness of the extended well-definedness check 103

14 Equality of streams 107
14.1 Formal definition 107
14.2 Soundness and (relative) completeness of the equality check . 110
14.3 Towards an algorithm for equality 115

15 Related and future work 123

Conclusion 127

16 Conclusion 129
16.1 Future work 130

Bibliography 131

Introduction

f

1 Introduction

Nowadays, applications often deal with data structures which are conceptually infinite, such as streams or infinite trees. Thus, a relevant problem in the design of programming languages is how to finitely represent something which is infinite, and, even harder, how to correctly manipulate such finite representations to reflect the expected behaviour on the infinite structure.

Well-established solutions to this problem are *lazy evaluation* and *coinductive data types*. In both such approaches, it is possible to represent arbitrary infinite structures, and to define in the standard inductive way functions which only require a finite portion of the structure to be examined, such as getting a finite prefix of a stream. A different approach, originally coming from (coinductive) logic programming, recently adapted to other programming paradigms, is *regular corecursion*, allowing, roughly, to obtain a result even for functions which conceptually require to examine the whole infinite structure, provided the latter is *regular*. The drawback is that non-regular structures cannot be handled. In this introduction, to provide the research context, we first briefly illustrate the above mentioned approaches, and then we describe the aims of the thesis.

LAZY EVALUATION A widely-used solution for the generation and manipulation of conceptually infinite data is *lazy evaluation*, as supported, e.g., in Haskell, and most stream libraries offered by mainstream languages, as `java.util.stream`. In Haskell, infinite data are obtained as the result of a recursive function, which is evaluated according to the call-by-need strategy.

For instance, we can declare the following functions

```
one_two = 1:2:one_two
from n = n:from(n+1)
```

The calls `one_two` and `from 0` represent the infinite list alternating 1 and 2, and the list of natural numbers, respectively. Typical recursive functions defined by pattern-matching (that is, inductively) on lists *may* work on such infinite lists as well, thanks to lazy evaluation. More precisely:

- functions which inspect only a finite portion, e.g., `take` returning a finite prefix, terminate.
- Functions which inspect the whole structure, like `member` checking whether an element belongs to the list, or `allpos` checking that all elements are positive, or `min` looking for the minimal element, may diverge; notably,

member and allpos only terminate in the positive and negative case, respectively, and min never terminates. Note that this happens for all infinite lists, even those for which the result could be obtained by a finite number of checks, as one_two.

- Finally, functions which return an infinite list, e.g., incr adding 1 to every element, can be lazily used as the original one; for instance, take 5 (incr one_two) terminates.

Finally, note that intuitively ill-formed definitions such as bad_stream = bad_stream are accepted by the Haskell compiler. Anyway, any operation which needs to inspect bad_stream is deemed to diverge because the corresponding stream is undetermined. Unfortunately, it is not decidable to check, even at runtime, whether the stream returned by a Haskell function is *well-defined*, that is, all of its elements can be computed[1]; indeed, the full expressive power of Haskell can be used to define streams by means of arbitrary recursive definitions. For similar reasons, it is not decidable to check at runtime whether the streams returned by two Haskell functions are equal.

COINDUCTIVE DATA TYPES Besides standard inductive data structures, proof assistants generally support the definition of *coinductive* structures. For instance, in Agda streams can be implemented by *coinductive records*. Using a coinductive record, we represent a data structure through the *observations* which can be made on it. For instance, a stream is completely determined by its *head* and its *tail*, which is a stream in turn, as shown below.

```
record Stream (A : Set) : Set where
coinductive
  field
    head : A
    tail : Stream A
```

An alternative solution is to use the mechanism of *thunks*. That is, roughly, a standard (inductive) type can be used, even though the data to be represented is infinite, where the tail has the type Thunk of suspended computations, used to simulate laziness. The following is a simplified version[2] of the library *colists* which are either finite or infinite lists:

```
data Colist (A : Set) : Set where
[] : Colist A
_::_ : A → Thunk (Colist A) → Colist A
```

In both cases, it should be noted that the point of view and aim of a proof assistant are different from that of a programming language. In particular, functions in Agda are required to be *total*, and this is ensured by a termination checker. For this reason:

- recursive functions which only inspect a finite portion can be defined, provided that enough information is given to the termination checker.

1 This is also known as a *productive* corecursive definition [14].
2 Omitting sizes which track the depth of data structures.

- Recursive functions which inspect the whole structure, hence would not terminate on an infinite list, *cannot even be defined*; however, they can be expressed as *predicates*, that is, as (dependent) types, exactly as data structures.
- Finally, functions which return an infinite list can be *coinductively* defined.

To exemplify the second point, the above mentioned `member`, `allpos`, and `min`, cannot be expressed as functions on `Stream` or `Colist`; however, we can turn them into predicates, as shown below for the first two:

```
data _member_ {A : Set}(x : A)(xs : Stream A) : Set where
  mem-h : x ≡ (Stream.head xs) → x member xs
  mem-t : x member (Stream.tl xs) → x member xs

record allpos (xs : Stream Nat) : Set where
coinductive
  field
    h : (Stream.head xs) > zero
    t : allpos (Stream.tail xs)
```

Note that the predicate `member` is defined *inductively*, whereas the predicate `allpos` is defined *coinductively*. It is worth noticing that the predicate `min` cannot be defined as a coinductive record in the natural way, since the coinductive interpretation would not give the expected meaning [13]. We explain this more in detail later on in the case of logic programming.

In this way, we can reason about the predicate, e.g., we can prove that the conductive definition of `allpos` is sound and complete with respect to the following specification using `member`:

```
(xs : Stream Nat) → ({n : Nat} → n member xs → n > 0)
```

However, we cannot provide an algorithmic procedure that, given a stream (or a colist), returns the expected result in as many cases as possible, as we would expect in a programming language.

REGULAR CORECURSION More recently, a different, in a sense complementary, approach, has been considered, called *regular corecursion*, which originates from *co-SLD resolution* [30, 31, 8], as briefly illustrated below. Let us consider the Prolog version of the predicates `member` and `allpos`.

```
member([X],X).
member([X|Xs],Y) ← member(Xs,Y)

allpos([])     ←
allpos([X|Xs]) ← X>0, all_pos(Xs)
```

In standard logic programming terms are defined inductively (that is, are finite), and predicates are defined inductively as well (that is, as sets of atoms which have a finite proof tree, or, equivalently, as least fixed points). To handle infinite structures, in coinductive logic programming (coLP), introduced in [30], predicates are defined on *infinite terms* (that is, terms are defined coinductively),

and can be defined coinductively as well (that is, as set of atoms which have a possibly infinite proof tree, or, equivalently, as greatest fixed points). In the example, also infinite lists, such as `[1,2,3,4,...]`, are considered, and the coinductive interpretation of `allpos` gives the expected meaning on such lists. As the reader can note, this is quite similar to defining a coinductive predicate in Agda. However, what happens *at the operational level*, hence in the language implementation, is very different. Indeed, the operational semantics only handles *regular* terms, that is, those which can be represented by equations, such as `L = [1,2|L]`. The corresponding algorithmic procedure (*co-SLD resolution*) is a modification of SLD resolution where, when the same goal is found the second time, it is considered successful. Hence, in the above example, the goal `all_pos(L)` succeeds.

As already noted in Agda, the coinductive interpretation does not always correspond to the intended meaning; for instance, for the predicate `member` the inductive interpretation works on infinite lists as well. Moreover, in some cases neither the inductive, nor the coinductive interpretation are correct. For instance, in the following logic program:

```
min([X],X) ←
min([X|Xs],M) ← min(Xs,M1), M is min(X,M1)
```

with the purely coinductive interpretation, if `l` is an infinite list, then the goal `min(l,n)` succeeds whenever `n` is less or equal than all the elements of `l`. As mentioned, the same happens in Agda, where the predicate cannot be defined in a purely coinductive way [13].

In summary, the key idea of regular corecursion is *cycle detection*, so that non-termination can be avoided, and an alternative action can be taken. In the original case of coinductive logic programming, cycle detection means detecting *the same goal* (modulo unification), and the alternative action is simply success.

OUTLINE The aim of this thesis is to enhance regular corecursion in two different directions: on one hand, to make it *flexible*, that is, to allow the programmer to specify the alternative action to be taken when a cyclic call is found; on the other hand, to extend regular corecursion *beyond regular terms*.

PART I - FLEXIBLE REGULAR CORECURSION IN coFJ A limitation of regular corecursion is that it is not flexible enough to correctly express certain predicates on regular terms, as shown by the above `min` example. In the logic paradigm, flexibility can be achieved by adding *coclauses*, which allow one to refine the coinductive interpretation, by, roughly, triggering another SLD-resolution when the same goal is found the second time [18]. When moving from goals to functions/methods calls, the same problem manifests more urgently, because a result should always be provided for already encountered calls. To solve this issue, the programmer should be allowed to specify the behaviour of recursive functions/methods on cyclic structures. For instance, in order to get the correct behaviour for function `min`, the programmer has to

specify that the head of the list should be returned when detecting a cyclic call. Language extensions supporting such flexible regular corecursion has been proposed in the object-oriented [9] and functional [22, 23] paradigms (see Chapter 8 for more details). However, none of these proposals provides formal arguments for the correctness of the given operational semantics, by proving that it is sound with respect to some model of the behaviour of functions (or predicates) on infinite structures. We want to bridge this gap by providing solid foundations for a programming paradigm natively supporting cyclic data structures.

CHAPTER 2 We discuss preliminary notions on inference systems, present the framework of *generalized inference systems*, and the *Featherweight Java* calculus, which will be the starting points of our extension.

CHAPTER 3 We introduce the coFJ calculus and present its *abstract semantics*, which will be used as reference semantics, and is given by a generalized inference system which manipulates infinite objects.

CHAPTER 4 We present the coFJ *operational semantics*, which, in contrast to the abstract one, is meant to work on finite cyclic objects, thus making it algorithmic and executable.

CHAPTER 5 We introduce a third semantics for coFJ with the aim of bridging the gap between the abstract and the operational one. The intermediate semantics manipulates infinite objects, but is based on the framework of regular corecursion.

CHAPTER 6 We show some examples of usage of coFJ and discuss several features of the calculus.

CHAPTER 7 We prove the soundness of the operational semantics with respect to the abstract one. This means that a result obtained from the former can also be obtained from the latter. The proof uses the intermediate semantics introduced in Chapter 5.

CHAPTER 8 We discuss related work and outline directions for future work concerning this part.

PART II - BEYOND REGULAR TERMS The second extension to regular corecursion aims at overcoming the limitations imposed by regular terms. In this part, our reference language is a simple functional language of numeric streams, which are an example of infinite data structure relevant for IoT applications. In the operational semantics, equations representing streams are extended with other typical stream operators besides the constructor. In such generalized approach, some non-regular streams can be represented as

well, as shown by the function `nat` below which returns the stream of natural numbers.

```
nat() = 0:(nat() [+] repeat(1))
```

where : is the stream constructor, `[+]` the pointwise addition and `repeat(1)` the stream constantly equal to `1`.

The calculus is presented in two flavours: in the first version, stream operators allowed in equations are tail and pointwise numeric operators. After discussing the expressive power of this version, we provide a second one where equations can also contain an *interleaving* operator.

In this part, for simplicity, we do not consider the orthogonal feature of making corecursion flexible, which has been deeply analyzed in the first part. Instead, we consider two additional features which become very challenging when additional operators are allowed in equations: *well-definedness* and *equality* of stream definitions.

A well-definedness check is added when a function returning a stream is invoked, to ensure that the result of the call is actually a stream, in the sense that the access to an arbitrary index will always succeed. To design a sound and complete well-definedness check for the first version of the calculus is rather simple, whereas the version with the interleaving operator requires a more tricky definition which, however, can still be proved to be sound and complete through a symbolic computation which mimics the access to an arbitrary index.

Moreover, for the first version of the calculus, we design an algorithm to check whether two different stream definitions have the same semantics. This is important, since cycle detection relies on stream equality, and allowing additional operators in equations makes much harder to detect that two definitions denote the same stream.

Chapter 9 We introduce the stream calculus with its operational semantics and show examples of streams that can be defined.

Chapter 10 We introduce the notion of well-definedness, and provide a sound and complete operational characterization.

Chapter 11 We discuss the expressive power of the calculus and prove that it is more expressive than polynomial streams. Then, we discuss some derived operators.

Chapter 12 We extend the calculus with an *interleaving* operator which gives a stream whose elements are alternatively those of the arguments.

Chapter 13 We introduce an extended well-definedness check that takes into account the interleaving operator.

Chapter 14 We tackle the problem of equality of stream definitions and provide a sound operational characterization.

Chapter 15 We discuss related work and outline directions for future work concerning this part.

relationship with published and submitted papers The content of Part I mainly originates from two papers published at *ECOOP 2020* [3] and *FTfJP 2020* [11]. In [3] we focused on the relation between coFJ abstract and operational semantics, introducing the intermediate one as a tool to prove the main soundness result. After that, in [11] we studied this latter semantics more in detail, and found it interesting in itself, since it shows that the mechanism of cycle detection and the representation of infinite values are independent issues. Here the discussion is more structured and streamlined than in the papers, with the intermediate semantics playing a crucial role in the study of coFJ.

The content of Part II is the result of my last two years of research. In particular, Chapter 9, concerning the syntax and semantics of the stream calculus, and Chapter 10, with the preliminary formalization of the well-definedness check, originate from a paper published at *ICTCS 2021* [4], where we presented the calculus for the first time. Chapter 11, Chapter 12 and Chapter 13, with the discussion on the expressive power and the enhanced version of the stream calculus and well-definedness check, originate from a paper published at *FLOPS 2022* [5] and an extension of the *ICTCS'21* paper submitted to the *Theoretical Computer Science* journal. Lastly, Chapter 14, concerning the equality check, originates from a paper published at *ICTCS 2022* [5] and its subsequent extension, which we are going to submit to *TCS*. With respect to all these papers, here we present the calculus and its features in their entirety, without space constraints and in a more streamlined fashion. In particular, our aim in the thesis is to introduce all the elements of the calculus in an incremental manner, so that we can study the impact of each enhancement separately.

PART I

Flexible corecursion in COFJ

2

Preliminary notions

In this chapter we present the starting points for defining coFJ: the standard notions on inference systems [1, 26], their recent generalization to *inference systems with corules* [7, 16, 17], and *Featherweight Java* (FJ) [21], a tiny subset of Java which has become the reference calculus to investigate properties and extensions of Java-like languages.

2.1 Generalized inference systems

Assuming a *universe* $\mathcal{U}$ of *judgments*, an *inference system* $\mathcal{I}$ is a set of *(inference) rules*, which are pairs $\frac{Pr}{c}$, with $Pr \subseteq \mathcal{U}$ the set of *premises*, and $c \in \mathcal{U}$ the *consequence* (a.k.a. *conclusion*). A rule with an empty set of premises is an *axiom*. A *proof tree* (a.k.a. *derivation*) for a judgment j is a tree whose nodes are (labelled with) judgments, j is the root, and there is a node c with children Pr only if there is a rule $\frac{Pr}{c}$.

The *inductive* and the *coinductive interpretation* of $\mathcal{I}$, denoted $Ind(\mathcal{I})$ and $CoInd(\mathcal{I})$, are the sets of judgments with, respectively, a finite[1], and a possibly infinite proof tree. In set-theoretic terms, let $F_{\mathcal{I}} : \wp(\mathcal{U}) \to \wp(\mathcal{U})$ be defined by $F_{\mathcal{I}}(S) = \{c \mid Pr \subseteq S, [Pr]c \in \mathcal{I}\}$. That is, $F_{\mathcal{I}}(S)$ is the set of judgments that can be inferred (in one step) from the judgments in S using the inference rules. We say that a set S is *closed* if $F_{\mathcal{I}}(S) \subseteq S$, *consistent* if $S \subseteq F_{\mathcal{I}}(S)$, that is, no new judgments can be inferred from a closed set, and all judgments in a consistent set can be inferred from the set itself. Then, it can be proved that $Ind(\mathcal{I})$ is the smallest closed set, and $CoInd(\mathcal{I})$ is the largest consistent set.

Inference systems have been recently generalized [7, 16, 17] as described below. An *inference system with corules*, or *generalized inference system*, is a pair $(\mathcal{I}, \mathcal{I}^{co})$ where $\mathcal{I}$ and $\mathcal{I}^{co}$ are inference systems, whose elements are called *rules* and *corules*, respectively. Corules can only be used in a special way, as defined in the following.

For a subset S of the universe, let $\mathcal{I}_{\sqcap S}$ denote the inference system obtained from $\mathcal{I}$ by keeping only rules with consequence in S. Let $(\mathcal{I}, \mathcal{I}^{co})$ be a generalized inference system. Then, its *interpretation* $Gen(\mathcal{I}, \mathcal{I}^{co})$ is defined by $Gen(\mathcal{I}, \mathcal{I}^{co}) = CoInd(\mathcal{I}_{\sqcap Ind(\mathcal{I} \cup \mathcal{I}^{co})})$.

In proof-theoretic terms, $Gen(\mathcal{I}, \mathcal{I}^{co})$ is the set of judgments that have a

1 Under the common assumption that sets of premises are finite, otherwise we should say well-founded.

possibly infinite proof tree in $\mathcal{I}$, where all nodes have a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{co}$, that is, the (standard) inference system consisting of rules and corules.

Note that a finite proof tree in $\mathcal{I}$ is a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{co}$ as well, hence the condition is only significant for nodes which are roots of an infinite path in the proof tree. We illustrate these notions by a simple example. As usual, sets of rules are expressed by *meta-rules* with side conditions, and analogously sets of corules are expressed by *meta-corules* with side conditions. (Meta-)corules will be written with thicker lines, to be distinguished from (meta-)rules. The following inference system defines the minimum element of a list, where ϵ is the empty list, and $x : u$ the list with head x and tail u.

$$\frac{}{min(x : \epsilon, x)} \qquad \frac{min(u, y)}{min(x{:}u, z)} \; z = \min(x, y)$$

The inductive interpretation gives the correct result only on finite lists, since for infinite lists an infinite proof is clearly needed. However, the coinductive one fails to be a function. For instance, for L the infinite list $2 : 1 : 2 : 1 : 2 : 1 : \ldots$, any judgment $min(L, x)$ with $x \leq 1$ can be derived, as shown below.

$$\frac{\frac{\frac{\cdots}{min(L,1)}}{min(1{:}L,1)}}{min(2{:}1{:}L,1)} \qquad \frac{\frac{\frac{\cdots}{min(L,0)}}{min(1{:}L,0)}}{min(2{:}1{:}L,0)}$$

By adding a corule (in this case a coaxiom), wrong results are "filtered out":

$$\frac{}{min(x{:}\epsilon, x)} \qquad \frac{min(u, y)}{min(x{:}u, z)} \; z = \min(x, y) \qquad \frac{}{min(x{:}u, x)}$$

Indeed, the judgment $min(2{:}1{:}L, 1)$ has the infinite proof tree shown above, and each node has a finite proof tree in the inference system extended by the corule:

$$\frac{\frac{\frac{\cdots}{min(L,1)}}{min(1{:}L,1)}}{min(2{:}1{:}L,1)} \qquad \frac{\frac{}{min(1{:}L,1)}}{min(2{:}1{:}L,1)}$$

The judgment $min(2{:}1{:}L, 0)$, instead, has the infinite proof tree shown above, but has *no finite proof tree* in the inference system extended by the corule. Indeed, since 0 does not belong to the list, the corule can never be applied. On the other hand, the judgment $min(L, 2)$ has a finite proof tree with the corule, but cannot be derived since it has no infinite proof tree. We refer to [7, 16, 17] for other examples.

Note that the inductive and coinductive interpretation of $\mathcal{I}$ are special cases, notably:

- the inductive interpretation of $\mathcal{I}$ is the interpretation of $(\mathcal{I}, \emptyset)$
- the coinductive interpretation of $\mathcal{I}$ is the interpretation of $(\mathcal{I}, \{\frac{\emptyset}{c} \mid c \in \mathcal{U}\})$.

In [7, 16, 17] it is shown that this corresponds to taking a fixed point of $F_{\mathcal{I}}$ which is, in general, neither the least, nor the greatest.

Let $(\mathcal{I}, \mathcal{I}^{co})$ be a generalized inference system. The *bounded coinduction principle* [7, 16, 17], a generalization of the standard coinduction principle, can

be used to prove *completeness* of $(\mathcal{I}, \mathcal{I}^{co})$ w.r.t. a set S (for "specification") of *valid* judgments.

THEOREM 2.1 (Bounded coinduction): If the following two conditions hold:

1. $S \subseteq Ind(\mathcal{I} \cup \mathcal{I}^{co})$, that is, each valid judgment has a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{co}$;
2. $S \subseteq F_{\mathcal{I}}(S)$, that is, each valid judgment is the consequence of a rule in $\mathcal{I}$ with premises in S

then $S \subseteq Gen(\mathcal{I}, \mathcal{I}^{co})$.

The standard coinduction principle can be obtained when $\mathcal{I}^{co} = \{\frac{\emptyset}{c} \mid c \in \mathcal{U}\}$; for this particular case the first condition trivially holds.

2.2 Syntax and semantics of FJ

Featherweight Java (FJ) is a simple calculus modeling the key features of Java-like languages, firstly proposed in [21]. The syntax and semantics in big-step style of FJ are shown in Figure 2.1.

We assume infinite sets of *class names* C, including the special class name `Object`, *field names* f, *method names* m, and *variables* x, including the special variable `this`. We omit cast since this feature does not add significant issues for our aims. We adopt a big-step, rather than a small-step style as in the original FJ definition, since in this way the semantics is directly defined by an inference system, denoted $\mathcal{I}_{\text{FJ}}$ in the following, which will be equipped with corules to support infinite objects. We write $\overline{cd}$ as metavariable for $cd_1 \ldots cd_n$, $n \geq 0$, and analogously for other sequences. We sometimes use the wildcard _ when the corresponding metavariable is not relevant.

A sequence of class declarations $\overline{cd}$ is called a *class table*. Each class has a canonical constructor whose parameters match the fields of the class, the inherited ones first. We assume standard FJ constraints, e.g., no field hiding and no method overloading. The only variables occurring in method bodies are parameters (including `this`). Values are *objects*, that is, constructor invocations where arguments are values in turn.

The judgment $e \Downarrow v$ is implicitly parametrized on a fixed class table. In the rules we use standard FJ auxiliary functions formally defined at the bottom of Figure 2.1. Notably, $\mathit{fields}(C)$ returns the sequence $f_1 \ldots f_n$ of the field names[2] of class C, in declaration order with the inherited first, and $\mathit{mbody}(C, m)$, for method m of class C, the pair of the sequence of parameters and the definition. Substitution $e[\overline{e}/\overline{x}]$, for $\overline{e}$ and $\overline{x}$ of the same length, is defined in the customary manner. Finally, for $\overline{e} = e_1 \ldots e_n$ and $\overline{v} = v_1 \ldots v_n$, $\overline{e} \Downarrow \overline{v}$ is an abbreviation for $e_1 \Downarrow v_1 \ldots e_n \Downarrow v_n$.

Rule (FJ-FIELD) models field access. If the selected field is actually a field of the receiver's class, then the corresponding value is returned as result.

2 We omit types since not relevant here. We discuss about type systems for coFJ later.

$$\begin{array}{llll}
cd & ::= & \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \} & \text{class declaration} \\
fd & ::= & C\,f; & \text{field declaration} \\
md & ::= & C\ m(C_1\,x_1,\ldots,C_n\,x_n)\ \{e\} & \text{method declaration} \\
e \in \mathcal{E} & ::= & x \mid e.f \mid \mathsf{new}\ C(\overline{e}) \mid e.m(\overline{e}) & \text{expression} \\
 & & & \\
v \in \mathcal{V} & ::= & \mathsf{new}\ C(\overline{v}) & \text{(finite) object}
\end{array}$$

$$\text{(FJ-FIELD)}\ \frac{e \Downarrow v}{e.f \Downarrow v_i}\quad \begin{array}{l} v = \mathsf{new}\ C(v_1,\ldots,v_n) \\ \mathit{fields}(C) = f_1 \ldots f_n \\ f = f_i,\ i \in 1..n \end{array} \qquad \text{(FJ-NEW)}\ \frac{\overline{e} \Downarrow \overline{v}}{\mathsf{new}\ C(\overline{e}) \Downarrow \mathsf{new}\ C(\overline{v})}$$

$$\text{(FJ-INVK)}\ \frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e[v_0/\texttt{this}][\overline{v}/\overline{x}] \Downarrow v}{e_0.m(\overline{e}) \Downarrow v}\quad \begin{array}{l} v_0 = \mathsf{new}\ C(_) \\ \mathit{mbody}(C,m) = (\overline{x}, e) \end{array}$$

$\mathit{fields}(\texttt{Object}) = \epsilon$

Given $\texttt{class}\ C\ \texttt{extends}\ C'\ \{C_1\,f_1\,;\ \ldots\ C_n\,f_n\,;\ \overline{md}\}$

$\mathit{fields}(C) = \mathit{fields}(C')\ f_1 \ldots f_n$

$$\mathit{mbody}(C,m) = \begin{cases} (x_1 \ldots x_n, e) & \text{if } C\ m(C_1\,x_1,\ldots,C_n\,x_n)\ \{e\} \in \overline{md} \\ \mathit{mbody}(C',m) & \text{otherwise} \end{cases}$$

FIGURE 2.1 FJ syntax and big-step rules

Rule (FJ-NEW) models object creation: if the argument expressions $\overline{e}$ evaluate to values $\overline{v}$, then the result is an object of class C. Rule (FJ-INVK) models method invocation. The receiver and argument expressions are evaluated first. Then, method look-up is performed, starting from the receiver's class, by the auxiliary function *mbody*. Lastly, the definition e of the method, where `this` is replaced by the receiver, and the parameters by the arguments, is evaluated, and its result is returned.

2.3 Towards infinite objects and codefinitions

We take as running example the following FJ implementation of lists of integers, equipped with some typical methods: `isEmpty` tests the emptiness, `incr` returns the list where all elements have been incremented by one, `allPos` checks whether all elements are positive, `member` checks whether the argument is in the list, and `min` returns the minimal element.

```
class List extends Object {
  bool isEmpty() {true}
  List incr() {new EmptyList()}
  bool allPos() {true}
  bool member(int x) {false}
}
class EmptyList extends List { }
class NonEmptyList extends List {
  int head; List tail;
  bool isEmpty() {false}
  List incr() {new NonEmptyList(this.head+1,this.tail.incr())}
  bool allPos() {if (this.head<=0) false else this.tail.allPos()}
  bool member(int x) {
    if (this.head==x) true
    else this.tail.member(x)
  }
  int min() {
    if (this.tail.isEmpty()) this.head
    else Math.min(this.tail.min(),this.head)
  }
}
```

We used some additional standard constructs, such as conditional and primitive types `bool` and `int` with their operations; to avoid to use abstract methods, `List` provides the default implementation on empty lists, overridden in `NonEmptyList`, except for method `min` which is only defined on non empty lists.

In FJ programs consist in a sequence of class declarations and an expression to be evaluated, which corresponds to the body of the `main` method in full Java. In this way, we can represent finite lists, for instance, the object

```
new NonEmptyList(2, new NonEmptyList(1, new EmptyList()))
```

which we will abbreviate as $[2, 1]$, represents a list of two elements, and it is easy to see that all the above method definitions provide the expected meaning on finite lists.

On the other hand, since the syntactic definition for objects is interpreted, like the others, inductively, in FJ objects are *finite*, hence we cannot represent, e.g., the infinite list of natural numbers $[0, 1, 2, 3, \ldots]$, abbreviated as $[0..]$, or the infinite list $[2, 1, 2, 1, 2, 1, \ldots]$, abbreviated as $[2, 1]^\omega$. To move from finite to infinite objects, it is enough to interpret the syntactic definition for values *coinductively*, so to obtain infinite terms as well. However, to make the extension significant, we should be able to *generate* such infinite objects as results of expressions, and to appropriately *handle* them by methods.

To generate infinite objects, e.g., the infinite lists mentioned above, a natural approach is to consider method definitions as *corecursive*, that is, to take the *coinductive* interpretation of the inference system in Figure 2.1. Consider the following class:

```
class ListFactory extends Object {
  NonEmptyList from(int x) {new NonEmptyList(x, this.from(x+1)}
  NonEmptyList two_one() {new NonEmptyList(2, this.one_two())}
  NonEmptyList one_two() {new NonEmptyList(1, this.two_one())}
}
```

With the standard FJ semantics, given by the inductive interpretation of the inference system in Figure 2.1, the method invocation `new ListFactory().from(0)` (abbreviated $\texttt{from}_0$ in the following) has no result, since there is no finite proof tree for a judgment of shape $\texttt{from}_0 \Downarrow _$. Taking the coinductive interpretation, instead, such call returns as result the infinite list of natural numbers $[0..]$, since there is an infinite proof tree for the judgment $\texttt{from}_0 \Downarrow [0..]$. Analogously, the method invocation `new ListFactory().two_one()` returns $[2, 1]^\omega$. Moreover, the method invocations $[0..]$`.incr()` and $[2, 1]^\omega$`.incr()` correctly return as result the infinite lists $[1..]$ and $[3, 2]^\omega$, respectively.

However, in many cases to consider method definitions as corecursive is not satisfactory, since it leads to non-determinism, as shown for inference systems in Section 2.1. For instance, for the method invocation $[0..]$`.allPos()` both judgments $[0..]$`.allPos()` $\Downarrow$ `true` and $[0..]$`.allPos()` $\Downarrow$ `false` are derivable, and analogously for $[2, 1]^\omega$`.allPos()`. In general, both results can be obtained for any infinite list of all positive numbers. A similar behavior is exhibited by method `member`: given an infinite list L which does not contain x, both judgments L`.member(`x`)` $\Downarrow$ `true` and L`.member(`x`)` $\Downarrow$ `false` are derivable. Finally, for the method invocation $[2, 1]^\omega$`.min()`, any judgment $[2, 1]^\omega$`.min()` $\Downarrow x$ with $x \leq 1$ can be derived.

To solve this problem, we introduced coFJ, an extension of Featherweight Java that allows the programmer to *control* the semantics of corecursive methods by adding a *codefinition*[3], that is, an alternative method body playing a special role. Depending on the codefinition, the purely coinductive interpreta-

3 The term "codefinition" is meant to suggest "alternative definition used to handle corecursion".

tion is refined, by filtering out some judgments. In the example, to achieve the expected meaning, the programmer should provide the following codefinitions.

```
class ListFactory extends Object {
  NonEmptyList from(int x) {
    new NonEmptyList(x, this.from(x+1)} corec {any}
  NonEmptyList one_two() {
    new NonEmptyList(1, this.two_one())} corec {any}
  NonEmptyList two_one() {
    new NonEmptyList(2, this.one_two())} corec {any}
}
class NonEmptyList extends List {
  int head; List tail;
  bool isEmpty() {false}
  List incr() {
    new NonEmptyList(this.head+1,this.tail.incr())} corec {any}
  bool allPos() {
    if (this.head <= 0) false else this.tail.allPos()
  } corec {true}
  bool member(int x) {
    if (this.head == x) true else this.tail.member(x)
  } corec {false}
  int min() {
    if (this.tail.isEmpty()) this.head
    else Math.min(this.tail.min(),this.head)
  } corec {this.head}
}
```

For the three methods of `ListFactory` and for the method `incr` the codefinition is `any`. This corresponds to keeping the coinductive interpretation as it is, as appropriate in these cases since it provides only the expected result. In the other three methods, instead, the effect of the codefinition is to filter the results obtained by the coinductive interpretation. The way this is achieved is explained in the following chapters. Finally, for method `isEmpty` no codefinition is added, since the inductive behaviour works on infinite lists as well.

3

coFJ and its abstract semantics

In this chapter we formally introduce coFJ by presenting its abstract semantics. We call it *abstract* because it allows reasoning on coFJ at a very high level of abstraction, as in a denotational semantics. In particular, this semantics handles *possibly infinite objects*, and judgments have *possibly infinite proof trees*. Moreover, it is *non-deterministic*, that is, expressions can be evaluated to more than one value. Throughout the chapter, we will also illustrate how the previous examples get the expected semantics, and show that, despite its non-determinism, coFJ is a conservative extension of FJ.

3.1 Formal definition of coFJ

The coFJ syntax and abstract semantics are given in Figure 3.1.

As the reader can note, the only difference with the FJ syntax in Figure 2.1 is that method declarations include, besides a definition e, an optional *codefinition* e', as denoted by the square brackets in the production. Furthermore, besides `this`, there is another special variable `any`, which can only occur in codefinitions. The codefinition will be used to provide an abstract semantics through an inference system with corules, where the role of `any` is to be a placeholder for an arbitrary value. For simplicity, we require the codefinition e' to be statically restricted to avoid recursive (even indirect) calls to the same method (we omit the standard formalization). Note that FJ is a (proper) subset of coFJ: indeed, an FJ class table is a coFJ class table with no codefinitions.

The syntactic definition for values is the same as before, but is now interpreted *coinductively*, as indicated by the symbol $::=_{\text{co}}$. In this way, infinite objects are supported. By replacing method parameters by arguments, we obtain *runtime expressions* admitting infinite objects as subterms. The sets $\mathcal{V}$ and $\mathcal{E}$ of FJ objects and expressions are subsets of $\mathcal{V}^{\text{a}}$ and $\mathcal{E}^{\text{a}}$, respectively. The judgment $e \Downarrow v$, with $e \in \mathcal{E}^{\text{a}}$ and $v \in \mathcal{V}^{\text{a}}$, is defined by an inference system with corules $(\mathcal{I}_{\text{FJ}}, \mathcal{I}^{co}_{\text{FJ}})$ where the rules $\mathcal{I}_{\text{FJ}}$ are those[1] of FJ, as in Figure 2.1, and the corules $\mathcal{I}^{co}_{\text{FJ}}$ are instances of two meta-corules.

Corule (abs-co-val) is needed to obtain a value for infinite objects, as shown below. Corule (abs-co-invk) is analogous to the standard rule for

1 To be precise, meta-rules are the same, with meta-variables e and v ranging on $\mathcal{E}^{\text{a}}$, and $\mathcal{V}^{\text{a}}$, respectively. However, we could have taken this larger universe in FJ as well without affecting the defined relation.

cd	$::=$	`class` C `extends` C' $\{\,\overline{fd}\ \overline{md}\,\}$	class declaration
fd	$::=$	$C\,f$;	field declaration
md	$::=$	$C\ m(C_1\,x_1,\ldots,C_n\,x_n)\ \{e\}\ [\texttt{corec}\ \{e'\}]$	method declaration with codefinition
$e \in \mathcal{E}$	$::=$	$x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e})$	expression
$v \in \mathcal{V}^a$	$::=_{co}$	$\texttt{new}\ C(\overline{v})$	possibly infinite object
$e \in \mathcal{E}^a$	$::=$	$x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e}) \mid v$	runtime expression

$$\text{(ABS-FIELD)}\ \frac{e \Downarrow v}{e.f \Downarrow v_i}\quad \begin{array}{l} v = \texttt{new}\ C(v_1,\ldots,v_n) \\ \mathit{fields}(C) = f_1 \ldots f_n \\ f = f_i,\ i \in 1..n \end{array} \qquad \text{(ABS-NEW)}\ \frac{\overline{e} \Downarrow \overline{v}}{\texttt{new}\ C(\overline{e}) \Downarrow \texttt{new}\ C(\overline{v})}$$

$$\text{(ABS-INVK)}\ \frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e[v_0/\texttt{this}][\overline{v}/\overline{x}] \Downarrow v}{e_0.m(\overline{e}) \Downarrow v}\quad \begin{array}{l} v_0 = \texttt{new}\ C(_) \\ \mathit{mbody}(C,m) = (\overline{x}, e) \end{array} \qquad \text{(ABS-CO-VAL)}\ \frac{}{v \Downarrow v}$$

$$\text{(ABS-CO-INVK)}\ \frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e'[v_0/\texttt{this}][\overline{v}/\overline{x}][v/\texttt{any}] \Downarrow v_{co}}{e_0.m(\overline{e}) \Downarrow v_{co}}\quad \begin{array}{l} v_0 = \texttt{new}\ C(_) \\ \mathit{co\text{-}mbody}(C,m) = (\overline{x}, e') \end{array}$$

FIGURE 3.1 coFJ syntax and abstract semantics

method invocation, but uses the codefinition, and the variable `any` can be non-deterministically substituted with an arbitrary value. The auxiliary function *co-mbody* is defined analogously to *mbody*, but it returns the codefinition. Note that, even when $\mathit{mbody}(C,m)$ is defined, $\mathit{co\text{-}mbody}(C,m)$ can be undefined since no codefinition has been specified. This can be done to force a purely inductive behaviour for the method.

3.2 Examples

As an example, we illustrate in Figure 3.2 the role of the two corules for the call `new ListFactory().from(0)`. For brevity, we write abbreviated class names. Furthermore, $\texttt{from}_n$ stands for the call `new ListFactory().from(n)` and $[n..]$ for the infinite object `new NonEmptyList(n,new NonEmptyList(n+1,...)))`.

In the top part of Figure 3.2, we show the infinite proof tree T_n which can be constructed, for any natural number n, for the judgment $\texttt{from}_n \Downarrow [n..]$ without the use of corules. We use standard rules (N-VAL) and (+) to deal with integer constants and addition.

To derive the judgment in the inference system with corules, each node in this infinite tree should have a finite proof tree with the corules. Notably, this should hold for nodes of shape $\texttt{from}_n \Downarrow [n..]$, and indeed the finite proof tree for such nodes is shown in the bottom part of the figure. Note that, in this example, the result for the call $\texttt{from}_n$, that is, the infinite list of natural numbers, is uniquely determined by the rules, hence the role of the corules is just to "validate" this result. To this end, the codefinition of the method `from` is the special variable `any`, which, when evaluating the codefinition, can be replaced

$$T_n = \dfrac{\dfrac{}{\texttt{new LF()} \Downarrow \texttt{new LF()}}\ (\text{ABS-NEW}) \quad \dfrac{}{n \Downarrow n}\ (\text{N-VAL}) \quad \dfrac{\dfrac{}{n \Downarrow n}\ (\text{N-VAL}) \quad T_{n+1}}{\texttt{new NEL(}n\texttt{,new LF().from(}n\texttt{+1))} \Downarrow [n..]}\ (\text{ABS-NEW})}{\texttt{from}_n \Downarrow [n..]}\ (\text{ABS-INVK})$$

$$T_{n+1} = \dfrac{\dfrac{}{\texttt{new LF()} \Downarrow \texttt{new LF()}}\ (\text{ABS-NEW}) \quad \dfrac{\dots}{n+1 \Downarrow n+1}\ (+) \quad \dfrac{\dfrac{}{n+1 \Downarrow n+1}\ (\text{N-VAL}) \quad T_{n+2}}{\texttt{new NEL(}n+1\texttt{,new LF().from(}n+1+1\texttt{))} \Downarrow [n+1..]}\ (\text{ABS-NEW})}{\texttt{new LF().from}_{n+1} \Downarrow [n+1..]}\ (\text{ABS-INVK})$$

$$(\text{ABS-CO-INVK})\ \dfrac{(\text{ABS-NEW})\ \dfrac{}{\texttt{new LF()} \Downarrow \texttt{new LF()}} \quad (\text{N-VAL})\ \dfrac{}{n \Downarrow n} \quad (\text{ABS-CO-VAL})\ \dfrac{}{[n..] \equiv \texttt{any[new LF()/this][}[n..]\texttt{/any]} \Downarrow [n..]}}{\texttt{from}_n \Downarrow [n..]}$$

FIGURE 3.2 Infinite (top) and finite (bottom) proof trees for $\texttt{from}_n \Downarrow [n..]$

by any value, hence, in particular, by the correct result $[n..]$. Corule (ABS-CO-VAL) is needed to obtain a finite proof tree for the infinite objects of shape $[n..]$. Similar infinite and finite proof trees can be constructed for the judgments `new ListFactory().two_one()` $\Downarrow [2, 1]^\omega$, $[0..]$`.incr()` $\Downarrow [1..]$ and $[2, 1]^\omega$`.incr()` $\Downarrow [3, 2]^\omega$, where $[2, 1]^\omega$ and $[3, 2]^\omega$ are abbreviations for the values v_1 s.t. $v_1 =$ `new NonEmptyList(2,new NonEmptyList(1,`v_1`))` and v_2 s.t. $v_2 =$ `new NonEmptyList(3,new NonEmptyList(2,`v_2`))`, respectively.

For the method call $[0..]$`.allPos()`, instead, both judgments $[0..]$`.allPos()` $\Downarrow$ `true` and $[0..]$`.allPos()` $\Downarrow$ `false` have an infinite proof tree. However, no finite proof tree using the codefinition can be constructed for the latter, whereas this is trivially possible for the former. Analogously, given an infinite list L which does not contain x, only the judgment L`.member(`x`)` $\Downarrow$ `false` has a finite proof tree using the codefinition.

Finally, for the method invocation $[2, 1]^\omega$`.min()`, for any $v \leq 1$ there is an infinite proof tree built without corules for the judgment $[2, 1]^\omega$`.min()` $\Downarrow v$ as shown in Figure 3.3.

However, only the judgment $[2, 1]^\omega$`.min()` $\Downarrow 1$ has a finite proof tree using the codefinition (Figure 3.4). In both figures ellipses are used to omit the less interesting parts of the proof trees; we use the standard rule (IF-F) for conditional, and the predefined function `Math.min` on integers.

3.3 Non-determinism and conservativity

The coFJ abstract semantics is inherently non-deterministic. Indeed, depending on the codefinition, the non-determinism of the coinductive interpretation may be kept. For instance, consider the following method declaration:

```
class C {
  C m() { this.m() } corec { any }
}
```

$$\dfrac{T_0 \qquad T_1}{[2,1]^\omega.\texttt{min()} \Downarrow v}\,(\textsc{abs-invk}) \qquad T_0 = \dfrac{\dfrac{}{2 \Downarrow 2}\,(\textsc{n-val}) \quad \dfrac{\dfrac{}{1 \Downarrow 1}\,(\textsc{n-val}) \quad T_0}{[1,2]^\omega \Downarrow [1,2]^\omega}\,(\textsc{abs-new})}{[2,1]^\omega \Downarrow [2,1]^\omega}\,(\textsc{abs-new})$$

$$T_1 = \dfrac{\dfrac{\vdots}{[2,1]^\omega.\texttt{tail.isEmpty()} \Downarrow \texttt{false}} \quad \dfrac{\dfrac{\overset{T_2}{\vdots}}{[2,1]^\omega.\texttt{tail.min()} \Downarrow v} \quad \dfrac{\vdots}{[2,1]^\omega.\texttt{head} \Downarrow 2}}{\texttt{Math.min(}[2,1]^\omega.\texttt{tail.min()}, [2,1]^\omega.\texttt{head)} \Downarrow v}}{\textbf{if } [2,1]^\omega.\texttt{tail.isEmpty()} \textbf{ then } [2,1]^\omega.\texttt{head} \textbf{ else } \texttt{Math.min(}[2,1]^\omega.\texttt{tail.min()}, [2,1]^\omega.\texttt{head)} \Downarrow v}\,(\textsc{if-f})$$

$$T_2 = \dfrac{\dfrac{\vdots}{[1,2]^\omega.\texttt{tail.isEmpty()} \Downarrow \texttt{false}} \quad \dfrac{\dfrac{\overset{T_1}{\vdots}}{[1,2]^\omega.\texttt{tail.min()} \Downarrow v} \quad \dfrac{\vdots}{[1,2]^\omega.\texttt{head} \Downarrow 1}}{\texttt{Math.min(}[1,2]^\omega.\texttt{tail.min()}, [1,2]^\omega.\texttt{head)} \Downarrow v}}{\textbf{if } [1,2]^\omega.\texttt{tail.isEmpty()} \textbf{ then } [1,2]^\omega.\texttt{head} \textbf{ else } \texttt{Math.min(}[1,2]^\omega.\texttt{tail.min()}, [1,2]^\omega.\texttt{head)} \Downarrow v}\,(\textsc{if-f})$$

FIGURE 3.3 Infinite proof tree for $[2,1]^\omega.\texttt{min()} \Downarrow v$ with $v \leq 1$ (main tree at the top left corner)

$$\dfrac{T_0 \quad \dfrac{\dfrac{\vdots}{[2,1]^\omega.\texttt{tail.isEmpty()} \Downarrow \texttt{false}} \quad \dfrac{(\textsc{abs-co-invk})\,\dfrac{\cdots \quad \dfrac{(\textsc{abs-co-val})\,\dfrac{}{[1,2]^\omega \Downarrow [1,2]^\omega}}{[1,2]^\omega.\texttt{head} \Downarrow 1}}{[2,1]^\omega.\texttt{tail.min()} \Downarrow 1} \quad \dfrac{\vdots}{[2,1]^\omega.\texttt{head} \Downarrow 2}}{\texttt{Math.min(}[2,1]^\omega.\texttt{tail.min()}, [2,1]^\omega.\texttt{head)} \Downarrow 1}}{\textbf{if } [2,1]^\omega.\texttt{tail.isEmpty()} \textbf{ then } [2,1]^\omega.\texttt{head} \textbf{ else } \texttt{Math.min(}[2,1]^\omega.\texttt{tail.min()}, [2,1]^\omega.\texttt{head)} \Downarrow 1}\,(\textsc{if-f})}{[2,1]^\omega.\texttt{min()} \Downarrow 1}\,(\textsc{abs-invk})$$

FIGURE 3.4 Finite proof tree with codefinition for $[2,1]^\omega.\texttt{min()} \Downarrow 1$ (T_0 as in Figure 3.3)

Method `m()` recursively calls itself. In the abstract semantics, the judgment `new C().m()` $\Downarrow v$ can be derived for any value v. In the operational semantics defined in Chapter 4, such method call evaluates to $(x, x : x)$, that is, the representation of *undetermined*.

However, determinism of FJ evaluation is preserved. Indeed, coFJ abstract semantics is a *conservative* extension of FJ semantics, as formally stated below.

THEOREM 3.1 (Conservativity): If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, then $(\mathcal{I}_{\text{FJ}}, \mathcal{I}_{\text{FJ}}^{co}) \vdash e \Downarrow v'$ iff $v = v'$.

Proof: Both directions can be easily proved by induction on the definition of $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$. For the left-to-right direction, the fact that each syntactic category has a unique applicable meta-rule is crucial. □

This theorem states that, whichever the codefinitions chosen, coFJ does not change the semantics of expressions evaluating to some value in FJ. That is, coFJ abstract semantics allows derivation of new values only for expressions whose semantics is undefined in standard FJ, as in the examples shown above. Note also that, if no codefinition is specified, then the coFJ abstract semantics *coincides* with the FJ one, because corule (ABS-CO-INVK) cannot be applied, hence no infinite proof trees can be built for the evaluation of FJ expressions.

4

coFJ operational semantics

In this chapter we introduce coFJ operational semantics. Differently from the abstract semantics, here methods manipulate *finite representations* (by equations) of possibly infinite objects, and their interpretation is defined by an *inductive* inference system based on *cycle detection*. That is, the operational semantics keeps track of pending method calls. Thus, when an already processed call is found, this does not lead to non-termination as in standard semantics of recursion, since the cycle is detected, and the corresponding codefinition is evaluated. This mechanism has been fruitfully employed in other programming paradigms [30, 23, 6], and this operational semantics, being inductive, is algorithmic and, thus, executable.

4.1 Formal definition

In contrast to the abstract semantics of the previous section, the aim is to define a semantics which leads to an interpreter for the calculus. To obtain this, there are two issues to be considered:

1. infinite (regular) objects should be represented in a finite way;
2. infinite (regular) proof trees should be replaced by finite proof trees.

In the following we explain how these issues are handled in the coFJ operational semantics.

To obtain (1), we use an approach based on *capsules* [22], which are essentially expressions supporting cyclic references. In our context, capsules are pairs (e, σ) where e is an FJ expression and σ is an *environment*, that is, a finite mapping from variables into FJ expressions. Moreover, the following *capsule property* is satisfied: writing $FV(e)$ for the set of free variables in e, $FV(e) \subseteq dom(\sigma)$ and, for all $x \in dom(\sigma)$, $FV(\sigma(x)) \subseteq dom(\sigma)$. An FJ source expression e is represented by the capsule $(e, \emptyset)$, where $\emptyset$ denotes the empty environment. In particular, values are pairs (v, σ) where v is an *open* FJ object, that is, an object possibly containing variables. In this way, cyclic objects can be obtained: for instance, $(x, x : \texttt{new NEL}(2, \texttt{new NEL}(1, x)))$ represents the infinite regular list $[2, 1]^\omega$ considered before, with NEL as an abbreviation of $NonEmptyList$.

To obtain (2), methods are *regularly corecursive*. This means that execution keeps track of the pending method calls, so that, when a call is encountered

the second time, this is detected[1], avoiding non-termination as it would happen with ordinary recursion. Regular corecursion in coFJ is *flexible*, since the behaviour of the method when a cycle is detected is specified by the codefinition.

Consider, for instance, the method call `new ListFactory().two_one();` thanks to regular corecursion, the result is the cyclic object $(x, x : \texttt{new NEL}(2, \texttt{new NEL}(1, x)))$. Indeed, the operational semantics associates a fresh variable, say, x, to the initial call, so that, when the same call is encountered the second time, the association $x : x$ is added in the environment, and the codefinition is evaluated where any is replaced by x. Hence, $(x, x : x)$ is returned as result, so that the result of the original call is $(x, x : \texttt{new NEL}(2, \texttt{new NEL}(1, x)))$. The call `new ListFactory().from(0)`, instead, does not terminate in the operational semantics, since no call is encountered more than once (the resulting infinite object is non-regular).

Consider now the call $[2, 1]^{\omega}$.`allPos`(). In this case, when the call is encountered the second time, after an intermediate call $[1, 2]^{\omega}$.`allPos`(), the result of the evaluation of the codefinition is `true`, so that the result of the original call is `true` as well.[2] If the codefinition were any, then the result would be $(x, x : x)$, that is, undetermined. Note that, if the list is finite, then no regular corecursion is involved, since the same call cannot occur more than once; the same holds if the list is cyclic, but contains a non-positive element, hence the method invocation returns `false`. The only case requiring regular corecursion is when the method is invoked on a cyclic list with all positive elements, as $[2, 1]^{\omega}$.

In the case of $[2, 1]^{\omega}$.`min`(), when the call is encountered the second time the result of the evaluation of the codefinition is 2, so that the result of the intermediate call $[1, 2]^{\omega}$.`min`() is 1, and this is also the result of the original call.

To formally express the approach described above, the judgment of the operational semantics has shape $e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'$ where: (e, σ) is the capsule to be evaluated; τ is a *call trace*, used to keep track of already encountered calls, that is, an injective map from *calls* $\mathrm{v}_0.m(\overline{\mathrm{v}})$ to (possibly tagged) variables, and (v, σ') is the capsule result. Variables in the codomain of the call trace have a tag ck during the checking step for the corresponding call, as detailed below. The pairs (e, σ) and (v, σ') are assumed to satisfy the capsule property.

We report coFJ syntax in Figure 4.1 for the reader's convenience, while the semantic rules are given in Figure 4.2. We denote by $\sigma\{x : \mathrm{v}\}$ the environment which gives v on x, and is equal to σ elsewhere, and analogously for other maps. Furthermore, we use the following notations, formally defined in Figure 4.3.

- *unfold*(v, σ) is the *unfolding* of v in σ, that is, the corresponding object, if any.
- $\sigma_1 \sqcup \sigma_2$ is the *union of environments*, defined if they agree on the common domain.

1 The semantics detects an already processed call by relying on capsule equivalence (Figure 4.3).
2 To be rigorous, a capsule of shape (`true`, _).

cd	$::=$	`class` C `extends` C' $\{\ \overline{fd}\ \overline{md}\ \}$	class declaration
fd	$::=$	$C\,f$;	field declaration
md	$::=$	$C\ m(C_1\,x_1,\ldots,C_n\,x_n)\ \{e\}$ [`corec` $\{e'\}$]	method declaration with codefinition
$e \in \mathcal{E}$	$::=$	$x \mid e.f \mid$ new $C(\overline{e}) \mid e.m(\overline{e})$	expression
$v \in \mathcal{V}^{\mathrm{a}}$	$::=_{\mathrm{co}}$	new $C(\overline{v})$	possibly infinite object
$e \in \mathcal{E}^{\mathrm{a}}$	$::=$	$x \mid e.f \mid$ new $C(\overline{e}) \mid e.m(\overline{e}) \mid v$	runtime expression

FIGURE 4.1 coFJ syntax

- $(\mathrm{v},\sigma)\approx(\mathrm{v}',\sigma')$ is the *equivalence of capsules.* As will be formalized in the first part of Chapter 7, equivalent capsules denote the same sets of abstract objects. This equivalence is extended by congruence to expressions, in particular to calls $\mathrm{v}_0.m(\overline{\mathrm{v}})$.
- $\tau_{\approx\sigma}$ is obtained by extending τ *up to equivalence* in σ. That is, detection of already encountered calls is performed up-to equivalence in the current environment.

When reading the rules, recall that they are expected to preserve the invariant that the result of evaluation (v,σ) satisfies the capsule property, that is, σ should be defined on all the variables possibly occurring in v.

Rule (VAL) is needed for objects which are not FJ objects. Rule (FIELD) is similar to that of FJ except that the capsule (v,σ') must be unfolded to retrieve the corresponding object. Furthermore, the resulting environment is that obtained by evaluating the receiver. Rule (NEW) is analogous to that of FJ. The resulting environment is the union of those obtained by evaluating the arguments.

There are four rules for method invocation. In all of them, as in the FJ rule, the receiver and argument expressions are evaluated first to obtain the call $c = \mathrm{v}_0.m(\overline{\mathrm{v}})$. The environment $\widehat{\sigma}$ is the union of those obtained by these evaluations. Then, the behavior is different depending on whether such call (meaning a call equivalent to c in $\widehat{\sigma}$) has already been processed or not.

Rules (INVK-OK) and (INVK-CHECK) handle[3] a call c which is encountered the first time, as expressed by the side condition $c \notin dom(\tau_{\approx\widehat{\sigma}})$. In both, the definition e, where the receiver replaces `this` and the arguments replace the parameters, is evaluated. Such evaluation is performed in the call trace τ updated to associate the call c with an unused variable x (in these two rules "x fresh" means that x does not occur in the derivations of $e_i,\sigma,\tau \Downarrow \mathrm{v}_i,\sigma'_i$, for all $i \in 0..n$), and produces the capsule (v,σ'). Then there are two cases, depending on whether $x \in dom(\sigma')$ holds.

If $x \notin dom(\sigma')$, then the evaluation of the definition for c has been performed without evaluating the codefinition ((INVK-OK)). That is, the same call has not been processed, hence the result has been obtained by standard recursion, and no additional check is needed.

3 The two rules could be merged together, but we prefer to make explicit the difference for the sake of clarity.

$\mathrm{v} \in \mathcal{V}^{\mathrm{op}}$	$::=$	$\mathsf{new}\ C(\overline{\mathrm{v}}) \mid x$	open object
σ	$::=$	$x_1 : \mathrm{v}_1 \ldots\ x_n : \mathrm{v}_n \quad (n \geq 0)$	environment
c	$::=$	$\mathrm{v}.m(\overline{\mathrm{v}})$	call
t	$::=$	$[\mathsf{ck}]$	optional checking tag
τ	$::=$	$c_1 : x_1^{t_1}, \ldots, c_n : x_n^{t_n} \quad (n \geq 0)$	call trace

$$\text{(val)}\ \frac{}{\mathrm{v}, \sigma, \tau \Downarrow \mathrm{v}, \sigma} \qquad \text{(field)}\ \frac{e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'}{e.f, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'} \quad \begin{array}{l} \mathit{unfold}(\mathrm{v}, \sigma') = \mathsf{new}\ C(\mathrm{v}_1, \ldots, \mathrm{v}_n) \\ \mathit{fields}(C) = f_1 ... f_n \\ f = f_i, i \in 1..n \end{array}$$

$$\text{(new)}\ \frac{e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 1..n}{\mathsf{new}\ C(e_1, \ldots, e_n), \sigma, \tau \Downarrow \mathsf{new}\ C(\mathrm{v}_1, \ldots, \mathrm{v}_n), \bigsqcup_{i \in 1..n} \sigma'_i}$$

In all the following rules: $\begin{array}{l} \overline{e} = e_1, \ldots, e_n \\ \overline{\mathrm{v}} = \mathrm{v}_1 \ldots \mathrm{v}_n \\ c = \mathrm{v}_0.m(\overline{\mathrm{v}}) \\ \widehat{\sigma} = \bigsqcup_{i \in 0..n} \sigma'_i \\ \mathit{unfold}(\mathrm{v}_0, \sigma'_0) = \mathsf{new}\ C(_) \end{array}$

$$\text{(invk-ok)}\ \frac{\begin{array}{c} e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n \\ e[\mathrm{v}_0/\mathtt{this}][\overline{\mathrm{v}}/\overline{x}], \widehat{\sigma}, \tau\{c : x\} \Downarrow \mathrm{v}, \sigma' \end{array}}{e_0.m(\overline{e}), \sigma, \tau \Downarrow \mathrm{v}, \sigma'} \quad \begin{array}{l} c \notin \mathit{dom}(\tau_{\approx\widehat{\sigma}}) \\ x \text{ fresh} \\ \mathit{mbody}(C, m) = (\overline{x}, e) \\ x \notin \mathit{dom}(\sigma') \end{array}$$

$$\text{(invk-check)}\ \frac{\begin{array}{c} e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n \\ e[\mathrm{v}_0/\mathtt{this}][\overline{\mathrm{v}}/\overline{x}], \widehat{\sigma}, \tau\{c : x\} \Downarrow \mathrm{v}, \sigma' \\ e[\mathrm{v}_0/\mathtt{this}][\overline{\mathrm{v}}/\overline{x}], \widehat{\sigma} \sqcup \sigma'\{x : \mathrm{v}\}, \tau\{c : x^{\mathsf{ck}}\} \Downarrow \mathrm{v}', \sigma'' \end{array}}{e_0.m(\overline{e}), \sigma, \tau \Downarrow x, \sigma'\{x : \mathrm{v}\}} \quad \begin{array}{l} c \notin \mathit{dom}(\tau_{\approx\widehat{\sigma}}) \\ x \text{ fresh} \\ \mathit{mbody}(C, m) = (\overline{x}, e) \\ x \in \mathit{dom}(\sigma') \\ (x, \sigma'\{x : \mathrm{v}\}) \approx (\mathrm{v}', \sigma'') \end{array}$$

$$\text{(corec)}\ \frac{\begin{array}{c} e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n \\ e'[\mathrm{v}_0/\mathtt{this}][\overline{\mathrm{v}}/\overline{x}][x/\mathsf{any}], \widehat{\sigma}\{x : x\}, \tau \Downarrow \mathrm{v}, \sigma' \end{array}}{e_0.m(\overline{e}), \sigma, \tau \Downarrow \mathrm{v}, \sigma'\{x : x\}} \quad \begin{array}{l} \tau_{\approx\widehat{\sigma}}(c) = x \\ \mathit{co\text{-}mbody}(C, m) = (\overline{x}, e') \end{array}$$

$$\text{(look-up)}\ \frac{e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n}{e_0.m(\overline{e}), \sigma, \tau \Downarrow x, \widehat{\sigma}} \quad \tau_{\approx\widehat{\sigma}}(c) = x^{\mathsf{ck}}$$

Figure 4.2 coFJ operational semantics

If $x \in dom(\sigma')$, instead, then the evaluation of the definition for c required to evaluate the codefinition ((INVK-CHECK)). In this case, an additional check is required (third premise). That is, $e[\mathrm{v}_0/\texttt{this}][\overline{\mathrm{v}}/\overline{x}]$ is evaluated once more under the assumption that v is the result of the call. Formally, evaluation takes place in an environment updated to associate x with v, and the variable x corresponding to the call is tagged with ck. The capsule result obtained in this way must be (equivalent to) that obtained by the first evaluation of the body of the method. In Chapter 6 we discuss in detail the role of this additional check, showing an example where it is necessary. If the check succeeds, then the final result is the variable x in the environment updated to associate x with v. Otherwise, rule (INVK-CHECK) cannot be applied since the last premise does not hold. For simplicity, we assume the result of c to be undefined in this case; an additional rule could be added raising a runtime error in case the result is different from the expected one, as should be done in an implementation.

The remaining rules handle an already encountered call c, that is, $\tau_{\approx\widehat{\sigma}}(c)$ is defined. The behaviour is different depending on whether the corresponding variable x is tagged or not.

If x is not tagged, then rule (COREC) evaluates the codefinition where the receiver object replaces `this`, the arguments replace the parameters, and, furthermore, the variable x found in the call trace replaces `any`. In addition, $\widehat{\sigma}$ is updated to associate x with x. In this way, the semantics keeps track of the application of rule (COREC). Note that, in the second premise of the rule, the environment $\widehat{\sigma}$ in the input capsule is the same as in the output capsule. This is correct since we assume that the codefinition e' is statically restricted to avoid recursive (even indirect) calls to the same method.

If x is tagged, instead, then we are in a checking step for the corresponding call. In this case, rule (LOOK-UP) simply returns the associated variable for a call; by definition of the operational semantics, in this case such a variable is always defined in the environment.

In this rule we can notice the importance of the tag on τ. Indeed, by keeping this distinction between calls encountered for the first time and calls to be checked, we can avoid possible applications of (LOOK-UP) instead of (COREC).

Figure 4.3 contains the formal definitions of the notations used in the rules.

Note that *unfold*, being inductively defined, can be undefined, denoted $\uparrow$, in presence of unguarded cycles among variables. Capsule equivalence, instead, is defined coinductively, so that, e.g., $(x, x : \texttt{new}\ C(x))$ is equivalent to $(x, x : \texttt{new}\ C(\texttt{new}\ C(x)))$. Capsule equivalence implicitly subsumes α-equivalence of variables whose unfolding is defined, e.g., $(x, x : \texttt{new}\ C(x))$ is equivalent to $(y, y : \texttt{new}\ C(y))$. Instead, α-equivalence of undetermined variables is given by an explicit renaming, which should preserve disjointness of cycles. For instance, $(\texttt{new}\ C(x, y), (x : y, y : x))$ is equivalent to $(\texttt{new}\ C(x, x), x : x)$, but is *not* equivalent to $(\texttt{new}\ C(x, y), (x : x, y : y))$. Indeed, in the latter case x and y can be instantiated independently. We will prove in Chapter 7 (Theorem 7.1) that the relation $\approx_R$, for some σ_1, σ_2-renaming R, is the operational counterpart of the fact that two capsules denote the same set of abstract val-

$$unfold(\text{v}, \sigma) = \begin{cases} \textsf{new } C(\overline{\text{v}}) & \text{if v} = \textsf{new } C(\overline{\text{v}}) \\ unfold(\sigma(\text{v}), \sigma) & \text{if v} = x \end{cases}$$
$$undet(\sigma) = \{x \in dom(\sigma) \mid unfold(x, \sigma) \uparrow\}$$

For σ_1 and σ_2 such that $\sigma_1(x) = \sigma_2(x)$ for all $x \in dom(\sigma_1) \cap dom(\sigma_2)$

$$(\sigma_1 \sqcup \sigma_2)(x) = \begin{cases} \sigma_1(x) & x \in dom(\sigma_1) \\ \sigma_2(x) & x \in dom(\sigma_2) \end{cases}$$

Set $\overset{\sigma}{\leftrightarrow}$ the least equivalence relation on $undet(\sigma)$ such that $x \overset{\sigma}{\leftrightarrow} y$ if $\sigma(x) = y$, $[x]$ the equivalence class of x, and $undet_{\leftrightarrow}(\sigma)$ the quotient.

A relation $R \subseteq undet(\sigma_1) \times undet(\sigma_2)$ is a σ_1, σ_2*-renaming* if it induces a (partial) bijection from $undet_{\leftrightarrow}(\sigma_1)$, still denoted R, to $undet_{\leftrightarrow}(\sigma_2)$. Given R a σ_1, σ_2-renaming, the relation $(x, \sigma_1) \approx_R (x', \sigma_2)$ is coinductively defined by:

$$\frac{}{(x, \sigma) \approx_R (x', \sigma')}\; xRx' \qquad \frac{(\text{v}_i, \sigma) \approx_R (\text{v}'_i, \sigma') \quad \forall i \in 1..n}{(\text{v}, \sigma) \approx_R (\text{v}', \sigma')}\; \begin{array}{l} unfold(\text{v}, \sigma) = \textsf{new } C(\text{v}_1, .., \text{v}_n) \\ unfold(\text{v}', \sigma') = \textsf{new } C(\text{v}'_1, .., \text{v}'_n) \end{array}$$

A σ_1, σ_2-renaming R is *strict* if, for $x, y \in undet(\sigma_1) \cap undet(\sigma_2)$, $[x]R[y]$ iff $x \overset{\sigma_1}{\leftrightarrow} y$ and $x \overset{\sigma_2}{\leftrightarrow} y$.

We write $(\text{v}, \sigma) \approx (\text{v}', \sigma')$ if $(\text{v}, \sigma) \approx_R (\text{v}', \sigma')$ for some strict R.

$$\tau_{\approx\sigma}(c') = \tau(c) \text{ for each } c' \text{ such that } (c', \sigma) \approx (c, \sigma)$$

Figure 4.3 coFJ auxiliary definitions

ues. The stronger strictness condition prevents erroneous identification of objects during evaluation, e.g., $(\mathsf{new}\ C(x, y), (x : x, y : y))$ is not equivalent to $(\mathsf{new}\ C(y, x), (y : y, x : x))$.

4.2 Determinism and conservativity

In contrast to coFJ abstract semantics, but like FJ, coFJ operational semantics is deterministic.

THEOREM 4.1 (Determinism): If $e, \sigma, \tau_1 \Downarrow \mathrm{v}_1, \sigma_1$ and $e, \sigma, \tau_2 \Downarrow \mathrm{v}_2, \sigma_2$ hold and $dom(\tau_1) = dom(\tau_2)$, then (v_1, σ_1) and (v_2, σ_2) are equal up-to α-equivalence.

Proof: The proof is by induction on the derivation for $e, \sigma, \tau_1 \Downarrow \mathrm{v}_1, \sigma_1$. The key point is that, once fixed e, σ and $dom(\tau_1)$, there is a unique applicable rule, hence both $e, \sigma, \tau_1 \Downarrow \mathrm{v}_1, \sigma_1$ and $e, \sigma, \tau_2 \Downarrow \mathrm{v}_2, \sigma_2$ are derived by the same rule. □

As the abstract semantics, the operational semantics is a conservative extension of the standard FJ semantics. This result follows from soundness with respect to the abstract semantics in next section, however the direct proof below provides some useful insight.

THEOREM 4.2 (Conservativity): If $\mathcal{I}_{\mathrm{FJ}} \vdash e \Downarrow v$, then $e, \emptyset, \emptyset \Downarrow \mathrm{v}, \sigma$ holds iff $\mathrm{v} = v$ and $\sigma = \emptyset$.

For the proof, we need some auxiliary lemmas and definitions. First, we note that FJ has the *strong determinism* property: each expression has at most one finite proof tree in $\mathcal{I}_{\mathrm{FJ}}$.

LEMMA 4.3 (FJ strong determinism): If $\mathcal{I}_{\mathrm{FJ}} \vdash e \Downarrow v_1$ by a proof tree t_1 and $\mathcal{I}_{\mathrm{FJ}} \vdash e \Downarrow v_2$ by a proof tree t_2, then $t_1 = t_2$ and $v_1 = v_2$.

Proof: By induction on the definition of $e \Downarrow v_1$. The key point is that each judgement is the consequence of exactly one rule. □

By relying on strong determinism, it is easy to see that in FJ a proof tree for an expression cannot contain another node labelled by the same expression. In other words, if the evaluation of e requires to evaluate e again, then the FJ semantics is undefined on e, as expected.

LEMMA 4.4 : A proof tree in $\mathcal{I}_{\mathrm{FJ}}$ for $e \Downarrow v$ cannot contain any other node $e \Downarrow v'$, for any v'.

Proof: By Lemma 4.3, there is a unique proof tree t for the expression e. Hence, a node $e \Downarrow v'$ in t would be necessarily the root of a subtree of t equal to t, that is, it is the root of t. □

DEFINITION 4.5 : Let $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$. A call trace τ is *disjoint* from $e \Downarrow v$ if in its proof tree[4] there are no instances of (FJ-INVK) where $v_0.m(\overline{v}) \in dom(\tau)$.

LEMMA 4.6 : If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, then, for all τ disjoint from $e \Downarrow v$, we have $e, \emptyset, \tau \Downarrow v, \emptyset$.

Proof: The proof is by induction on the definition of $e \Downarrow v$.

(FJ-FIELD) Let τ be a call trace disjoint from $e.f \Downarrow v_i$. Since $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, with $v = \texttt{new}\ C(v_1, \ldots, v_n)$, holds by hypothesis, and τ is, by definition, also disjoint from $e \Downarrow v$, we get $e, \emptyset, \tau \Downarrow v, \emptyset$ by induction hypothesis. Then, since $unfold(v, \emptyset) = v$, we get $e.f, \emptyset, \tau \Downarrow v_i, \emptyset$ by rule (FIELD).

(FJ-NEW) Let τ be a call trace disjoint from $\texttt{new}\ C(e_1, \ldots, e_n) \Downarrow \texttt{new}\ C(v_1, \ldots, v_n)$. For all $i \in 1..n$, since $\mathcal{I}_{\text{FJ}} \vdash e_1 \Downarrow v_i$ holds by hypothesis, and τ is, by definition, also disjoint from $e_i \Downarrow v_i$, we get $e_i, \emptyset, \tau \Downarrow v_i, \emptyset$ by induction hypothesis. Then, we get $\texttt{new}\ C(e_1, \ldots, e_n), \emptyset, \tau \Downarrow \texttt{new}\ C(v_1, \ldots, v_n), \emptyset$ by rule (NEW).

(FJ-INVK) Let τ be a call trace disjoint from $e_0.m(e_1, \ldots, e_n) \Downarrow v$. For all $i \in 0..n$, since $\mathcal{I}_{\text{FJ}} \vdash e_i \Downarrow v_i$ holds by hypothesis, and τ is, by definition, also disjoint from $e_i \Downarrow v_i$, we get $e_i, \emptyset, \tau \Downarrow v_i, \emptyset$ by induction hypothesis. Set $\overline{v} = v_1 \ldots v_n$ and $e' = e[v_0/\texttt{this}][\overline{v}/\overline{x}]$. By hypothesis, $\mathcal{I}_{\text{FJ}} \vdash e' \Downarrow v$ and, by definition, τ is also disjoint from $e' \Downarrow v$; furthermore, by Lemma 4.4, e' cannot occur twice in the proof tree for $e' \Downarrow v$, hence $\tau\{v_0.m(\overline{v}) : x\}$ is disjoint from $e' \Downarrow v$, for any fresh variable x. Then, by induction hypothesis, we have $e', \emptyset, \tau\{v_0.m(\overline{v}) : x\} \Downarrow v, \emptyset$, thus we get $e_0.m(e_1, \ldots, e_n), \emptyset, \tau \Downarrow v, \emptyset$ by rule (INVK-OK).

□

We can now prove the conservativity result for coFJ operational semantics.

Proof (Theorem 4.2)*:* The right-to-left direction follows from Lemma 4.6, since $\emptyset$ is disjoint from any expression, while the other direction follows from the right-to-left one and Theorem 4.1. □

For coFJ operational semantics we can prove an additional result, characterizing derivable judgements which produce an empty environment. The meaning is that all results obtained *without using the codefinitions* are original FJ results.

LEMMA 4.7 : If $e, \emptyset, \tau \Downarrow \mathrm{v}, \emptyset$ holds, then v is an FJ value v, and $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$.

4 Unique thanks to Lemma 4.3.

5

coFJ intermediate semantics

In this chapter, we introduce coFJ *intermediate* semantics, which bridges the gap between the abstract and the operational one. This semantics is *abstract* in the sense that values are interpreted coinductively, which means that objects are possibly infinite and, moreover, evaluation is non-deterministic. However, differently from typical coinductive handling of infinite values, it is *inductive*, since it relies on detection of cyclic calls. As we will see later in Chapter 7, the intermediate semantics is crucial in the proof of soundness of the operational semantics with respect to the abstract one; however, we believe that this semantics is interesting in itself, as motivated below.

In the abstract semantics (Chapter 3), methods manipulate possibly *infinite* objects, e.g., the infinite list of natural numbers, and evaluation is modeled as a *relation* between expressions and values, defined by an *inference system with corules* [7, 16]. Hence, in particular, the input/output relation denoted by a method is essentially the coinductive interpretation of its recursive definition, which however, depending on the codefinition, can be restricted to a smaller relation.

In the operational semantics (Chapter 4), methods manipulate *finite representations* (by equations) of possibly infinite objects, and their interpretation is defined by an *inductive* inference system based on *cycle detection*. That is, the operational semantics keeps track of pending method calls. Thus, when an already processed call is found, this does not lead to non-termination as in standard semantics of recursion, since the cycle is detected, and the corresponding codefinition is evaluated.

Due to its double nature, the intermediate semantics shows that the mechanism of cycle detection outlined in previous chapters is orthogonal to the problem of representation of infinite values, which is of course fundamental for an executable semantics. Thus, the two issues can be investigated independently.

Moreover, we believe this semantics to be the inductive counterpart of the abstract semantics restricted to *regular* proof trees, that is, trees with a finite number of different subtrees. That is, besides being *sound* with respect to the abstract semantics as we proved in [3], we conjecture this semantics to be *complete* as well.

Whereas we leave such completeness proof to further work, in this chapter we describe in detail the semantics, and illustrate a key feature to obtain completeness. This feature is *non-determinism* in cycle detection, that is, the

cd	$::=$	`class` C `extends` C' $\{\ \overline{fd}\ \overline{md}\ \}$	class declaration
fd	$::=$	$C\,f$;	field declaration
md	$::=$	$C\ m(C_1\,x_1, \ldots, C_n\,x_n)\ \{e\}$ [`corec` $\{e'\}$]	method declaration with codefinition
$e \in \mathcal{E}$	$::=$	$x \mid e.f \mid$ `new` $C(\overline{e}) \mid e.m(\overline{e})$	expression
$v \in \mathcal{V}^{\mathrm{a}}$	$::=_{\mathrm{co}}$	`new` $C(\overline{v})$	possibly infinite object
$e \in \mathcal{E}^{\mathrm{a}}$	$::=$	$x \mid e.f \mid$ `new` $C(\overline{e}) \mid e.m(\overline{e}) \mid v$	runtime expression

FIGURE 5.1 coFJ syntax

fact that it does not necessarily happens the first time a cycle is found.

5.1 Formal definition

We report coFJ syntax in Figure 5.1 for the reader's convenience. The rules for the intermediate semantics are give in Figure 5.2.

The semantic judgment has shape $e, \rho, S \Downarrow_{\mathrm{IN}} v, S'$ and, comparing with $e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'$ in the operational semantics, no variables are introduced for calls; ρ and S play the role of the ck and non ck part of τ, respectively, keeping track of already processed calls. Moreover, ρ directly associates to a call its value to be used in the checking step, which in σ is associated to the corresponding variable. Finally, S' plays the role of σ', tracing the calls for which the codefinition has been evaluated, hence the checking step will be needed. The rules are analogous to those of Figure 4.2, with the difference that, for an already processed call $c \in S$, either rule (IN-invk-ok) or rule (IN-corec) can be applied. In other words, evaluation of the codefinition is not necessarily triggered when the *first* cycle is detected.

By replacing method parameters by arguments, we obtain *runtime expressions* admitting infinite objects as subterms. The set $\mathcal{E}$ of FJ (source) expressions is a subset of the set $\mathcal{E}^{\mathrm{a}}$ of runtime expressions. We shortly denominate *call* a method call where the receiver and arguments have been evaluated.

To give more details about this semantics, we have that when a call is found the first time, it is added to S. If, during the evaluation of the corresponding method body, the same call is found (*cyclic call*), this is detected thanks to its presence in S, and the codefinition can be evaluated instead.

The set S' keeps track of calls *to be checked*, that is, those for which the codefinition has been evaluated. Indeed, if the evaluation of a method body has been completed by using the codefinition for cyclic calls, then an association from the call to the resulting value v is added to ρ, and an additional *checking step* is performed. That is, the method body is evaluated once more assuming v as result for cyclic calls, and v should be obtained in turn.

Rule (in-val) states that an object evaluates to itself, and is needed for infinite objects, whereas for finite objects rule (in-new) would be enough. Rules (in-field) and (in-new) are standard; only note that the set of calls to be checked in the conclusion is the union of those in the premise(s).

$e \in \mathcal{E}^{a}$	$::=$	$x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e}) \mid v$	runtime expression
c	$::=$	$v.m(\overline{v})$	call
S	$::=$	$c_1\ \ldots\ c_n \quad (n \geq 0)$	set of calls
ρ	$::=$	$c_1{:}v_1 \ldots c_n{:}v_n \quad (n \geq 0)$	environment

$$\text{(IN-VAL)}\ \frac{}{v, \rho, S \Downarrow_{\text{IN}} v, \emptyset}$$

$$\text{(FIELD)}\ \frac{e, \rho, S \Downarrow_{\text{IN}} v, S'}{e.f, \rho, S \Downarrow_{\text{IN}} v_i, S'} \quad \begin{array}{l} v = \texttt{new}\ C(v_1, \ldots, v_n) \\ \mathit{fields}(C) = f_1 \ldots f_n \\ f = f_i,\ i \in 1..n \end{array}$$

$$\text{(IN-NEW)}\ \frac{e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 1..n}{\texttt{new}\ C(e_1, \ldots, e_n), \rho, S \Downarrow_{\text{IN}} \texttt{new}\ C(v_1, \ldots, v_n), \bigcup_{i \in 1..n} S'_i}$$

In all the following rules: $\begin{array}{l} \overline{e} = e_1, \ldots, e_n \\ \overline{v} = v_1 \ldots v_n \\ c = v_0.m(\overline{v}) \\ v_0 = \texttt{new}\ C(_) \end{array}$

$$\text{(IN-INVK-OK)}\ \frac{\begin{array}{c} e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n \\ e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S' \end{array}}{e_0.m(\overline{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i \cup S'} \quad \begin{array}{l} c \notin S' \text{ or } c \in S \\ \mathit{mbody}(C, m) = (\overline{x}, e) \end{array}$$

$$\text{(IN-INVK-CHECK)}\ \frac{\begin{array}{c} e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n \\ e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S' \\ e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho\{c : v\}, S \Downarrow_{\text{IN}} v, S'' \end{array}}{e_0.m(\overline{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i \cup (S' \setminus \{c\})} \quad \begin{array}{l} c \notin S \\ \mathit{mbody}(C, m) = (\overline{x}, e) \\ c \in S' \end{array}$$

$$\text{(IN-COREC)}\ \frac{\begin{array}{c} e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n \\ e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho, S \Downarrow_{\text{IN}} v, S' \end{array}}{e_0.m(\overline{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i \cup S' \cup \{c\}} \quad \begin{array}{l} c \in S \\ \mathit{co\text{-}mbody}(C, m) = (\overline{x}, e') \\ c \notin \mathit{dom}(\rho) \end{array}$$

$$\text{(IN-LOOK-UP)}\ \frac{e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n}{e_0.m(\overline{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i} \quad \rho(c) = v$$

FIGURE 5.2 coFJ intermediate semantics

Analogously to the operational semantics, there are four rules for method invocation. In all of them the receiver and argument expressions are evaluated to obtain the call $c = v.m(\overline{v})$ (first premise). Then, the behavior is different depending on whether such call has been already processed.

For a call processed for the first time ($c \notin S$), either rule (in-invk-ok) or rule (in-invk-check) can be applied. In both cases, the call is added to the set S, and the method body e, where the receiver replaces `this` and the arguments replace the parameters, is evaluated (second premise).

After this, if $c \notin S'$, then the codefinition has not been used for such call. That is, the result has been obtained by standard recursion, and no additional check is needed, hence rule (in-invk-ok) is applied. If $c \in S'$, instead, then rule (in-invk-check) is applied, where an additional check is performed (third premise). That is, the method body (with the same substitutions) is evaluated once more under the assumption that v is the result of the call, as modeled by updating ρ with the association to c of the resulting value. If the same value is returned, we can conclude that it is the correct result of the evaluation of the method call. Note that, since the call is now checked, c is removed from S' in the conclusion of the rule. Otherwise, rule (in-invk-check) cannot be applied since the last premise does not hold.

For an already processed call, instead, there are three possibilities.

If $c \in S$, then rule (in-corec) can be applied, which evaluates the codefinition where the receiver object replaces `this`, the arguments replace the parameters, and, furthermore, an arbitrary value replaces `any`. In addition, c is added to the set of calls to be checked. In this way, the semantics keeps track of the application of rule (in-corec).

If $c \in dom(\rho)$, then we are in a checking step for this call. In this case, rule (in-look-up) can be applied, which simply returns the associated value.

Note that these two rules are mutually exclusive, because, if we are in the checking step for a call c, that is, $c \in dom(\rho)$, we cannot apply rule (in-corec), thanks to the side condition of this rule.

In both cases, rule (in-invk-ok) can be applied instead. Notably, if $c \in S$, then either rule (in-corec) or rule (in-invk-ok) can be non-deterministically applied. The latter choice means that cycle detection is postponed. Analogously, if $c \in dom(\rho)$ then either (in-look-up) or rule (in-invk-ok) can be non-deterministically applied.

5.2 Examples

In this section we discuss examples to show the main feature of the intermediate semantics, that is, its non-determinism. An example of program is reported below.

```
class List extends Object {
  bool allPos() {true}
  List noRep() { new EmptyList() }
}
```

```
class EmptyList extends List { }

class NonEmptyList extends List {

  int head; List tail;

  bool allPos() {
    if (this.head<=0) false
    else this.tail.allPos()
  } corec {true}

  List noRep() {
        let l = this.tail.noRep() in
    if (l.member(this.head)) l
    else new NonEmptyList(this.head,l)
    } corec { new EmptyList() }
}

class ListFactory extends Object {
  NonEmptyList oneTwo() {
      new NonEmptyList(1,this.twoOne())
  } corec {any}
  NonEmptyList twoOne() {
    new NonEmptyList(2,this.oneTwo())
  } corec {any}
}
```

The structure of the program, the definition of method `allPos` and the other auxiliary constructs are analogous to Section 2.3. We recall a few notions for convenience.

The first three classes provide a standard implementation of lists. We use abbreviations to denote `List` objects, e.g.,

```
new NonEmptyList(1, new NonEmptyList(2, EmptyList))
```

will be denoted by [1,2].

We used some additional standard constructs, such as conditional and primitive types `bool` and `int`; to avoid to use abstract methods, `List` provides the default implementation of `allPos` and `noRep` on empty lists, overridden in `NonEmptyList`.

Class `ListFactory` defines methods for building infinite lists. For instance, by invoking `oneTwo()`, we can build the list containing infinitely many alternating occurences of 1 and 2, abbreviated $[1, 2]^\omega$ in the following. Indeed, after an intermediate call `twoOne()`, the same call `oneTwo()` is processed, and (IN-COREC) can be applied, which in this case returns an arbitrary value u, so we obtain $1{:}2{:}u$ for the initial call. In the checking step, we should obtain in turn $1{:}2{:}u$ by evaluating `oneTwo()` assuming $1{:}2{:}u$ as result of the cyclic call, and this only happens for $u = [1, 2]^\omega$.

Method `allPos`, checking that all the elements of a list of integers are positive, shows a more significant use of the codefinition. By invoking $[1, 2]^\omega$`.allPos()`,

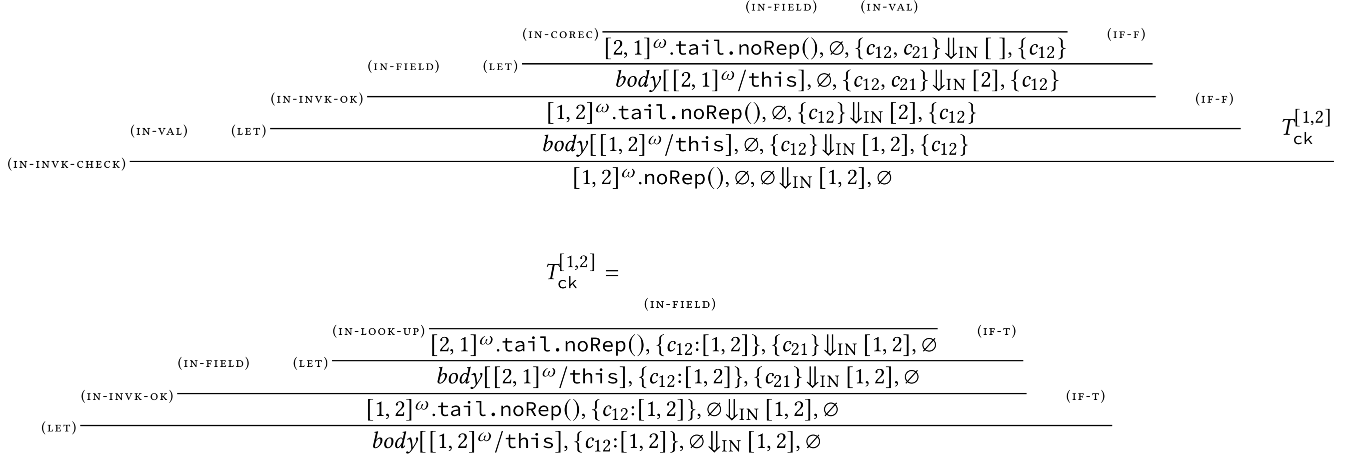

FIGURE 5.3 Proof tree for $[1,2]^\omega$`.noRep()`$, \emptyset, \emptyset \Downarrow_{\text{IN}} [1,2], \emptyset$

when the cyclic call is detected, the codefinition returns `true`, hence the initial call as well. The checking step assuming `true` as result is successful.

Note that, as discussed in previous section, rule (in-corec) is not necessarily applied when the *first* cycle is detected. For instance, in the $[1,2]^\omega$`.allPos()` example, (in-corec) could be applied for an intermediate call $[2,1]^\omega$`.allPos()`. However, in this example, this non-determinism does not affect the final result.

An interesting method to discuss is `noRep()`, producing the list obtained from the receiver by removing duplicates. We assume a let-in construct with the standard semantics. Again the default implementation is given in `List`.

By calling $[1,2]^\omega$`.noRep()`, we expect both $[1,2]$ and $[2,1]$ to be correct results. Indeed, both are solutions of the equation corresponding to the recursive definition of the result of the call, and indeed both are derived in the abstract semantics in Chapter 3. Possible derivations for the two results in the inductive abstract semantics are shown in Figure 5.3 and Figure 5.4, respectively.

We write c_{12} for the call $[1,2]^\omega$`.noRep()`, and c_{21} for $[2,1]^\omega$`.noRep()`, and *body* for the body of the method. Moreover, in order to keep the derivations readable, we focus on the rules for function calls, whereas, for some others, we only indicate the name of the applied rule, omitting the instantiations.

In both Figure 5.3 and Figure 5.4, rule (in-val) evaluates the receiver object to itself, and the evaluation of the body, where the receiver replaces `this`, is performed.

In the first example, rule (in-corec) is applied when the starting call c_{12} is found a second time, hence the call to be checked is such initial call, and we start the derivation by applying rule (in-invk-check). In this way, the codefinition is evaluated, and the empty list is returned. Then, rule (if-f) builds the list [2], which is turned into $[1,2]$ by another application of the same rule. Lastly, the result [1,2] is returned and the checking step can start. This last phase is performed in the tree $T^{[1,2]}_{\text{ck}}$. Here, the body of the method,

$$\dfrac{(\text{IN-VAL}) \quad \dfrac{\dfrac{(\text{IN-FIELD}) \quad T_{aux} \quad T^{[2,1]}_{\mathtt{ck}}}{[1,2]^\omega.\mathtt{tail.noRep}(), \emptyset, \{c_{12}\} \Downarrow_{\text{IN}} [2,1], \emptyset}(\text{IN-INVK-CHECK}) \quad (\text{IF-T})}{body[[1,2]^\omega/\mathtt{this}], \emptyset, \{c_{12}\} \Downarrow_{\text{IN}} [2,1], \emptyset}(\text{LET})}{[1,2]^\omega.\mathtt{noRep}(), \emptyset, \emptyset \Downarrow_{\text{IN}} [2,1], \emptyset}(\text{IN-INVK-OK})$$

$$T_{aux} = \dfrac{\dfrac{(\text{IN-FIELD}) \quad \dfrac{\dfrac{(\text{IN-FIELD}) \quad (\text{IN-VAL})}{[1,2]^\omega.\mathtt{tail.noRep}(), \emptyset, \{c_{12}, c_{21}\} \Downarrow_{\text{IN}} [\,], \{c_{21}\}}(\text{IN-COREC}) \quad (\text{IF-F})}{body[[1,2]^\omega/\mathtt{this}], \emptyset, \{c_{12}, c_{21}\} \Downarrow_{\text{IN}} [1], \{c_{21}\}}(\text{LET})}{[2,1]^\omega.\mathtt{tail.noRep}(), \emptyset, \{c_{12}, c_{21}\} \Downarrow_{\text{IN}} [1], \{c_{21}\}}(\text{IN-INVK-OK}) \quad (\text{IF-F})}{body[[2,1]^\omega/\mathtt{this}], \emptyset, \{c_{12}, c_{21}\} \Downarrow_{\text{IN}} [2,1], \{c_{21}\}}(\text{LET})$$

$$T^{[2,1]}_{\mathtt{ck}} = \dfrac{\dfrac{(\text{IN-FIELD}) \quad \dfrac{\dfrac{(\text{IN-FIELD})}{[1,2]^\omega.\mathtt{tail.noRep}(), \{c_{21}{:}[2,1]\}, \{c_{12}\} \Downarrow_{\text{IN}} [2,1], \emptyset}(\text{IN-LOOK-UP}) \quad (\text{IF-T})}{body[[1,2]^\omega/\mathtt{this}], \{c_{21}{:}[2,1]\}, \{c_{12}\} \Downarrow_{\text{IN}} [2,1], \emptyset}(\text{LET})}{[2,1]^\omega.\mathtt{tail.noRep}(), \{c_{21}{:}[2,1]\}, \emptyset \Downarrow_{\text{IN}} [2,1], \emptyset}(\text{IN-INVK-OK}) \quad (\text{IF-T})}{body[[2,1]^\omega/\mathtt{this}], \{c_{21}{:}[2,1]\}, \emptyset \Downarrow_{\text{IN}} [2,1], \emptyset}(\text{LET})$$

FIGURE 5.4 Proof tree for $[1,2]^\omega.\mathtt{noRep}(), \emptyset, \emptyset \Downarrow_{\text{IN}} [2,1], \emptyset$ (main tree at the top)

where the receiver replaces `this`, is evaluated once more assuming [1,2] as result for the call c_{12}. The structure of the tree is analogous to that for the previous body evaluation, until the same call is processed again. Then, rule (IN-LOOK-UP) rather than (IN-COREC) is applied, and $[1,2]$ is returned. Finally, since the two evaluations of the body lead to the same result, the list $[1,2]$ is returned in the consequence of rule (IN-INVK-CHECK).

In the second example, the call to be checked is c_{21}. Thus, we start the derivation by applying rule (IN-INVK-OK). Then, when c_{21} is found for the first time, rule (IN-INVK-CHECK) is applied, and c_{21} is added to the set S. In this way, when the same call is processed for the second time, rule (IN-COREC) is applied, and the codefinition returns the empty list. The difference from the first example is that now the two rules (IF-F) build [2,1] as result for the starting call c_{12}. The checking step, performed in $T^{[2,1]}_{\mathtt{ck}}$, is analogous to the one described for Figure 5.3.

5.3 Discussion

As already discussed, coFJ abstract semantics is far more expressive than the operational one. Indeed, given the following factory method:

```
class ListFactory extends Object {
 NonEmptyList fromTo(int x) {
  new NonEmptyList(x,new ListFactory.fromTo(x+1))
 } corec {Any}
}
```

a method call such as `new ListFactory.fromTo(0)` evaluates to the infinite list of natural numbers in the abstract semantics, whereas it is undefined in the

semantics based on regular corecursion, since no call will be ever processed twice. Hence, the operational semantics is not *complete* with respect to the abstract one, and to bridge this gap we introduced the intermediate semantics. As we will see later, this latter semantics will be used to prove the soundness result between the executable semantics and the abstract one (this result will be discussed in Chapter 7). However, one could wonder whether a semantics based on regular corecursion can be complete with respect to the *regular* restriction of the abstract semantics, that is, with respect to the evaluation judgments which can be proved by a *regular* proof tree[1]. We conjecture, and plan to prove in further work, that this is true for the abstract inductive semantics described in this paper. Instead, this is not true for the operational semantics, which, as appropriate for an executable semantics, is *deterministic*, in the sense that a strategy of earlier loop detection is chosen. Referring to the `noRep` example before, only the result $[1, 2]$ is obtained, by a proof tree analogous to that shown in Figure 5.3.

We expect the proof of completeness to rely on the result in [17], which shows soundness and completeness of an inductive characterization of the *regular* interpretation of an inference system, that is, judgments which have a regular proof tree.

More generally, what discussed in this chapter shows that a semantics based on regular corecursion (cycle detection) is orthogonal to the problem of representing infinite values. Indeed, note that non-regular values, such as the infinite list of natural number $[0..]$, *are* handled by the intermediate semantics: for instance, we can safely evaluate the call $[0..]$`.member(5)` to `true`, exactly as it happens with lazy evaluation; however, we cannot *generate* the list $[0..]$ as result of a method.

1 That is, a tree with a finite number of different subtrees.

6

Advanced examples

This chapter provides some more complex examples to better understand the operational semantics of coFJ in Chapter 4 and its relationship with the abstract semantics in Chapter 3.

Examples on lists We first show an example motivating the additional checking step (third premise) in rule (invk-check). Essentially, the success of this check for some capsule result corresponds to the existence of an infinite tree in the abstract semantics, whereas the fact that this capsule result is obtained by assuming the codefinition as result of the cyclic call (second premise) corresponds to the existence of a finite tree which uses the codefinition.

Assume to add to our running example of lists of integers a method that returns the sum of its elements. For infinite regular lists, that is, lists ending with a cycle, a result should be returned if the cycle has sum 0, for instance for a list ending with infinitely many 0s, and no result if the cycle has sum different from 0. This can be achieved as follows.

```
class List extends Object { ...
   int sum() {0}
}
class NonEmptyList extends List { ...
     int sum() {this.head + this.tail.sum()} corec {0}
}
```

It is easy to see that the abstract semantics correctly formalizes the expected behavior of this function. For instance, an infinite tree for a judgment $[2, 1]^{\omega}.\mathtt{sum}() \Downarrow v$ only exists for $v = 2 + 1 + v$, and there are no solutions of this equation, hence there is no result. In the operational semantics, by evaluating the body assuming the codefinition as result of the cyclic call (second premise of rule (invk-check)) the spurious result 3 would be returned. This is avoided by the third premise, which evaluates the method body assuming 3 as result of the cyclic call. Since we *do not* get 3 in turn as result, evaluation is stuck, as expected.

Note that the stuckness situation is detected: the last side-condition of rule (invk-check) fails, and a dynamic error (not modeled for simplicity, see the comments to the rule) is raised, likely an exception in an implementation. On the other hand, computations which *never* reach (a base case or) an already encountered call still do not terminate in this operational semantics, exactly as in the standard one, and the fact that this does not happen should be *proved* by suitable techniques, see the Conclusion.

All the examples shown until now have a constant codefinition. We show now an example where this is not enough. Consider the method `remPos()` that removes positive elements. A first attempt at a coFJ definition is the following:

```
class NonEmptyList extends List { ...
  List remPos() {
    if(this.head > 0) this.tail.remPos()
    else new NonEmptyList(this.head,this.tail.remPos())}
  corec {new EmptyList()}
```

Is this definition correct? Actually, it provides the expected behavior on finite lists, and cyclic lists where the cycle contains only positive elements. However, when the cycle contains at least one non positive element, there is no result. For instance, consider the method call $[0,1]^\omega$.`remPos()`. In the abstract semantics, an infinite tree can be constructed for the judgment $[0,1]^\omega.\texttt{remPos}() \Downarrow v$ only if $v = 0 : v$, and this equation clearly only holds for $v = [0]^\omega$. However, no finite tree can be constructed for this judgment using the codefinition. Note that, in the operational semantics, without the additional check (third premise of rule (INVK-CHECK)), we would get the spurious result $[0]$. In order to have a coFJ definition complete with respect to the expected behavior, we should provide a different codefinition for lists with infinitely many non-positive elements.

```
class NonEmptyList extends List { ...
  List remPos() {
    if(this.head > 0) this.tail.remPos()
    else new NonEmptyList(this.head,this.tail.remPos())}
  corec {if (this.allPos() then new EmptyList() else any}
```

Arithmetic with rational and real numbers All real numbers in the closed interval $\{0..1\}$ can be represented by infinite lists $[d_1, d_2, \ldots]$ of decimal digits; more precisely, the infinite list $[d_1, d_2, \ldots]$ represents the real number which is the limit of the series $\sum_{i=1}^{\infty} 10^{-i} d_i$.

It is well-known that all rational numbers in $\{0..1\}$ correspond to either a terminating or repeating decimal, hence they can be represented by infinite regular lists of digits, where terminating decimals end with either an infinite sequence of 0 or an infinite sequence of 9; for instance, the terminating decimal $\frac{1}{2}$ can be represented equivalently by either $[5, 0, 0, \ldots]$ or $[4, 9, 9, \ldots]$, while the repeating decimal $\frac{1}{3}$ is represented by $[3, 3, \ldots]$.

Therefore, in coFJ all rational numbers in $\{0..1\}$ can be effectively represented with infinite precision at the level of the operational semantics; to this aim, we can declare a class `Number` with the two fields `digit` of type `int` and `others` of type `Number`: `digit` contains the leftmost digit, that is, the most significant, while `others` refers to the remaining digits, that is, the number we would obtain by a single left shift (corresponding to multiplication by 10). Since also non-regular values are allowed, in the abstract semantics class `Number` can be used to represent also all irrational numbers in $\{0..1\}$.

We now show how it is possible to compute in coFJ the addition of rational

numbers in {0..1} with infinite precision. We first define the method `carry` which computes the carry of the addition of two numbers: its result is 0 if the sum belongs to {0..1}, 1 otherwise.

```
class Number extends Object { // numbers in {0..1}
  int digit; // leftmost digit
  Number others; // all other digits

  int carry(Number num){ // returns 0 if this+num<=1, 1 otherwise
    if (this.digit+num.digit!=9) (this.digit+num.digit)/10
    else this.others.carry(num.others)
  } corec {0}
}
```

The two numbers `this` and `num` are inspected starting from the most significant digits: if their sum is different from 9, then the carry can be computed without inspecting the other digits, hence the integer division by 10 of the sum is returned. Corecursion is needed when the sum of the two digits equals 9; in this case the carry is the same obtained from the addition of `this.others` and `num.others`.

Finally, in the codefinition the carry 0 is returned; indeed, the codefinition is evaluated only when the sum of the digits for all positions inspected so far is 9 and the same patterns of digits are encountered for the second time. This can only happen for pairs of numbers whose addition is $[9, 9, \ldots]$, that is, 1, hence the computed carry must be 0.

In the abstract semantics, when the addition of n_1 and n_2 yields 1, without the corules it is possible to derive $n_1.\texttt{carry}(n_2) \Downarrow v$ with an infinite proof tree for any value v, but thanks to the codefinition all spurious values are filtered and the only correct value 0 is kept; indeed, if `0` is replaced with `any` in the codefinition, then in the operational semantics the value returned by $n_1.\texttt{carry}(n_2)$ is undetermined when the addition of n_1 and n_2 yields 1, whereas in the abstract semantics any integer is returned for the same case.

Based on method `carry`, we can define method `add` which computes the addition of two numbers, excluding the possible carry in case of overflow.

```
class Number extends Object { ... // declarations as above
  Number add(Number num){ // returns this+num
    new Number(
      (this.digit+num.digit+this.others.carry(num.others))%10,
      this.others.add(num.others))} corec {any}
}
```

For each position, the corresponding digits of `this` and `num` are added to the carry computed for the other digits (`this.others.carry(num.others)`), then the reminder of the division by 10 gives the most significant digit of the result, whereas the others are obtained by corecursively calling the method on the remaining digits (`this.others.add(num.others)`). Since this call is guarded by a constructor call, the codefinition is `any`.

Note that, in the abstract semantics, methods `carry` and `add` correctly work also for irrational numbers.

Method add above is simple, but has the drawback that the same carries are computed more times; hence, in the worst case, the time complexity is quadratic in the period[1] of the two involved repeating decimals. To overcome this issue, we present a more elaborate example where carries are computed only once for any position; this is achieved by method `all_carries` below, which returns the sequence of all carries (hence, a list of binary digits).

Method `simple_add` corecursively adds all digits without considering carries, while method add, defined on top of `simple_add` and `all_carries`, computes the final result. This new version of add is not recursive and, hence. does not need a codefinition.

```
class Number extends Object { ... // declarations as above
  Number all_carries(Number num){ // carries for all positions
    this.simple_carries(num).complete()
  }
  Number simple_carries(Number num){ // carries computed
      immediately
    if(this.digit+num.digit!=9)
      new Number((this.digit+num.digit)/10,
        this.others.simple_carries(num.others))
    else new Number(9,this.others.simple_carries(num.others))
  } corec {any}

  Number complete(){ // computes missing carries marked with 9
    if(this.digit!=9) new Number(this.digit,this.others.complete
        ())
    else this.fill(this.carry_lookahead()).complete()
  } corec {any}

  Number fill(int dig){ // fills with dig all next missing
      carries
    if(this.digit!=9) this else new Number(dig,this.others.fill(
        dig))
  } corec {any}

  int carry_lookahead(){ // returns the next computed carry
    if(this.digit!=9) this.digit else this.others.carry_lookahead
        ()
  } corec {0}

  Number simple_add(Number num){ // addition without carries
    new Number((this.digit+num.digit)%10,
      this.others.simple_add(num.others))
  } corec {any}

  Number add(Number num){
    this.simple_add(num).simple_add(this.all_carries(num).others)
  }
```

1 Indeed, the worst case scenario is when the carry propagates over all digits because their sum is always 9, and this can happen only if the two numbers have the same period.

```
}
```

Distances on graphs The last example of this section involves graphs, which are the paradigmatic example of cyclic data structure. Our aim is to compute the *distance*, that is, the minimal length of a path, between two vertexes[2]. Consider a graph (V, adj) where V is the set of vertexes and $adj : V \longrightarrow \wp(V)$ gives, for each vertex, the set of the adjacent vertexes. Each vertex has an identifier `id` assumed to be unique. We assume a class `Nat`$^\infty$, with subclasses `Nat` with an integer field, and `Infty` with no fields, for naturals and ∞ (distance between unconnected nodes), respectively. Such classes offer methods `succ()` for the successor, and `min(Nat`$^\infty$ `n)` for the minimum, with the expected behaviour (e.g., `succ` in class `Nat`$^\infty$ returns ∞).

```
class Vertex extends Object {
  Id id; AdjList adjVerts;
  Nat∞dist(Id id) {
    this.id==id?new Nat(0):this.adjVerts.dist(id).succ()}
  corec {new Infty()}
}

class AdjList extends Object { }
class EAdjList extends AdjList {
  Nat∞dist(Id id) { new Infty() }
}
class NEAdjList extends AdjList {
  Vertex vert; AdjList adjVerts;
  Nat∞dist(Id id) {this.vert.dist(id).min(this.adjVerts.dist(id)
      )}
}
```

Clearly, if the destination `id` and the source node coincide, then the distance is 0. Otherwise, the distance is obtained by incrementing by one the minimal distance from an adjacent to `id`, computed by method `dist()` of `AdjList` called on the adjacency list. The codefinition of method `dist()` of class `Vertex` is needed since, in presence of a cycle, ∞ is returned and non-termination is avoided. The same approach can be adopted for visiting a graph: instead of keeping trace of already encountered nodes, cycles are implicitly handled by the loop detection mechanism of coFJ.

2 The example can be easily adapted to weighted paths.

7

Soundness

In this chapter we provide a proof of soundness of the operational semantics with respect to the abstract one, which roughly means that a value derived using the rules in Figure 4.2 can also be derived by those in Figure 3.1. However, this statement needs to be refined, since values in the two semantics are different: possibly infinite objects in the abstract semantics, and capsules in the operational semantics.

We define a relation from capsules to abstract objects, formally express soundness through this relation, and introduce an intermediate semantics to carry out the proof in two steps.

7.1 From capsules to infinite objects

Intuitively, given a capsule (v, σ), we get an abstract value by instantiating variables in v with abstract values, in a way consistent with σ. To make this formal, we need some preliminary definitions.

A *substitution* θ is a function from variables to abstract values. We denote by $e\,\theta$ the abstract expression obtained by applying θ to e. In particular, if e is an open value v, then $\mathrm{v}\,\theta$ is an abstract value. Given an environment σ and a substitution θ, the substitution $\sigma[\theta]$ is defined by:

$$\sigma[\theta](x) = \begin{cases} \sigma(x)\,\theta & x \in dom(\sigma) \\ \theta(x) & x \notin dom(\sigma) \end{cases}$$

Then, a *solution* of σ is a substitution θ such that $\sigma[\theta] = \theta$. Let $\mathsf{Sol}(\sigma)$ be the set of solutions of σ. Finally, if (e, σ) is a capsule, we define the set of abstract expressions it denotes as $[\![e, \sigma]\!] = \{ e\,\theta \mid \theta \in \mathsf{Sol}(\sigma) \}$. Note that $[\![\mathrm{v}, \sigma]\!] \subseteq \mathcal{V}^{\mathrm{a}}$, for any capsule (v, σ). We now show an operational characterization of the semantic equality.

THEOREM 7.1 : $[\![\mathrm{v}_1, \sigma_1]\!] = [\![\mathrm{v}_2, \sigma_2]\!]$ iff $(\mathrm{v}_1, \sigma_1) \approx_R (\mathrm{v}_2, \sigma_2)$, for some σ_1, σ_2-renaming R.

To prove this result we need some auxiliary definitions and lemmas. The *tree expansion* of a capsule (v, σ) is the possibly infinite open value coinductively defined as follows:

$$T(\mathrm{v}, \sigma) = \begin{cases} x & \mathrm{v} = x \text{ and } unfold(x, \sigma) \uparrow \\ \mathsf{new}\ C(T(\mathrm{v}_1, \sigma), \ldots, T(\mathrm{v}_n, \sigma)) & unfold(\mathrm{v}, \sigma) = \mathsf{new}\ C(\mathrm{v}_1, \ldots, \mathrm{v}_n) \end{cases}$$

The next proposition shows relations between solutions and tree expansion of a capsule.

PROPOSITION 7.2 : Let (v,σ) be a capsule and $\theta \in \mathsf{Sol}(\sigma)$, then

1. if $\mathit{unfold}(\mathrm{v},\sigma)\uparrow$ then $\mathrm{v} = x$ and $x \overset{\sigma}{\leftrightarrow} x$
2. $\mathit{FV}(T(\mathrm{v},\sigma)) \subseteq \{x \in \mathit{dom}(\sigma) \mid x \overset{\sigma}{\leftrightarrow} x\}$
3. if $x \overset{\sigma}{\leftrightarrow} y$ then $\theta(x) = \theta(y)$
4. if $\mathit{unfold}(\mathrm{v},\sigma) = \mathsf{new}\ C(\mathrm{v}_1,\ldots,\mathrm{v}_n)$ then $\mathrm{v}\,\theta = \mathsf{new}\ C(\mathrm{v}_1\,\theta,\ldots,\mathrm{v}_n\,\theta)$
5. $\mathrm{v}\,\theta = T(\mathrm{v},\sigma)\,\theta$

Given a relation R on variables, we will denote by Rop the opposite relation and by $=_R$ the equality of possibly infinite open values up-to R, coinductively defined by the following rules:

$$\frac{}{x =_R y}\ xRy \qquad \frac{t_i =_R s_i \quad \forall i \in 1..n}{\mathsf{new}\ C(t_1,\ldots,t_n) =_R \mathsf{new}\ C(s_1,\ldots,s_n)}$$

It is easy to check that

- R is a σ_1,σ_2-renaming iff Rop is a σ_2,σ_1-renaming,
- $(\mathrm{v}_1,\sigma_1)\approx_R(\mathrm{v}_2,\sigma_2)$ iff $(\mathrm{v}_2,\sigma_2)\approx_{\mathit{Rop}}(\mathrm{v}_1,\sigma_1)$,
- $t_1 =_R t_2$ iff $t_2 =_{\mathit{Rop}} t_1$.

We have the following lemmas:

LEMMA 7.3 : $(\mathrm{v}_1,\sigma_1)\approx_R(\mathrm{v}_2,\sigma_2)$ iff $T(\mathrm{v}_1,\sigma_1){=_R}T(\mathrm{v}_2,\sigma_2)$, for each σ_1,σ_2-renaming R.

Proof: The proof is immediate by coinduction in both directions. □

LEMMA 7.4 : If $T(\mathrm{v}_1,\sigma_1){=_R}T(\mathrm{v}_2,\sigma_2)$, where R is a σ_1,σ_2-renaming, then $[\![\mathrm{v}_1,\sigma_1]\!] = [\![\mathrm{v}_2,\sigma_2]\!]$.

PROPOSITION 7.5 : If $[\![\mathrm{v}_1,\sigma_1]\!] = [\![\mathrm{v}_2,\sigma_2]\!]$ then

1. if $\mathit{unfold}(\mathrm{v}_1,\sigma_1)\uparrow$ then $\mathit{unfold}(\mathrm{v}_2,\sigma_2)\uparrow$,
2. if $\mathit{unfold}(\mathrm{v}_1,\sigma_1) = \mathsf{new}\ C(\mathrm{v}_{1,1},\ldots,\mathrm{v}_{1,n})$ then $\mathit{unfold}(\mathrm{v}_2,\sigma_2) = \mathsf{new}\ C(\mathrm{v}_{2,1},\ldots,\mathrm{v}_{2,n})$ and, for all $i \in 1..n$, $[\![\mathrm{v}_{1,i},\sigma_1]\!] = [\![\mathrm{v}_{2,i},\sigma_2]\!]$.

LEMMA 7.6 : If $[\![\mathrm{v}_1,\sigma_1]\!]{=}[\![\mathrm{v}_2,\sigma_2]\!]$ then $T(\mathrm{v}_1,\sigma_1) =_R T(\mathrm{v}_2,\sigma_2)$, for some σ_1,σ_2-renaming R.

Proof (Theorem 7.1): The right-to-left direction follows from Lemma 7.3 and Lemma 7.4, while the other direction follows from Lemma 7.6 and Lemma 7.3. □

Since by definition $\approx$ is equal to $\approx_R$ for some R, applying Lemma 7.3 and Lemma 7.4 we get that if $(\mathrm{v}_1,\sigma_1)\approx(\mathrm{v}_2,\sigma_2)$ then $[\![\mathrm{v}_1,\sigma_1]\!] = [\![\mathrm{v}_2,\sigma_2]\!]$. Actually we can prove a stronger result:

LEMMA 7.7 : If $(\mathrm{v}_1,\sigma_1)\approx_R(\mathrm{v}_2,\sigma_2)$ for some strict σ_1,σ_2-renaming R, then, for each solution $\theta \in \mathsf{Sol}(\sigma_1 \cap \sigma_2)$, there are $\theta_1 \in \mathsf{Sol}(\sigma_1)$ and $\theta_2 \in \mathsf{Sol}(\sigma_2)$ such that $\mathrm{v}_1\,\theta_1 = \mathrm{v}_2\,\theta_2$ and, for all $x \in \mathit{dom}(\sigma_1 \cap \sigma_2)$, $\theta_1(x) = \theta(x) = \theta_2(x)$.

7.2 Statement and proof

We can now formally state the soundness result:

THEOREM 7.8 : If $e, \emptyset, \emptyset \Downarrow \mathrm{v}, \sigma$, then, for all $v \in [\![\mathrm{v}, \sigma]\!]$, $(\mathcal{I}_{\mathrm{FJ}}, \mathcal{I}_{\mathrm{FJ}}^{co}) \vdash e \Downarrow v$.

This main result is about the evaluation of source expressions, hence both the environment and the call trace are empty. To carry out the proof we need to generalize the statement.

THEOREM 7.9 (Soundness): If $e, \sigma, \emptyset \Downarrow \mathrm{v}, \sigma'$, then, for all $\theta \in \mathsf{Sol}(\sigma')$, $(\mathcal{I}_{\mathrm{FJ}}, \mathcal{I}_{\mathrm{FJ}}^{co}) \vdash e\,\theta \Downarrow \mathrm{v}\,\theta$.

To show that this is actually a generalization, set $\sigma_1 \leq \sigma_2$ if $dom(\sigma_1) \subseteq dom(\sigma_2)$, and, for all $x \in dom(\sigma_1)$, $\sigma_1(x) = \sigma_2(x)$. We use the following lemmas.

LEMMA 7.10 : If $\sigma_1 \leq \sigma_2$, then $\mathsf{Sol}(\sigma_2) \subseteq \mathsf{Sol}(\sigma_1)$.

Proof: Let $\theta \in \mathsf{Sol}(\sigma_2)$. Since $\sigma_1 \leq \sigma_2$, we have that, for all $x \in dom(\sigma_1)$, $\sigma_1(x) = \sigma_2(x)$. Hence, $\sigma_1[\theta](x) = \sigma_1(x)\,\theta = \sigma_2(x)\,\theta = \sigma_2[\theta](x) = \theta(x)$. Thus we get $\theta \in \mathsf{Sol}(\sigma_2)$. □

LEMMA 7.11 : If $e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'$, then $\sigma \leq \sigma'$.

In the statement of Theorem 7.9, thanks to Lemma 7.11, we know that $\sigma \leq \sigma'$, hence, by Lemma 7.10, $\theta \in \mathsf{Sol}(\sigma)$, thus $e\,\theta \in [\![e, \sigma]\!]$. Theorem 7.9 implies Theorem 7.8, since, when $\sigma = \emptyset$, e is closed, hence $e\,\theta = e$, and all elements in $[\![\mathrm{v}, \sigma']\!]$ have shape $\mathrm{v}\,\theta$ with $\theta \in \mathsf{Sol}(\sigma')$.

In order to prove Theorem 7.9, we rely on the intermediate semantics defined in Chapter 5.

By relying on the intermediate semantics, we can prove Theorem 7.9 in two steps:

1. Firstly we prove that the operational semantics is sound w.r.t. the intermediate semantics (Theorem 7.12).
2. Then, we show that the intermediate semantics is sound w.r.t. the abstract semantics (Theorem 7.14).

At the beginning of Chapter 4, we mentioned two issues for an operational semantics: representing infinite objects in a finite way, and replacing infinite (regular) proof trees by finite proof trees. This proof technique nicely shows that the two issues are orthogonal: notably, detection of cyclic calls is independent from the format of values.

To express the soundness of the operational semantics w.r.t. the intermediate one, we need to formally relate the two judgments. First of all, a call trace τ is the disjoint union of two maps τ^{ck} and $\tau^{\neg\mathrm{ck}}$ into tagged and non-tagged variables, respectively. Then, given an environment σ, we define the following sets of (operational) calls:

- $S^{\tau} = dom(\tau^{\neg\mathsf{ck}})$
- $S^{\tau,\sigma} = dom(\sigma \circ \tau^{\neg\mathsf{ck}})$, where $\circ$ is the composition of partial functions
- $S^{\tau,\sigma,\sigma'} = S^{\tau,\sigma'} \setminus S^{\tau,\sigma}$

For S set of calls and θ substitution, we abbreviate the set of abstract calls $S\,\theta$ by S_θ. Note that $S^{\tau,\sigma}_\theta \subseteq S^{\tau}_\theta$ and, if $\sigma_1 \leq \sigma_2$, then $S^{\tau,\sigma_1}_\theta \subseteq S^{\tau,\sigma_2}_\theta$. Finally, $\rho^{\tau}_\theta(c\,\theta) = v$ iff $v = \theta(\tau^{\mathsf{ck}}(c))$.

Then, the soundness result can be stated as follows:

THEOREM 7.12 (Soundness operational w.r.t. intermediate): If $e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'$ then, for all $\theta \in \mathsf{Sol}(\sigma')$, there exists S such that $S^{\tau,\sigma,\sigma'}_\theta \subseteq S \subseteq S^{\tau,\sigma'}_\theta$ and, $e\,\theta, \rho^{\tau}_\theta, S^{\tau}_\theta \Downarrow_{\mathrm{IN}} \mathrm{v}\,\theta, S$.

In particular, the bounds on S ensure that it is empty when $\tau = \emptyset$. Hence, if $e, \sigma, \emptyset \Downarrow \mathrm{v}, \sigma'$ (hypothesis of Theorem 7.9), then $e\,\theta, \emptyset, \emptyset \Downarrow_{\mathrm{IN}} \mathrm{v}\,\theta, \emptyset$, that is, the hypothesis of Theorem 7.14 below holds.

The proof of the theorem uses the following corollary of Lemma 7.7.

COROLLARY 7.13 : If $(\mathrm{v}_1, \sigma_1) \approx (\mathrm{v}_2, \sigma_2)$, $\theta_1 \in \mathsf{Sol}(\sigma_1)$, $\sigma_1 \leq \sigma_2$, then there is $\theta_2 \in \mathsf{Sol}(\sigma_2)$ such that $\mathrm{v}_1\,\theta_1 = \mathrm{v}_2\,\theta_2$ and, for all $x \in dom(\sigma_1)$, $\theta_1(x) = \theta_2(x)$. Moreover, if $\sigma_1 = \sigma_2$, then $\mathrm{v}_1\,\theta_1 = \mathrm{v}_2\,\theta_1$.

Proof: Immediate from Lemma 7.7, since, if $\sigma_1 \leq \sigma_2$ then $\sigma_1 \cap \sigma_2 = \sigma_1$, and, if $\sigma_1 = \sigma_2$ then $\sigma_1 \cap \sigma_2 = \sigma_1$. □

We now state the second step of the proof: the soundness result of the intermediate semantics with respect to the abstract semantics.

THEOREM 7.14 (Soundness intermediate w.r.t. abstract): If $e, \emptyset, \emptyset \Downarrow_{\mathrm{IN}} v, \emptyset$, then $(\mathcal{I}_{\mathrm{FJ}}, \mathcal{I}^{co}_{\mathrm{FJ}}) \vdash e \Downarrow v$.

The proof uses the bounded coinduction principle (Theorem 2.1), and requires some lemmas. Recall that $\mathcal{I}_{\mathrm{FJ}} \cup \mathcal{I}^{co}_{\mathrm{FJ}} \vdash e \Downarrow v$ means that the judgment $e \Downarrow v$ has a finite proof tree in the (standard) inference system consisting of FJ rules and coFJ corules.

LEMMA 7.15 : If $e, \emptyset, S \Downarrow_{\mathrm{IN}} v, S'$ then $\mathcal{I}_{\mathrm{FJ}} \cup \mathcal{I}^{co}_{\mathrm{FJ}} \vdash e \Downarrow v$ holds.

Proof: By induction on the definition of $e, \emptyset, S \Downarrow_{\mathrm{IN}} v, S'$.

(IN-VAL) By hypothesis, we have that $v, \emptyset, S \Downarrow_{\mathrm{IN}} v, \emptyset$. The thesis is immediate by applying rule (ABS-CO-VAL).

(IN-FIELD) By hypothesis, we have that $e, \emptyset, S \Downarrow_{\mathrm{IN}} v, S'$. By inductive hypothesis, $\mathcal{I}_{\mathrm{FJ}} \cup \mathcal{I}^{co}_{\mathrm{FJ}} \vdash e \Downarrow v$ holds. Thus, we can apply rule (ABS-FIELD) and get the thesis.

(IN-NEW) By hypothesis, we have that $e_i, \emptyset, S \Downarrow_{\mathrm{IN}} v_i, S'_i$ for all $i \in 1..n$. By inductive hypothesis, $\mathcal{I}_{\mathrm{FJ}} \cup \mathcal{I}^{co}_{\mathrm{FJ}} \vdash e_i \Downarrow v_i$ holds for all $i \in 1..n$. Thus, we can apply rule (ABS-NEW) and get the thesis.

(IN-INVK-OK)-(IN-INVK-CHECK) By hypothesis, $e_i, \emptyset, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 0..n$, and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \emptyset, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds. By inductive hypothesis, $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}^{co}_{\text{FJ}} \vdash e_i \Downarrow v_i$ holds for all $i \in 0..n$ and and also $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}^{co}_{\text{FJ}} \vdash e[v_0/\texttt{this}][\overline{v}/\overline{x}] \Downarrow v$ holds. Thus, we can apply rule (ABS-INVK) and get the thesis. Note that, in order to get the thesis, the third premise of rule (INVK-CHECK) has not been used.

(IN-COREC) By hypothesis, $e_i, \emptyset, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 0..n.$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \emptyset, S \Downarrow_{\text{IN}} v, S'$ holds. By inductive hypothesis, $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}^{co}_{\text{FJ}} \vdash e_i \Downarrow v_i$ holds for all $i \in 0..n$, and also $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}^{co}_{\text{FJ}} \vdash e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}] \Downarrow v$ holds. Thus, we can apply rule (ABS-CO-INVK) and get the thesis.

(IN-LOOK-UP) This case is empty since to apply the rule it should be $\rho \neq \emptyset$.

□

LEMMA 7.16 : If $e, \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$, and $e, \rho\{c : v\}, S \Downarrow_{\text{IN}} v, \tilde{S}$, then $\tilde{S} \subseteq S'$.

LEMMA 7.17 : If $e, \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds, and $c \notin S'$, then $e, \rho, S \Downarrow_{\text{IN}} v, S'$.

Proof: For brevity, we use $\hat{S}$ in place of $S \cup \{c\}$. In rules (IN-INVK-OK), (IN-INVK-CHECK), (IN-COREC) and (IN-LOOK-UP), $\hat{S}$ abbreviates $S \cup \{c'\}$, so to distinguish between different calls that could be present in the call trace at the same time.
By induction on the definition of $e, \rho, \hat{S} \Downarrow_{\text{IN}} v, S'$.

(IN-VAL) By hypothesis, $v, \rho, \hat{S} \Downarrow_{\text{IN}} v, \emptyset$. We can trivially get the thesis by rule (IN-VAL)

(IN-FIELD) By hypothesis, $e, \rho, \hat{S} \Downarrow_{\text{IN}} v, S'$, and $c \notin S'$. By inductive hypothesis, $e, \rho, S \Downarrow_{\text{IN}} v, S'$ holds. Thus, we can apply rule (IN-FIELD) and get the thesis.

(IN-NEW) By hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$, and $c \notin S'_i$ for all $i \in 1..n$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. Thus, we can apply rule (IN-NEW) and get the thesis.

(IN-INVK-OK) By hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$, and $c' \notin S'_i$ for all $i \in 0..n$. Also by hypothesis, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, \hat{S} \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds, and $c' \notin S'$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 0..n$ and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds. Thus, we can apply rule (IN-INVK-OK) and get the thesis.

(IN-INVK-CHECK) By hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$, and $c' \notin S'_i$, hence by inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds, for all $i \in 0..n$. Moreover, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, \hat{S} \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds with $c' \notin S'$ (note that it is $c \neq c'$ since the first side condition holds), hence, by inductive hypothesis, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$. Finally, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho\{c : v\}, \hat{S} \Downarrow_{\text{IN}} v, S''$ holds, and, from Lemma 7.16, $c' \notin S''$, hence, by inductive hypothesis, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho\{c : v\}, S \Downarrow_{\text{IN}} v, S''$ holds. Thus, we can apply rule

(IN-INVK-CHECK) and get the thesis.

(IN-COREC) By hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$ holds, and $c' \notin S'_i$ for all $i \in 0..n$. Also by hypothesis, $e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho, \hat{S} \Downarrow_{\text{IN}} v, S'$ holds and $c' \notin S'$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 0..n$ and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho, S \Downarrow_{\text{IN}} v, \varnothing$ holds. Thus, we can apply rule (IN-COREC) and get the thesis.

(IN-LOOK-UP) By hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$ holds, and $c' \notin S'_i$ for all $i \in 1..n$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds $\forall i \in 0..n$. Thus, we can apply rule (IN-LOOK-UP) and get the thesis.

□

LEMMA 7.18 : If $e, \rho, S \Downarrow_{\text{IN}} v, S'$, then the following judgments hold:

1. $e, \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$
2. $e, \rho\{c : v'\}, S \Downarrow_{\text{IN}} v, S'$ for any $v' \in \mathcal{V}^{\text{a}}$.

Proof: The proof of both points is by induction on the definition of $e, \rho, S \Downarrow_{\text{IN}} v, S'$. For brevity, we use $\hat{S}$ in place of $S \cup \{c\}$ and $\hat{\rho}$ in place of $\rho\{c : v'\}$. In rules (IN-INVK-OK), (IN-INVK-CHECK), (IN-COREC) and (IN-LOOK-UP), $\hat{S}$ abbreviates $S \cup \{c'\}$, and $\hat{\rho}$ abbreviates $\rho\{c : v'\}$, so to distinguish between different calls that could be present in the environments at the same time.

For the first part we have:

(IN-VAL) By hypothesis, $v, \rho, S \Downarrow_{\text{IN}} v, \varnothing$. We can trivially get the thesis by rule (IN-VAL)

(IN-FIELD) By hypothesis, $e, \rho, S \Downarrow_{\text{IN}} v, S'$ holds. By inductive hypothesis, $e, \rho, \hat{S} \Downarrow_{\text{IN}} v, S'$ holds. Thus, we get the thesis by applying rule (IN-FIELD).

(IN-NEW) By hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. By inductive hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. Thus, we get the thesis by applying rule (IN-NEW).

(IN-INVK-OK) By hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$, and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds. If $c = c'$ we immediately get the thesis. Otherwise, by inductive hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$, and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, \hat{S} \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds. Thus, we can apply rule (IN-INVK-OK) and get the thesis.

(IN-INVK-CHECK) By hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$, and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho\{c : v\}, S \Downarrow_{\text{IN}} v, S''$ hold. If $c = c'$ we immediately get the thesis. Otherwise, by inductive hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, \hat{S} \cup \{c\} \Downarrow_{\text{IN}} v, S'$ and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho\{c : v\}, \hat{S} \Downarrow_{\text{IN}} v, S''$ hold. Thus, we can apply rule (IN-INVK-CHECK) and get the thesis.

(IN-COREC) By hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. Also by hypothesis, $e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho, S \Downarrow_{\text{IN}} v, S'$ holds. If $c = c'$,

that is, if the call we are adding to S is actually the one for which the codefinition is being evaluated, we immediately get the thesis. Otherwise, by inductive hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$ and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho, \hat{S} \Downarrow_{\text{IN}} v, S'$ holds. Then, we can apply rule (IN-COREC) and get the thesis.

(IN-LOOK-UP) By hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. By inductive hypothesis, $e_i, \rho, \hat{S} \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. Thus, we get the thesis by applying rule (IN-LOOK-UP).

□

LEMMA 7.19 : If $e, \rho\{c : v'\}, S \Downarrow_{\text{IN}} v, S'$ and $c, \rho, S \Downarrow_{\text{IN}} v', \emptyset$, then $e, \rho, S \Downarrow_{\text{IN}} v, S'$.

Proof: In the proof of this lemma we will use ρ' in place of $\rho\{c : v'\}$. In rules (IN-INVK-OK), (IN-INVK-CHECK), (IN-COREC) and (IN-LOOK-UP), ρ' will be used in place of $\rho\{c' : v'\}$ so to distinguish between different calls that could be present in the call trace at the same time.
We proceed by induction on the definition of $e, \rho', S \Downarrow_{\text{IN}} v, S'$.

(IN-VAL) By hypothesis, $v, \rho', S \Downarrow_{\text{IN}} v, \emptyset$. The thesis trivially holds by applying rule (IN-VAL).

(IN-FIELD) By hypothesis, $e, \rho', S \Downarrow_{\text{IN}} v, S'$. By inductive hypothesis, $e, \rho, S \Downarrow_{\text{IN}} v, S'$ holds. Thus, we can apply rule (IN-FIELD) and get the thesis.

(IN-NEW) By hypothesis, $e_i, \rho', S \Downarrow_{\text{IN}} v_i, S'$ holds for all $i \in 1..n$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. Thus, we can apply rule (IN-NEW) and get the thesis.

(IN-INVK-OK) By hypothesis, $e_i, \rho', S \Downarrow_{\text{IN}} v_i, S'$ holds for all $i \in 0..n$. Also by hypothesis, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho', S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds. In order to use the inductive hypothesis, we apply Lemma 7.18 to the hypothesis $c', \rho, S \Downarrow_{\text{IN}} v', \emptyset$ and obtain $c', \rho, S \cup \{c\} \Downarrow_{\text{IN}} v', \emptyset$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$, and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds. Thus, we can apply rule (IN-INVK-OK) and get the thesis.

(IN-INVK-CHECK) By hypothesis, $e_i, \rho', S \Downarrow_{\text{IN}} v_i, S'$ holds for all $i \in 0..n$. Also by hypothesis, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho', S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho'\{c : v\}, S \Downarrow_{\text{IN}} v, S''$ hold. In order to use the inductive hypothesis, we apply Lemma 7.18 to the hypothesis $c', \rho, S \Downarrow_{\text{IN}} v', \emptyset$ and obtain $c', \rho\{c : v\}, S \cup \{c\} \Downarrow_{\text{IN}} v', \emptyset$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$, $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ and $e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho\{c : v\}, S \Downarrow_{\text{IN}} v, S''$ hold. Thus, we can apply rule (IN-INVK-CHECK) and get the thesis.

(IN-COREC) By hypothesis, $e_i, \rho', S \Downarrow_{\text{IN}} v_i, S'$ holds for all $i \in 0..n$. Also by hypothesis, $e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho', S \Downarrow_{\text{IN}} v, S'$ holds. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$ and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho, S \Downarrow_{\text{IN}} v, \emptyset$ holds. Thus, we can apply rule (IN-COREC) and get the thesis.

(IN-LOOK-UP) By hypothesis, $e_i, \rho', S \Downarrow_{\text{IN}} v_i, S'$ holds for all $i \in 0..n$. By inductive hypothesis, $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i$ holds for all $i \in 1..n$. In order to proceed, we have to distinguish between two cases (recall that $c' = v'_0.m'(\overline{v}')$ and $c = v_0.m(\overline{v})$):

- If $c' \neq c$, then $\rho'(c) = \rho(c) = v$, hence we get the thesis by applying rule (LOOK-UP).
- If $c' = c$, then, since $c', \rho, S \Downarrow_{\text{IN}} v', S'$ holds by hypothesis, and $e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'$ holds by inductive hypothesis for all $i \in 0..n$, the judgment $e_0.m(\overline{e}), \rho, S \Downarrow_{\text{IN}} v', S'$ can still be derived.

□

We can now prove Theorem 7.14.

Proof(Theorem 7.14)*:* We take as specification the set $A = \{(e, v) \mid e, \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset\}$, and we use bounded coinduction (Theorem 2.1). We have to prove the following:

BOUNDEDNESS For all $(e, v) \in A$, $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}_{\text{FJ}}^{co} \vdash e \Downarrow v$ holds.

CONSISTENCY For all $(e, v) \in A$, there exist a rule in the abstract semantics having $e \Downarrow v$ as consequence, and such that all its premises are elements of A.

Boundedness follows immediately from Lemma 7.15. We now prove consistency.

Consider a pair $(e, v) \in A$, hence we know that $e, \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$ is derivable. We proceed by case analysis on the last applied rule in the derivation of this judgement.

(IN-VAL) We know that $e = v = \texttt{new}\ C(v_1, \ldots, v_n)$. We choose as candidate rule (ABS-NEW). We have to show that, for all $i \in 1..n$, $(v_i, v_i) \in A$, that is, $v_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds. We can get the thesis thanks to rule (IN-VAL).

(IN-FIELD) We know that $e = e'.f$ and $e', \emptyset, \emptyset \Downarrow_{\text{IN}} \texttt{new}\ C(v_1 \ldots v_n), \emptyset$. We choose as candidate rule (ABS-FIELD), with conclusion $e'.f \Downarrow v_i$. We have to show that $(e', \texttt{new}\ C(v_1 \ldots v_v)) \in A$, that is, $e', \emptyset, \emptyset \Downarrow_{\text{IN}} \texttt{new}\ C(v_1 \ldots v_v), \emptyset$ holds, but this is true by hypothesis.

(IN-NEW) We know that $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds for all $i \in 1..n$. We choose as candidate rule (ABS-NEW). We have to show that, for all $i \in 1..n$, $(e_i, v_i) \in A$, that is, $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds, but this is true by hypothesis.

(IN-INVK-OK) We know that $e = e_0.m(\overline{e})$, $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds for all $i \in 0..n$, $c = v_0.m(\overline{v})$, $mbody(C, m) = (\overline{x}, e')$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \emptyset, \{c\} \Downarrow_{\text{IN}} v, \emptyset$ holds. We choose as candidate rule (ABS-INVK). We have to show that, for all $i \in 0..n$, $(e_i, v_i) \in A$, and $(e'[v_0/\texttt{this}][\overline{v}/\overline{x}], v) \in A$. That is, that the following judgments hold: $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ for all $i \in 0..n$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$. The judgments in the first set hold

by hypothesis. The last judgment holds thanks to Lemma 7.17, where $S' = \emptyset$.

(IN-INVK-CHECK) We know that $e = e_0.m(\overline{e})$, $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds for all $i \in 0..n$, $c = v_0.m(\overline{v})$, $mbody(C, m) = (\overline{x}, e')$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \{c : v\}, \emptyset \Downarrow_{\text{IN}} v, \emptyset$ holds. We choose as candidate rule (ABS-INVK). We have to show that for all $i \in 0..n$, $(e_i, v_i) \in A$, and $(e'[v_0/\texttt{this}][\overline{v}/\overline{x}], v) \in A$. That is, that the following judgments hold: $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ for all $i \in 0..n$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$. The judgments in the first set hold by hypothesis. The last judgment holds thanks to Lemma 7.19, since from the hypothesis we easily get $c, \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$.

(IN-COREC) Empty case since to apply the rule it should be $S \neq \emptyset$.

(IN-LOOK-UP) Empty case since to apply the rule it should be $\rho \neq \emptyset$.

□

8

Related and future work

In the first part of the thesis, we focused on making regular corecursion *flexible*, which means that the programmer can specify the behaviour in case a cycle is detected. Language constructs to achieve such flexibility have been proposed in the logic [2, 6, 18], functional [22, 23], and object-oriented [9, 10] paradigm.

LOGIC PARADIGM As already mentioned, the idea of regular corecursion (keeping track of pending function/method calls, so to detect cyclic calls), originates from *co-SLD resolution* [30, 31, 8], which is a sound resolution procedure based on cycle detection. Namely, it is a modification of SLD resolution where, when the same goal is found the second time, it is considered successful. In this way it is possible to define coinductive predicates. Correspondingly, models are subsets of the *complete Herbrand basis*, that is, the set of ground atoms built on arbitrary (finite or infinite) terms, and the declarative semantics is the greatest fixed point of the monotone function associated with a program. Anyway, in coinductive logic programming, only standard coinduction is supported. The notion of `finally` clause, introduced in [2], allows the programmer to specify a fact to be resolved when a cycle is detected, instead of simply accepting the atom. The approach has been refined in [6, 18], by relying on the formal framework of generalized inference systems. That is, the programmer can write special clauses corresponding to corules, called *coclauses*, so that, when an atom is found for the second time, standard SLD resolution is triggered in the program enriched by the coclauses.

FUNCTIONAL PARADIGM *CoCaml*[1] [22, 23] is a fully-fledged extension of OCaml supporting non-well-founded data types and corecursive functions. CoCaml, as OCaml, allows programmers to declare regular values through the let-rec construct, and, moreover, detects cyclic calls as in our approach. However, whereas coFJ immediately evaluates the cyclic call by using the codefinition, the CoCaml approach is divided in two phases. First, a system of equations is constructed, associating with each call a variable and partially evaluating the body of functions, where calls are replaced with associated variables. Then, the system of equations is given to a *solver* specified in the function definition. Solvers can be either pre-defined or written by the programmer in order to enhance flexibility. An advantage that we see in our approach is that the programmer has to write the codefinition (standard code)

1 `www.cs.cornell.edu/Projects/CoCaml`

rather than working at the meta-level to write a solver, which is in a sense a fragment of the interpreter. A precise comparison is difficult for the lack of a simple operational model of the CoCaml mechanism.

OBJECT-ORIENTED PARADIGM An early version of coFJ was proposed in [9]. At that time, however, the framework of inference systems with corules was still to come, so there was no formal model against which to check the given operational semantics, which, indeed, derived spurious results in some cases, as illustrated in Chapter 6. The operational semantics provided in Chapter 4 solves this problem, and is proved to be sound with respect to the abstract semantics. Moreover, we adopt a simpler representation of cyclic objects through capsules [22]. A type system has been proposed [10] for the previous version of coFJ to prevent *unsafe* use of the "undetermined" value. We leave to further work the investigation of typing issues for the approach presented in the thesis.

FUTURE WORK For what concerns the implementation of the coFJ calculus, as for now we have a prototype[2], that implements the *abstract* semantics on top of a Prolog meta-interpreter supporting flexible regular corecursion [6, 18]. In this way, the inference system is naturally translated in Prolog[3], cyclic terms are natively supported, and their equality handled by unification. A fully-fledged interpreter of the *operational* semantics should directly handle these issues and, moreover, attempt to do some optimization.

The current approach does not deal with types: an important concern is to guarantee *type soundness*, statically ensuring that an undetermined value never occurs as receiver of field access or method invocation, as investigated in [10] for the previous coFJ version [9].

Another issue is how to train developers to write codefinitions. Standard recursion is non-trivial as well for beginners, whereas it becomes quite natural after understanding its mechanism. For regular corecursion the same holds, with the additional difficulty of reasoning on infinite structures. Intuitively, the codefinition can be regarded as a base case to be applied when a loop is detected. Moreover, again as for standard recursion, this novel programming style could be integrated with proof techniques to show the correctness of algorithms on cyclic data structures. Such proofs could be mechanized in proof assistants, as Agda, that provide built-in support for coinductive definitions and proofs by coinduction.

Finally, a non-trivial challenge is how to integrate regular corecursion, requiring to detect "the same call", with the notion of mutable state. Likely, some immutability constraints will be needed, or a variant of the model where such a check requires a stateless computation. Another solution is to consider the check as an assertion that can be disabled if the programmer has verified the correctness of the method by hand or assisted by a tool.

Such a style could be conveniently integrated with proof assistants, such as Agda, that provide built-in support for coinductive definitions and proofs

2 https://person.dibris.unige.it/zucca-elena/coFJ_implementation.zip
3 A logic program can be seen as an inference system where judgments are atoms.

by coinduction, to formally prove the correctness of algorithms on cyclic data structures.

The semantics of flexible regular corecursion is the operational counterpart of the abstract one, obtained by considering recursive functions as relations, and recursive definitions (with codefinition) as inference systems (with corules). We prove that the operational semantics is *sound* with respect to that interpretation. Obviously, *completeness* does not hold in general, since the abstract semantics deals with not only cyclic data structures (such as $[2, 1]^\omega$), but arbitrary non-well-founded structures (such as the list of natural numbers). Even considering only regular proof trees in the abstract semantics, in some subtle cases there is more than one admissible result[4], whereas the operational semantics, being deterministic, finds "the first" among such results, as reasonable in an implementation. Such completeness issues could be investigated in further work.

4 For instance, the list with no repetitions extracted from $[1, 2]^\omega$ can be either $[1, 2]$ or $[2, 1]$, as seen in Chapter 5.

PART II

Beyond regular terms

9

Stream calculus

In this chapter we formally introduce non-regular streams by presenting the syntax and discussing the rules of the operational semantics. Similarly to coFJ operational semantics, the behaviour is deterministic, and the aim is to provide a semi-algorithm. We conclude the chapter by providing many examples.

9.1 Formal definition

Figure 9.1 shows the syntax of the calculus.

$\overline{fd}$	$::=$	$fd_1 \ldots fd_n$	program
fd	$::=$	$f(\overline{x}) = se$	function declaration
e	$::=$	$se \mid ne \mid be$	expression
se	$::=$	$x \mid \texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2 \mid ne\texttt{:}se \mid se\hat{}\ \mid se_1\ [nop]\ se_2 \mid f(\overline{e})$	stream expression
ne	$::=$	$x \mid se(ne) \mid ne_1\ nop\ ne_2 \mid 0 \mid 1 \mid 2 \mid \ldots$	numeric expression
be	$::=$	$x \mid \texttt{true} \mid \texttt{false} \mid \ldots$	boolean expression
nop	$::=$	$+ \mid - \mid * \mid /$	numeric operation

FIGURE 9.1 Stream calculus: syntax

A program is a sequence of (mutually recursive) function declarations, for simplicity assumed to only return streams. Stream expressions are variables, conditional expressions, expressions built by stream operators, and function calls. We consider the following stream operators: constructor (prepending a numeric element), tail (we write $se\hat{}$ to indicate the tail of the stream expression se), and pointwise arithmetic operations. Numeric expressions include the access to the i-th[1] element of a stream. We use $\overline{fd}$ to denote a sequence $fd_1, \ldots, fd_n$ of function declarations, and analogously for other sequences.

The operational semantics, given in Figure 9.2, is based on two key ideas:

1. (some) infinite streams are represented in a finite way
2. evaluation keeps track of already processed function calls

To obtain point (1), an equational system is modeled by an *environment* ρ mapping a finite set of variables into *(open) stream values* s, built on top

1 For simplicity, here indexing and numeric expressions coincide, even though indexes are expected to be natural numbers, while values in streams can range over a larger numeric domain.

c	$::=$	$f(\overline{v})$	(evaluated) call
v	$::=$	$s \mid n \mid b$	value
s	$::=$	$x \mid n\!:\!s \mid s\hat{} \mid s_1\ [nop]\ s_2$	(open) stream value
i, n	$::=$	$0 \mid 1 \mid 2 \mid \ldots$	index, numeric value
b	$::=$	$\texttt{true} \mid \texttt{false}$	boolean value
τ	$::=$	$c_1 \mapsto x_1\ \ldots\ c_n \mapsto x_n \quad (n \geq 0)$	call trace
ρ	$::=$	$x_1 \mapsto s_1 \ldots x_n \mapsto s_n \quad (n \geq 0)$	environment

$$\text{(VAL)}\ \frac{}{v, \rho, \tau \Downarrow (v, \rho)} \qquad \text{(CONS)}\ \frac{ne, \rho, \tau \Downarrow (n, \rho) \quad se, \rho, \tau \Downarrow (s, \rho')}{ne\!:\!se, \rho, \tau \Downarrow (n\!:\!s, \rho')} \qquad \text{(TAIL)}\ \frac{se, \rho, \tau \Downarrow (s, \rho')}{se\hat{}, \rho, \tau \Downarrow (s\hat{}, \rho')}$$

$$\text{(NOP)}\ \frac{se_1, \rho, \tau \Downarrow (s_1, \rho_1) \quad se_2, \rho, \tau \Downarrow (s_2, \rho_2)}{se_1\ [nop]\ se_2, \rho, \tau \Downarrow (s_1\ [nop]\ s_2, \rho_1 \sqcup \rho_2)}$$

$$\text{(IF-T)}\ \frac{be, \rho, \tau \Downarrow (\texttt{true}, \rho) \quad se_1, \rho, \tau \Downarrow (s, \rho')}{\texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2, \rho, \tau \Downarrow (s, \rho')} \qquad \text{(IF-F)}\ \frac{be, \rho, \tau \Downarrow (\texttt{false}, \rho) \quad se_2, \rho, \tau \Downarrow (s, \rho')}{\texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2, \rho, \tau \Downarrow (s, \rho')}$$

$$\text{(ARGS)}\ \frac{e_i, \rho, \tau \Downarrow (v_i, \rho_i) \quad \forall i \in 1..n \qquad f(\overline{v}), \widehat{\rho}, \tau \Downarrow (s, \rho')}{f(\overline{e}), \rho, \tau \Downarrow (s, \rho')} \quad \begin{array}{l} \overline{e} = e_1, \ldots, e_n \text{ not of shape } \overline{v} \\ \overline{v} = v_1, \ldots, v_n \\ \widehat{\rho} = \bigsqcup_{i \in 1..n} \rho_i \end{array}$$

$$\text{(INVK)}\ \frac{se[\overline{v}/\overline{x}], \rho, \tau\{f(\overline{v}) \mapsto x\} \Downarrow (s, \rho')}{f(\overline{v}), \rho, \tau \Downarrow (x, \rho'\{x \mapsto s\})} \quad \begin{array}{l} f(\overline{v}) \notin dom(\tau_{\approx\rho}) \\ x \text{ fresh} \\ fbody(f) = (\overline{x}, se) \\ wd(\rho', x, s) \end{array} \qquad \text{(COREC)}\ \frac{}{f(\overline{v}), \rho, \tau \Downarrow (x, \rho)} \quad \tau_{\approx\rho}(f(\overline{v})) = x$$

$$\text{(AT)}\ \frac{se, \rho, \tau \Downarrow (s, \rho') \quad ne, \rho, \tau \Downarrow (i, \rho)}{se(ne), \rho, \tau \Downarrow (n, \rho)} \quad at_{\rho'}(s, i) = n$$

$$\text{(AT-VAR)}\ \frac{at_\rho(\rho(x), i) = n'}{at_\rho(x, i) = n'} \qquad \text{(AT-CONS-0)}\ \frac{}{at_\rho(n\!:\!s, 0) = n} \qquad \text{(AT-CONS-SUCC)}\ \frac{at_\rho(s, i-1) = n'}{at_\rho(n\!:\!s, i) = n'} \quad i > 0$$

$$\text{(AT-TAIL)}\ \frac{at_\rho(s, i+1) = n}{at_\rho(s\hat{}, i) = n} \qquad \text{(AT-NOP)}\ \frac{at_\rho(s_1, i) = n_1 \quad at_\rho(s_2, i) = n_2}{at_\rho(s_1\ [nop]\ s_2, i) = n_1\ nop\ n_2}$$

FIGURE 9.2 Stream calculus: operational semantics

of stream variables, numeric values and the stream operators; consequently, the *result* of the evaluation of a stream expression is a pair (s, ρ), similarly as done with *capsules* [22] to support cyclic references. For instance, $(x, x \mapsto n : x)$ denotes the stream constantly equal to n.

To obtain point (2) above, evaluation has an additional parameter which is a *call trace* similar to the one used in coFJ, that is, a map from function calls where arguments are values (dubbed *calls* for short in the following) into variables.

Altogether, the semantic judgment has shape $e, \rho, \tau \Downarrow (v, \rho')$, where e is the expression to be evaluated, ρ the current environment defining possibly cyclic stream values that can occur in e, τ the call trace, and (v, ρ') the result. The semantic judgments should be indexed by an underlying (fixed) program, omitted for sake of simplicity. Rules use the following auxiliary definitions:

- $\rho \sqcup \rho'$ is the union of two environments, which is well-defined if they have disjoint domains; $\rho\{x \mapsto s\}$ is the environment which gives s on x and coincides with ρ elsewhere; we use analogous notations for call traces.
- $se[\overline{v}/\overline{x}]$ is obtained by the parallel substitution of the variables $\overline{x}$ with the values $\overline{v}$.
- $fbody(f)$ returns the pair of the parameters and the body of the declaration of f, if any, in the assumed program.

Moreover, the rules are parametric with respect to the *well-definedness* and *equality* judgments, for which different definitions will be discussed in Chapters 10 and 14:

- $wd(\rho, x, s)$, that is, by adding the association $x \mapsto s$ to the (well-defined) environment ρ, we still get a well-defined environment
- $v_1 \approx_\rho v_2$, that is, v_1 and v_2 are considered equal in the environment ρ.

The well-definedness judgment is used as a runtime check which should be passed to obtain the result of a method invocation. The equality judgment is indirectly used since the look-up of a call in the the call trace is performed *modulo equality*, that is:

- $c \approx_\rho c'$ iff $c = f(v_1, \ldots, v_n)$, $c' = f(v'_1, \ldots, v'_n)$ and $v_i \approx_\rho v'_i$ for all $i \in 1..n$
- $\tau_{\approx\rho}(c) = x$ if $\tau(c') = x$, $c' \approx_\rho c$ for some c'
- hence, $c \notin dom(\tau_{\approx\rho})$ if there is no $c' \in dom(\tau)$ s.t. $c \approx_\rho c'$.

We denote by $vars(\rho)$ the set of variables occurring in ρ, by $fv(\rho)$ the set of its free variables, that is, $vars(\rho) \setminus dom(\rho)$, and say that ρ is *closed* if $fv(\rho) = \emptyset$, *open* otherwise, and analogously for a result (v, ρ).

Intuitively, a closed result (s, ρ) is well-defined if it denotes a unique stream (infinite sequence of numeric values), and a closed environment ρ is well-defined if, for each $x \in dom(\rho)$, (x, ρ) is well-defined. In other words, the

corresponding set of equations admits a unique solution. For instance, the environment $\{x \mapsto x\}$ is not well-defined, since it is undetermined (any stream satisfies the equation $x = x$); the environment $\{x \mapsto x[+]y, y \mapsto 1:y\}$ is not well-defined as well, since it is undefined (the two equations $x = x \mapsto x[+]y, y = 1:y$ admit no solutions for x). Finally, two stream values s and s' such that the results (s, ρ) and (s', ρ) are closed and well-defined are considered equal in ρ if the unique solutions of the two results are the same stream.

These notions can be generalized to open results and environments, assuming that free variables denote unique streams, as will be formalized in Chapters 10 and 14.

Rules for values and conditional are straightforward. In rules (CONS), (TAIL) and (NOP), arguments are evaluated, while the stream operator is applied without any further evaluation; the fact that the tail and pointwise operators are treated as the stream constructor $_:_$ is crucial to get results which denote non-regular streams, as will be illustrated in Section 9.2. However, when non-constructors are allowed to occur in values, ensuring well-defined results becomes more challenging, because the usual simple syntactic constraints that can be safely used for constructors [14] no longer work (see more details in Chapters 10 and 16).

The rules for function call are based on a mechanism of cycle detection, analogous to that used in the operational and intermediate semantics of coFJ, in Chapter 4 and Chapter 5, respectively. There are two main differences. The first one is that here, for separation of concerns, we are *not* considering flexible corecursion. Hence, the result of a cyclic call, rather than an arbitrary value obtained by evaluating the codefinition, is always the variable associated with the call in the call trace, see rule (COREC) in Figure 9.2, as it happens in coFJ when the codefinition is `any`. As a consequence, the additional check that such value is indeed a solution of the corresponding equation is not needed, hence there is no analogous of rule (INVK-CHECK) in Figure 4.2. The second difference is that here rule (INVK) is equipped with a runtime check ensuring that the result is a well-defined stream.

Moreover, a minor difference is that here rules for function call are given in a modular way. That is, evaluation of arguments is handled by a separate rule (ARGS), whereas the following two rules handle (evaluated) calls.

Rule (INVK) is applied when a call is considered for the first time, as expressed by the first side condition. The body is retrieved by using the auxiliary function *fbody*, and evaluated in a call trace where the call has been mapped into a fresh variable. Then, it is checked that adding the association from such variable to the result of the evaluation of the body keeps the environment well-defined. If the check succeeds, then the final result consists of the variable associated with the call and the updated environment. For simplicity, here execution is stuck if the check fails; an implementation should raise a runtime error instead.

Rule (COREC) is applied when a call is considered for the second time, as expressed by the first side condition (note that cycle detection takes place

$f() = g()$
$g() = 1 : f()$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1, \emptyset, \{f() \mapsto x, g() \mapsto y\} \Downarrow (1, \emptyset)}\,(\text{VAL}) \quad \dfrac{}{f(), \emptyset, \{f() \mapsto x, g() \mapsto y\} \Downarrow (x, \emptyset)}\,(\text{COREC})}{1 : f(), \emptyset, \{f() \mapsto x, g() \mapsto y\} \Downarrow (1 : x, \emptyset)}\,(\text{CONS})}{g(), \emptyset, \{f() \mapsto x\} \Downarrow (y, \{y \mapsto 1 : x\})}\,(\text{INVK})}{f(), \emptyset, \emptyset \Downarrow (x, \{x \mapsto y, y \mapsto 1 : x\})}\,(\text{INVK})$$

FIGURE 9.3 Example of derivation

$f() = g(2 : f())$
$g(s) = 1 : s$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{2, \emptyset, \{f() \mapsto x\} \Downarrow (2, \emptyset)}\,(\text{VAL}) \quad \dfrac{}{f(), \emptyset, \{f() \mapsto x\} \Downarrow (x, \emptyset)}\,(\text{COREC})}{2 : f(), \emptyset, \{f() \mapsto x\} \Downarrow (2 : x, \emptyset)}\,(\text{CONS}) \quad T_1}{g(2 : f()), \emptyset, \{f() \mapsto x\} \Downarrow (y, \{y \mapsto 1 : 2 : x\})}\,(\text{ARGS})}{f(), \emptyset, \emptyset \Downarrow (x, \{x \mapsto y, y \mapsto 1 : 2 : x\})}\,(\text{INVK})$$

$$T_1 = \dfrac{\dfrac{}{1 : 2 : x, \emptyset, \{g(2 : x) \mapsto y, f() \mapsto x\} \Downarrow (1 : 2 : x, \emptyset)}\,(\text{VAL})}{g(2 : x), \emptyset, \{f() \mapsto x\} \Downarrow (y, \{y \mapsto 1 : 2 : x\})}\,(\text{INVK})$$

FIGURE 9.4 Example of derivation

up to equivalence in the environment). The variable x is returned as result. However, there is no associated value in the environment yet; in other words, the result (x, ρ) is open at this point. This means that x is undefined until the environment is updated with the corresponding value in rule (INVK). However, x can be safely used as long as the evaluation does not require x to be inspected; for instance, x can be safely passed as an argument to a function call.

For instance, in the program `f()=g() g()=1:f()` the the judgment `f()`$, \emptyset, \emptyset \Downarrow (x, \rho)$, with $\rho = \{x \mapsto y, y \mapsto$ `1:`$x\}$, is derivable; however, while the final result (x, ρ) is closed, the derivation contains also judgments with open results, as happens for `f()`$, \emptyset, \{$`f()` $\mapsto x,$ `g()` $\mapsto y\} \Downarrow (x, \emptyset)$ and `g()`$, \emptyset, \{$`f()` $\mapsto x\} \Downarrow (y, \{y \mapsto$ `1:`$x\})$. The full derivation is displayed in Figure 9.3.

As another example, in the program `f()=g(2:f()) g(s)=1:s`, the derivation of the judgment `f()`$, \emptyset, \emptyset \Downarrow (x, \rho)$ with $\rho = \{x \mapsto y, y \mapsto$ `1:2:`$x\}$ is built from the derivation of `g(2:`x`)`$, \emptyset, \{$`f()` $\mapsto x\} \Downarrow (y, \{y \mapsto$ `1:2:`$x\})$, corresponding to the evaluation of `g(2:`x`)` where x is an operand of the stream constructor whose result is passed as argument to the call to g, in spite of the fact that x is not defined yet. The full derivation is displayed in Figure 9.4.

Finally, rule (AT) computes the i-th element of a stream expression. After evaluation of the arguments, the numeric result is obtained by the auxiliary

$$undef() = (undef()(0)) : undef()$$

$$\dfrac{T_1 \quad \dfrac{}{undef(), \emptyset, \{undef() \mapsto x\} \Downarrow (x, \emptyset)}\,(\text{COREC})}{\dfrac{(undef()(0)) : undef(), \emptyset, \{undef() \mapsto x\} \Downarrow (?, ?)}{undef(), \emptyset, \emptyset \Downarrow (?, ?)}\,(\text{INVK})}\,(\text{CONS})$$

$$T_1 = {}_{(\text{AT})}\dfrac{\dfrac{}{undef(), \emptyset, \{undef() \mapsto x\} \Downarrow (x, \emptyset)}\,(\text{COREC}) \quad \dfrac{}{0, \emptyset, \{undef() \mapsto x\} \Downarrow (0, \emptyset)}\,(\text{VAL})}{undef()(0), \emptyset, \{undef() \mapsto x\} \Downarrow (?, ?)} \quad at_\emptyset(x, 0) = ?$$

FIGURE 9.5 Example of stuck derivation

judgment $at_\rho(s, i) = n$, inductively defined in the bottom part of the figure. If the stream value is a variable (AT-VAR), then the evaluation is propagated to the associated stream value in the environment, if any. If, instead, the variable is free in the environment, then execution is stuck; again, an implementation should raise a runtime error instead. Figure 9.5 shows an example of stuck derivation.

If the stream value is built by the constructor, then the result is the first element of the stream if the index is 0 (AT-CONS-0); otherwise, the evaluation is recursively propagated to its tail with the predecessor index (AT-CONS-SUCC). Conversely, if the stream is built by the tail operator (AT-TAIL), then the evaluation is recursively propagated to the stream argument with the successor index. Finally, if the stream is built by a pointwise operation (AT-NOP), then the evaluation is recursively propagated to the operands with the same index and then the corresponding arithmetic operation is computed on the results.

9.2 Examples

First we show some simple examples, to explain how regular corecursion works. Then we provide some more significant examples.

Consider the following function declarations:

```
repeat(n) = n:repeat(n)
one_two() = 1:two_one()
two_one() = 2:one_two()
```

With the standard semantics of recursion, the calls, e.g., `repeat(0)` and `one_two()` lead to non-termination. Thanks to regular corecursion, instead, these calls terminate, producing as result $(x, \{x \mapsto \texttt{0:}x\})$, and $(x, \{x \mapsto \texttt{1:}y, y \mapsto \texttt{2:}x\})$, respectively. Indeed, when initially invoked, the call `repeat(0)` is added in the call trace with an associated fresh variable, say x. In this way, when evaluating the body of the function, the recursive call is detected as cyclic, the variable x is returned as its result, and, finally, the stream value $\texttt{0:}x$ is associated in the

environment with the result x of the initial call. The evaluation of `one_two()` is similar, except that another fresh variable y is generated for the intermediate call `two_one()`. The formal derivations are given below.

$$\dfrac{\dfrac{(\text{VALUE})\quad \dfrac{}{\texttt{repeat(0)}, \emptyset, \{\texttt{repeat(0)} \mapsto x\} \Downarrow (x, \emptyset)}\,(\text{COREC})}{\texttt{0:repeat(0)}, \emptyset, \{\texttt{repeat(0)} \mapsto x\} \Downarrow (0\!:\!x, \emptyset)}\,(\text{CONS})}{\texttt{repeat(0)}, \emptyset, \emptyset \Downarrow (x, \{x \mapsto 0\!:\!x\})}\,(\text{INVK})$$

$$\dfrac{\dfrac{(\text{VALUE})\quad \dfrac{\dfrac{\dfrac{(\text{VALUE})\quad \dfrac{}{\texttt{one_two()}, \emptyset, \{\texttt{one_two()} \mapsto x,\ \texttt{two_one()} \mapsto y\} \Downarrow (x, \emptyset)}\,(\text{COREC})}{\texttt{2 : one_two()}, \emptyset, \{\texttt{one_two()} \mapsto x,\ \texttt{two_one()} \mapsto y\} \Downarrow (2 : x, \emptyset)}\,(\text{CONS})}{\texttt{two_one()}, \emptyset, \{\texttt{one_two()} \mapsto x\} \Downarrow (y, \{y \mapsto 2 : x\})}\,(\text{INVK})}{\texttt{1 : two_one()}, \emptyset, \{\texttt{one_two()} \mapsto x\} \Downarrow (1 : y, \{y \mapsto 2 : x\})}\,(\text{CONS})}{}}{\texttt{one_two()}, \emptyset, \emptyset \Downarrow (x, \{x \mapsto 1 : y,\ y \mapsto 2 : x\})}\,(\text{INVK})$$

For space reasons, we did not report the application of rule (VALUE). In both derivations, note that rule (COREC) is applied, without evaluating the body once more, when the cyclic call is detected.

The following examples show function definitions whose calls return non-regular streams, notably, the natural numbers, the natural numbers raised to the power of a number, the factorials, the powers of a number, the Fibonacci numbers, and the stream obtained by pointwise increment by one.

```
nat() = 0:(nat()[+]repeat(1))
nat_to_pow(n) =                  //nat_to_pow(n)(i)=i^n
  if n <= 0 then repeat(1) else nat_to_pow(n-1)[*]nat()
fact() = 1:((nat()[+]repeat(1))[*]fact())
pow(n) = 1:(repeat(n)[*]pow(n)) //pow(n)(i)=n^i
fib() = 0:1:(fib()[+]fib()^)
incr(s) = s[+]repeat(1)
```

The definition of `nat` uses regular corecursion, since the recursive call `nat()` is cyclic. Hence the call `nat()` returns $(x, \{x \mapsto 0\!:\!(x[+]y), y \mapsto 1\!:\!y\})$. The definition of `nat_to_pow` is a standard inductive one where the argument strictly decreases in the recursive call. Hence, the call, e.g., `nat_to_pow(2)`, returns
$(x_2, \{x_2 \mapsto x_1[*]x, x_1 \mapsto x_0[*]x, x_0 \mapsto y, y \mapsto 1\!:\!y, x \mapsto 0\!:\!(x[+]y'), y' \mapsto 1\!:\!y'\})$.
The definitions of `fact`, `pow`, and `fib` are regularly corecursive. For instance, the call `fact()` returns $(z, z \mapsto (x[+]y)[*]z, x \mapsto 0\!:\!(x[+]y'), y \mapsto 1 : y, y' \mapsto 1 : y')$. As another example, the call `pow(2)` returns $(x, 1 : (y[*]x), y \mapsto 1 : y)$ and, by accessing the elements of the stream, we get the desired result. For instance, pow(2)(3)=2^3. The definition of `incr` is non-recursive, hence always converges, and the call `incr(s)` returns $(x, \{x \mapsto s[+]y, y \mapsto 1\!:\!y\})$. The following alternative definition

```
incr_reg(s) = (s(0)+1):incr_reg(s^)
```

relies, instead, on regular corecursion. Note the difference: the latter version can terminate only for regular streams, as in `incr_reg(one_two())`, since, eventually, in the recursive call, the expression `s^` turns out to denote the initial stream (see Chapter 14); however, the computation does not terminate

for non-regular streams, as in `incr_reg(nat())`, which, however, converges with `incr`.

The following function computes the stream of partial sums of the first $i+1$ elements of a stream s, that is, `sum`$(s)(i) = \sum_{k=0}^{i} s(k)$:

```
sum(s) = s(0):(s^[+]sum(s))
```

Such a function is useful for computing streams whose elements approximate a series with increasing precision; for instance, the following function returns the stream of partial sums of the first $i+1$ elements of the Taylor series of the exponential function:

```
sum_expn(n) = sum(pow(n)[/]fact())
```

Function `sum_expn` calls `sum` with the argument `pow(n)[/]fact()` corresponding to the stream of all terms of the Taylor series of the exponential function; hence, by accessing the i-th element of the stream, we have the following approximation of the series:

$$\texttt{sum_expn}(n)(i) = \sum_{k=0}^{i} \frac{n^k}{k!} = 1 + n + \frac{n^2}{2!} + \frac{n^3}{3!} + \frac{n^4}{4!} + \cdots + \frac{n^i}{i!}$$

Lastly, we present a couple of examples showing how it is possible to define primitive operations provided by IoT platforms for real time analysis of data streams; we start with `aggr(n,s)`, which allows aggregation (by addition) of contiguous data in the stream `s` w.r.t. a frame of length `n`:

```
aggr(n,s) = if n<=0 then repeat(0) else s[+]aggr(n-1,s^)
```

For instance, `aggr(3,`s`)` returns the stream s' s.t. $s'(i) = s(i)+s(i+1)+s(i+2)$. From `aggr`, we can easily define `avg(n,s)` to compute the stream of average values of `s` in the frame of length `n`:

```
avg(n,s) = aggr(n,s)[/]repeat(n)
```

10

Well-definedness check

The operational semantics of the stream calculus given in the previous chapter is parametric with respect to a well-definedness judgment. In this chapter, we first provide an abstract definition of well-definedness which formalizes the intuitive meaning. Then, we show that it is possibile to give an operational definition, directly leading to an algorithm, which is sound and complete with respect to the abstract definition (Theorem 10.4).

Intuitively, an environment is well-defined if each variable in its domain denotes a unique stream. We provide now a formal definition in abstract terms.

Semantically, a stream σ is an infinite sequence of numeric values, that is, a function which returns, for each index $i \geq 0$, the i-th element $\sigma(i)$. Given a result (s, ρ), we get a stream by instantiating variables in s with streams, in a way consistent with ρ, and evaluating operators. To make this formal, we need some preliminary definitions.

A *substitution* θ is a function from a finite set of variables to streams. We denote by $[\![s]\!]\theta$ the stream obtained by applying θ to s, and evaluating operators, as formally defined below.

$$[\![x]\!]\theta = \theta(x)$$

$$([\![n\!:\!s]\!]\theta)(i) = \begin{cases} n & i = 0 \\ ([\![s]\!]\theta)(i-1) & i \geq 1 \end{cases}$$

$$([\![s\hat{}\,]\!]\theta)(i) = [\![s]\!]\theta(i+1) \qquad i \geq 0$$

$$([\![s_1\ [nop]\ s_2]\!]\theta)(i) = [\![s_1]\!]\theta(i)\ nop\ [\![s_2]\!]\theta(i) \qquad i \geq 0$$

Given an environment ρ and a substitution θ with domain $vars(\rho)$, the substitution $\rho[\theta]$ is defined by:

$$\rho[\theta](x) = \begin{cases} [\![\rho(x)]\!]\theta & x \in dom(\rho) \\ \theta(x) & x \in fv(\rho) \end{cases}$$

Then, a *solution* of ρ is a substitution θ with domain $vars(\rho)$ such that $\rho[\theta] = \theta$.

A closed environment ρ is *well-defined* if it has exactly one solution, denoted $\mathsf{sol}(\rho)$. For instance, $\{x \mapsto 1\!:\!x\}$ and $\{y \mapsto 0\!:\!(y[+]x),\ x \mapsto 1 : x\}$ are well-defined, since their unique solutions map x to the infinite stream of ones, and y to the stream of natural numbers, respectively. Instead, for $\{x \mapsto 1[+]x\}$ there are no solutions. Lastly, an environment can be undetermined: for instance, any substitution mapping x into an arbitrary stream is a solution of $\{x \mapsto x\}$.

m	$::=$	$x_1 \mapsto n_1 \dots x_n \mapsto n_k \quad (n \geq 0)$	map from variables to natural numbers

$$\text{(MAIN)}\ \frac{\mathsf{wd}_{\rho'}(x, \emptyset)}{wd(\rho, x, v)}\ \rho' = \rho\{x \mapsto v\} \qquad \text{(WD-VAR)}\ \frac{\mathsf{wd}_{\rho}(\rho(x), m\{x \mapsto 0\})}{\mathsf{wd}_{\rho}(x, m)}\ x \notin dom(m)$$

$$\text{(WD-COREC)}\ \frac{}{\mathsf{wd}_{\rho}(x, m)}\ \begin{array}{l} x \in dom(m) \\ m(x) > 0 \end{array} \qquad \text{(WD-FV)}\ \frac{}{\mathsf{wd}_{\rho}(x, m)}\ x \notin dom(\rho)$$

$$\text{(WD-CONS)}\ \frac{\mathsf{wd}_{\rho}(s, m^{+1})}{\mathsf{wd}_{\rho}(n\mathbin{:}s, m)} \qquad \text{(WD-TAIL)}\ \frac{\mathsf{wd}_{\rho}(s, m^{-1})}{\mathsf{wd}_{\rho}(s\hat{}, m)} \qquad \text{(WD-NOP)}\ \frac{\mathsf{wd}_{\rho}(s_1, m) \quad \mathsf{wd}_{\rho}(s_2, m)}{\mathsf{wd}_{\rho}(s_1\ [nop]\ s_2, m)}$$

FIGURE 10.1 Operational definition of well-defined environments

An open environment ρ is well-defined if, for each θ with domain $fv(\rho)$, it has exactly one solution θ' such that $\theta \subseteq \theta'$. For instance, the open environment $\{y \mapsto 0\mathbin{:}(y[+]x)\}$ is well-defined.

We now consider the non-trivial problem of ensuring that a closed environment ρ is well-defined; if environments would be allowed to contain only the stream constructor, then it would suffice to require all non-free variables to be *guarded* by the stream constructor [14]. For instance, the environment $\{x \mapsto 1\mathbin{:}x\}$ satisfies such a syntactic condition, and is well-defined, while, in the non well-defined environment $\{x \mapsto x\}$, the variable x is not guarded by the constructor.

However, when non constructors as the tail and pointwise operators come into play, the fact that variables are guarded by the stream constructor no longer ensures that the environment is well-defined. Let us consider for instance $\rho = \{x \mapsto 0\mathbin{:}x\hat{}\}$, corresponding to the definition of the stream

```
bad_stream = 0:bad_stream^
```

In this case, ρ is not well-defined since it admits infinite solutions (all streams starting with 0), although variable x is guarded by the stream constructor.

To ensure that an environment is well-defined a more complex check is needed: in Figure 10.1 we provide an operational characterization.

The judgment $wd(\rho, x, s)$ used in the side condition of rule (INVK) holds if $\mathsf{wd}_{\rho'}(x, \emptyset)$ holds with $\rho' = \rho\{x \mapsto s\}$. The judgment $\mathsf{wd}_{\rho}(s, \emptyset)$ means that the result (s, ρ) is well-defined. That is, restricting the domain of ρ to the variables reachable from s (either occurring in s, or, transitively, in values associated with reachable variables) we get a well-defined environment; thus, $wd(\rho, x, s)$ holds if adding the association of s with x preserves the well-definedness of ρ.

The additional argument m in the judgment $\mathsf{wd}_{\rho}(s, m)$ is a map from variables to natural numbers. We write m^{+1} and m^{-1} for the maps $\{(x, m(x) + 1) \mid x \in dom(m)\}$, and $\{(x, m(x) - 1) \mid x \in dom(m)\}$, respectively.

In rule (MAIN), this map is initially empty. In rule (WD-VAR), a variable x defined in the environment is added in the map, with initial value 0, the first time it is found. In rule (WD-COREC), when it is found the second time, it is checked that more constructors than tail operators have been traversed. In

$$\dfrac{\dfrac{}{\mathsf{wd}_\rho(x,\{x\mapsto +1\})}\,(\text{WD-COREC}) \quad \dfrac{\dfrac{\dfrac{}{\mathsf{wd}_\rho(y,\{x\mapsto +1, y\mapsto +1\})}\,(\text{WD-COREC})}{\mathsf{wd}_\rho(1\colon y,\{x\mapsto +1, y\mapsto 0\})}\,(\text{WD-CONS})}{\mathsf{wd}_\rho(y,\{x\mapsto +1\})}\,(\text{WD-VAR})}{\dfrac{\dfrac{\mathsf{wd}_\rho(x\ [+]\ y,\{x\mapsto +1\})}{\mathsf{wd}_\rho(0\colon(x\ [+]\ y),\{x\mapsto 0\})}\,(\text{WD-CONS})}{\mathsf{wd}_\rho(x,\emptyset)}\,(\text{WD-VAR})}\,(\text{WD-NOP})$$

$$\dfrac{\dfrac{\dfrac{}{at_\rho(0\colon(x\ [+]\ y),0)=0}\,(\text{AT-CONS-0})}{\vdots \over at_\rho(x,i-1)=i-1}\,(\text{AT-VAR}) \quad \dfrac{\dfrac{}{at_\rho(1\colon y,0)=1}\,(\text{AT-CONS-0})}{\vdots \over at_\rho(y,i-1)=1}\,(\text{AT-VAR})}{\dfrac{\dfrac{at_\rho(x\ [+]\ y,i-1)=i}{at_\rho(0\colon(x\ [+]\ y),i)=i}\,(\text{AT-CONS-SUCC})}{at_\rho(x,i)=i}\,(\text{AT-VAR})}\,(\text{AT-OP})$$

FIGURE 10.2 Derivations of $\mathsf{wd}_\rho(x,\emptyset)$ and $at_\rho(x,i) = i$ with $\rho = (x, \{x \mapsto 0\colon(x\ [+]\ y),\ y \mapsto 1\colon y\})$

rule (WD-FV), a free variable is considered well-defined.[1] In rules (WD-CONS), (WD-TAIL), and (WD-NOP), the value associated with a variable is incremented/decremented by one each time a constructor and tail operator are traversed, respectively.

We show now some examples of derivation of well-definedness and access to the i-th element.

In Figure 10.2 we consider the result $(x, \{x \mapsto 0\colon(x\ [+]\ y),\ y \mapsto 1\colon y\})$, obtained by evaluating the call `nat()` with the program below. Its semantics is the stream of natural numbers.

```
repeat(1) = 1:repeat(1)
nat() = 0:(nat()[+]repeat(1))
```

In Figure 10.3 we consider a more tricky example, that is, the result $(x, \{x \mapsto 0\colon 1\colon(2 : x\hat{})\hat{}\})$, whose semantics is the stream $0, 1, 1, 1, \ldots$.

We show now that well-definedness of a result is a necessary and sufficient condition for termination of access to an arbitrary index. To formally express and prove this statement, we introduce some definitions and notations.

First of all, since the numeric value obtained as result is not relevant for the following technical treatment, for simplicity we will write $at_\rho(s, i)$ rather than $at_\rho(s, i) = n$. We call *derivation* an either finite or infinite proof tree.

We write $\mathsf{wd}_\rho(s', m') \vdash \mathsf{wd}_\rho(s, m)$ to mean that $\mathsf{wd}_\rho(s', m')$ is a premise of a (meta-)rule where $\mathsf{wd}_\rho(s, m)$ is the consequence, and $\vdash^\star$ for the reflexive and transitive closure of this relation. Moreover, $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s, m)$, with $x \notin \mathcal{X}$, means that in the path there can be nodes of shape $\mathsf{wd}_\rho(y, _)$ only

1 Indeed, non-well-definedness can only be detected on closed results.

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\mathsf{wd}_\rho(x, \{x \mapsto -1\})}\ (\text{WD-COREC})}{\mathsf{wd}_\rho(x\hat{}, \{x \mapsto -2\})}\ (\text{WD-TAIL})}{\mathsf{wd}_\rho(2 : x\hat{}, \{x \mapsto -1\})}\ (\text{WD-CONS})}{\mathsf{wd}_\rho((2 : x\hat{})\hat{}, \{x \mapsto -2\})}\ (\text{WD-TAIL})}{\mathsf{wd}_\rho(1 : (2 : x\hat{})\hat{}, \{x \mapsto -1\})}\ (\text{WD-CONS})}{\mathsf{wd}_\rho(0 : 1 : (2 : x\hat{})\hat{}, \{x \mapsto 0\})}\ (\text{WD-CONS})}{\mathsf{wd}_\rho(x, \emptyset)}\ (\text{WD-VAR})
$$

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{at_\rho(x, i-1) = 1}\ (\text{AT-VAR})}{at_\rho(x\hat{}, i-2) = 1}\ (\text{AT-TAIL})}{at_\rho(2 : x\hat{}, i-1) = 1}\ (\text{AT-CONS-SUCC})}{at_\rho((2 : x\hat{})\hat{}, i-2) = 1}\ (\text{AT-TAIL})}{at_\rho(1 : (2 : x\hat{})\hat{}, i-1) = 1}\ (\text{AT-CONS-SUCC})}{at_\rho(0 : 1 : (2 : x\hat{})\hat{}, i) = 1}\ (\text{AT-CONS-SUCC})}{at_\rho(x, i) = 1}\ (\text{AT-VAR})
$$

FIGURE 10.3 Derivations of $\mathsf{wd}_\rho(x, \emptyset)$ and $at_\rho(x, i) = 1$ with $\rho = \{x \mapsto 0 : 1 : (2 : x\hat{})\hat{}\}$ and $i > 1$.

for $y \in \mathcal{X}$ and non-repeated. We use analogous notations for the judgment $at_\rho(s, i)$.

LEMMA 10.1 :

1. If $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s, i)$, then $at_\rho(x, i'+k) \vdash^\star_{\mathcal{X}} at_\rho(s, i+k)$, for each $k \geq 0$.
2. A judgment $\mathsf{wd}_\rho(s, \emptyset)$ has no derivation iff the following condition holds:
   (WD-STUCK) $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}'} \mathsf{wd}_\rho(\rho(x), m\{x \mapsto 0\}) \vdash \mathsf{wd}_\rho(x, m) \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s, \emptyset)$
   for some $x \in dom(\rho)$, $\mathcal{X}', \mathcal{X}$, and m', m s.t.
   $x \notin dom(m)$, $m'(x) = k \leq 0$.
3. The derivation of $at_\rho(s, j)$ is infinite iff the following condition holds:
   (AT-∞) $at_\rho(x, i+k) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x), i) \vdash at_\rho(x, i) \vdash^\star_{\mathcal{X}} at_\rho(s, j)$
   for some $x \in dom(\rho)$, $\mathcal{X}', \mathcal{X}$, and $i, k \geq 0$.

Proof:

1. Immediate by induction on the rules.
2. For each $\mathsf{wd}_\rho(s, m)$ there is exactly one applicable rule, unless in the case $\mathsf{wd}_\rho(x, m)$ with $m(x) \leq 0$. Since $vars(s)$ is a finite set, the derivation cannot be infinite. Hence, there is no derivation for $\mathsf{wd}_\rho(s, \emptyset)$ iff there is a finite path from $\mathsf{wd}_\rho(s, \emptyset)$ of judgments on variables in $dom(\rho)$, and a (first) repeated variable, that is, of the shape below, where $x \notin dom(m)$ and $m'(x) \leq 0$.

   $$\mathsf{wd}_\rho(x, m')\mathsf{wd}_\rho(x_n, _) \ldots \mathsf{wd}_\rho(x_1, _)\mathsf{wd}_\rho(x, m)\mathsf{wd}_\rho(y_m, _) \ldots \mathsf{wd}_\rho(y_1, _) \ldots \mathsf{wd}_\rho(s, \emptyset)$$

   Hence, condition (WF-STUCK) holds.
3. For each $at_\rho(s, i)$ there is exactly one applicable rule, unless in the case $at_\rho(x, i)$ with $x \notin dom(\rho)$. Moreover, since ρ has finite domain, the derivation $at_\rho(s, j)$ is infinite iff there is an infinite path from $at_\rho(s, j)$ of judgments on variables in $dom(\rho)$, and a (first) repeated variable with a greater or equal index, hence, thanks to Lemma 10.1-(Item 1), of the

shape below, where $m, n, k \geq 0$:

$$\dots at_\rho(x_n, i_n + k)\dots at_\rho(x_1, i_1 + k) at_\rho(x, i + k) at_\rho(x_n, i_n)\dots$$

$$\dots at_\rho(x_1, i_1) at_\rho(x, i) at_\rho(y_m, j_m)\dots at_\rho(y_1, j_1)\dots at_\rho(s, j)$$

That is, condition (AT-∞) holds, with $\mathcal{X}' = \{x_1, \dots, x_n\}$, and $\mathcal{X} = \{y_1, \dots, y_m\}$.

□

LEMMA 10.2 : For $x \in dom(m)$, the following conditions are equivalent:

1. $at_\rho(x, i') \vdash^\star_\mathcal{X} at_\rho(s, i)$ for some i', i
2. $\mathsf{wd}_\rho(x, m') \vdash^\star_\mathcal{X} \mathsf{wd}_\rho(s, m)$ for some m' such that $m'(x) = m(x) + i - i'$.

Proof:

1⇒ 2 The proof is by induction on the length of the path in $at_\rho(x, i') \vdash^\star_\mathcal{X} at_\rho(s, i)$.

BASE The length of the path is 0, hence we have $at_\rho(x, i) \vdash^\star_\emptyset at_\rho(x, i)$. We also have $\mathsf{wd}_\rho(x, m) \vdash^\star_\emptyset \mathsf{wd}_\rho(x, m)$, and $m(x) = m(x) + i - i$, as requested.

INDUCTIVE STEP By cases on the rule applied to derive $at_\rho(s, i)$.

((AT-VAR)) We have $at_\rho(y, i)$, with $y \neq x$ since the length of the path is > 0, and $at_\rho(x, i') \vdash^\star_{\mathcal{X}\setminus\{y\}} at_\rho(\rho(y), i)$. Moreover, we can derive $\mathsf{wd}_\rho(y, m)$ by rule (WF-VAR), and by inductive hypothesis we also have $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}\setminus\{y\}} \mathsf{wd}_\rho(\rho(y), m\{y \mapsto 0\})$, and $m'(x) = m\{y \mapsto 0\}(x) + i - i'$, hence we get the thesis.

(AT-CONS-0) Empty case, since the derivation for $at_\rho(n : s, 0)$ does not contain a node $at_\rho(x, i')$.

(AT-CONS) We have $at_\rho(n : s, i)$, and $at_\rho(x, i') \vdash^\star_\mathcal{X} at_\rho(s, i - 1)$. Moreover, we can derive $\mathsf{wd}_\rho(n : s, m)$ by rule (WF-CONS), and by inductive hypothesis we also have $\mathsf{wd}_\rho(x, m') \vdash^\star_\mathcal{X} \mathsf{wd}_\rho(s, m^{+1})$, with $m'(x) = m^{+1}(x) + (i - 1) - i'$, hence we get the thesis.

(AT-TAIL) This case is symmetric to the previous one.

(AT-NOP) We have $at_\rho(s_1 \; [nop] \; s_2, i)$, and either $at_\rho(x, i') \vdash^\star_\mathcal{X} at_\rho(s_1, i)$, or $at_\rho(x, i') \vdash^\star_\mathcal{X} at_\rho(s_2, i)$. Assume the first case holds, the other is analogous. Moreover, we can derive $\mathsf{wd}_\rho(s_1 \; [nop] \; s_2, m)$ by rule (WD-NOP), and by inductive hypothesis we also have $\mathsf{wd}_\rho(x, m') \vdash^\star_\mathcal{X} \mathsf{wd}_\rho(s_1, m)$, with $m'(x) = m(x) + i - i'$, hence we get the thesis.

2⇒ 1 The proof is by induction on the length of the path in $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s, m)$.

BASE The length of the path is 0, hence we have $\mathsf{wd}_\rho(x, m) \vdash^\star_\emptyset \mathsf{wd}_\rho(x, m)$. We also have, for an arbitrary i, $at_\rho(x, i) \vdash^\star_\emptyset at_\rho(x, i)$, and $m(x) = m(x) + i - i$, as requested.

INDUCTIVE STEP By cases on the rule applied to derive $\mathsf{wd}_\rho(s, m)$.

(WD-VAR) We have $\mathsf{wd}_\rho(y, m)$, with $y \notin dom(m)$, $y \neq x$ since $x \in dom(m)$, and $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}\setminus\{y\}} \mathsf{wd}_\rho(\rho(y), m\{y \mapsto 0\})$. By inductive hypothesis we have $at_\rho(x, i') \vdash^\star_{\mathcal{X}\setminus\{y\}} at_\rho(\rho(y), i)$ for some i', i such that $m'(x) = m(x) + i - i'$. Moreover, since $y \in dom(\rho)$, $at_\rho(\rho(y), i) \vdash at_\rho(y, i)$ by rule (AT-VAR), hence we get $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(y, i)$.

(WD-COREC) Empty case, since the derivation for $\mathsf{wd}_\rho(y, m)$ would not contain a node $\mathsf{wd}_\rho(x, m)$.

(WD-FV) Empty case, since the derivation for $\mathsf{wd}_\rho(y, m)$ would not contain a node $\mathsf{wd}_\rho(x, m)$.

(WD-CONS) We have $\mathsf{wd}_\rho(n : s, m)$, and $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s, m^{+1})$. By inductive hypothesis we have $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s, i)$ for some i', i such that $m'(x) = m^{+1}(x) + i - i'$. Moreover, $at_\rho(s, i) \vdash at_\rho(n : s, i{+}1)$ by rule (AT-CONS-SUCC), hence we get $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(n : s, i + 1)$ with $m'(x) = m(x) + i + 1 - i'$, as requested.

(WD-TAIL) We have $\mathsf{wd}_\rho(s\hat{\ }, m)$, and $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s, m^{-1})$. By inductive hypothesis we have $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s, i)$ for some i', i such that $m'(x) = m^{-1}(x) + i - i'$. We can assume $i > 0$ thanks to Lemma 10.1-(Item 1). Hence, $at_\rho(s, i) \vdash at_\rho(n : s, i - 1)$ by rule (AT-TAIL), hence we get $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s\hat{\ }, i - 1)$ with $m'(x) = m(x) + i - 1 - i'$, as requested.

(WD-NOP) We have $\mathsf{wd}_\rho(s_1 [nop] s_2, m)$, and either $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s_1, m)$, or $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s_2, m)$. Assume the first case holds, the other is analogous. By inductive hypothesis we have $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s_1, i)$, for some i', i such that $m'(x) = m(x) + i - i'$. Moreover, we can derive $at_\rho(s_1\ [nop]\ s_2, i)$ by rule (AT-NOP), hence we get the thesis.

□

LEMMA 10.3 : For $x \notin dom(m)$, the following conditions are equivalent:

1. $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s, i)$ for some i', i
2. $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s, m)$ for some m' such that $x \notin dom(m')$.

Proof: Easy variant of the proof of Lemma 10.2. □

THEOREM 10.4 : $\mathsf{wd}_\rho(s, \emptyset)$ is derivable iff, for all j, $at_\rho(s, j)$ either has no derivation or a finite derivation.

Proof: We prove that $at_\rho(s, j)$ has an infinite derivation for some j iff $\mathsf{wd}_\rho(s, \emptyset)$ has no derivation.

$\Rightarrow$ Lemma 10.1-(Item 3), we have that the following condition holds:

(AT-∞) $at_\rho(x, i+k) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x), i) \vdash at_\rho(x, i) \vdash^\star_{\mathcal{X}} at_\rho(s, j)$
for some $x \in dom(\rho)$, $\mathcal{X}'$, $\mathcal{X}$, and $i, k \geq 0$.

Then, starting from the right, Lemma 10.3 we have $\mathsf{wd}_\rho(x, m) \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s, \emptyset)$ for some m such that $x \notin dom(m)$; by rule (WD-VAR) we have $\mathsf{wd}_\rho(\rho(x), m\{x \mapsto 0\}) \vdash \mathsf{wd}_\rho(x, m)$, and finally by Lemma 10.2 we have:

(WD-STUCK) $\mathsf{wd}_\rho(x, m') \vdash^\star_{\mathcal{X}'} \mathsf{wd}_\rho(\rho(x), m\{x \mapsto 0\}) \vdash \mathsf{wd}_\rho(x, m) \vdash^\star_{\mathcal{X}} \mathsf{wd}_\rho(s, \emptyset)$
for some $x \in dom(\rho)$, $\mathcal{X}'$, $\mathcal{X}$, and m', m s.t.
$x \notin dom(m)$, $m'(x) = k \leq 0$.

hence we get the thesis.

$\Leftarrow$ By Lemma 10.1-(Item 2) ,we have that the condition (WD-STUCK) above holds. Then, starting from the left, Lemma 10.2 we have $at_\rho(x, i') \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x), i)$ for some i', i such that $i - i' = k \leq 0$; by rule (AT-VAR) we have $at_\rho(\rho(x), i) \vdash at_\rho(x, i)$, and by Lemma 10.3 we have $at_\rho(x, j') \vdash^\star_{\mathcal{X}} at_\rho(s, j)$ for some j', j. If $i = j' + h$, $h \geq 0$, then by Lemma 10.1-(1) we have

$$at_\rho(x, i+k) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x), i) \vdash at_\rho(x, i) \vdash^\star_{\mathcal{X}} at_\rho(s, j+h)$$

If $j' = i + h$, $h \geq 0$, then by Lemma 10.1-(Item 1) we have

$$at_\rho(x, i+k+h) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x), i) \vdash at_\rho(x, i+h) \vdash^\star_{\mathcal{X}} at_\rho(s, j).$$

In both cases, the derivation of $at_\rho(s, j)$ is infinite.

□

11

Expressive power

In this chapter we study the expressive power of the calculus introduced so far. We first show that the calculus is more expressive than polynomials, then consider several operators for combining two streams which can be derived from those of the calculus, and finally prove that interleaving cannot be expressed in the calculus.

11.1 Streams and polynomials

A polynomial $P(x)$ over the real numbers naturally represents the stream of numbers σ_P s.t. $\sigma_P(i) = P(i)$, for each index $i \geq 0$, where $P(i)$ denotes the evaluation of P at i.

In the calculus it is possible to encode σ_P for any polynomial $P(x)$, by using the primitive pointwise operators `[+]` and `[*]` and the auxiliary functions `repeat`, `nat` and `nat_to_pow` whose definitions, copied below[1] for convenience, have been first introduced in Section 9.2.

```
repeat(n) = n:repeat(n)
nat() = 0:(nat()[+][1])
nat_to_pow(n) = if n <= 0 then [1] else nat_to_pow(n-1)[*]nat()
```

The encoding $enc(P(x))$ can be defined by induction on the degree of the polynomial $P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$:

$enc(a_0) =$`[`a_0`]`
$enc(a_n x^n + P'(x)) =$`([`a_n`][*]nat_to_pow(`n`))[+]`$enc(P(x))$ `with` $n > 0$`,`
`degree of` $P'(x) < n$

By applying the semantic rules in Figure 9.2 one can prove by induction on the index i that for all natural numbers n `repeat(`n`)(`i`)`, `nat()(`i`)` and `nat_to_pow(`n`)(`i`)` evaluate[2] to n, i and n^i, respectively. By using these facts the correctness of the encoding shown above, that is, $enc(P(x))(i)$ evaluates to $P(i)$ for all indexes i, can be proved by induction on the degree of $P(x)$ an by applying the semantic rules.

Besides the result above, one can show that the expressive power of the calculus goes beyond polynomial functions; indeed, the stream σ s.t. $\sigma(0) = 1$ and $\sigma(i) = 0$ for all $i > 0$, defined by `1:[0]`, cannot be expressed by a polynomial, by virtue of the fundamental theorem of algebra stating that every

1 We recall that `[`n`]` is an abbreviation for `repeat(`n`)`.
2 By assuming that 0^0 denotes the agreed value 1.

non-zero polynomial over the real numbers has a finite number of roots. Besides this very simple example, the calculus is able to express more interesting forms of streams which are not definable with polynomial functions. As a first example, by using the pointwise operator [/] and by virtue of the encoding of polynomials shown above, it is possible to define streams expressible with *rational functions*, that is, functions of the form $\frac{p(x)}{q(x)}$, where $p(x)$ and $q(x)$ are polynomials and $q(x) \neq 0$.

Finally, Chapter 9 contains other examples of streams not expressible with polynomials; for instance `fact()`, `pow(`n`)`, for $n > 1$, and `fib()` cannot be expressed by any polynomial because their elements grow exponentially with the index i.

11.2 Other binary operators on streams

The calculus provides only four primitive binary operators for combining streams, corresponding to the arithmetic operations applied pointwise; however, other combinators can be derived from them. Consider, for instance, the following definition:

```
combine(mask,s1,s2)=(mask[*]s1)[+](([1][-]mask)[*]s2)
```

Function `combine` defines a generic operator parametric in a stream of bits `mask` which defines for each index i which element between `s1(`i`)` and `s2(`i`)` has to be considered when combining the two streams `s1` and `s2`: the computed stream returns `s1(`i`)` on index i if `mask(`i`)=1`, `s2(`i`)` otherwise. As an example of use of `combine`, it is possible to define the following `swap` functions swapping elements of a stream at even and odd indexes:

```
alt_mask=1:0:alt_mask
swap(s)=combine(alt_mask,s^,0:s)
```

The tail operator in `s^` allows left shifting of one position of the elements of `s`; analogously, the stream constructor in `0:s` (the value of the first element is actually irrelevant for the result, since it will be discarded by the `combine` function) performs right shifting. Finally, `combine` with `alt_mask` keeps the elements of `s^` and `0:s` with even and odd index, respectively; therefore, for all $i \geq 0$, `swap(s)(`$2i$`)=s(`$2i+1$`)`, while `swap(s)(`$2i+1$`)=s(`$2i$`)`.

ZIP Zip is a common binary operator used to combine streams: σ_1 *zip* σ_2 returns the stream σ s.t. $\sigma(i) = (\sigma_1(i), \sigma_2(i))$ for all indexes i. Such an operator cannot be encoded in the calculus because, for simplicity, only streams of numbers are supported; anyway, we see no technical difficulties in extending the calculus with streams of pairs and the *zip* operator which applies the pair constructor pointwise to the elements of the streams, similarly as happens for the arithmetic pointwise operators; the rules for the operational semantics and the check for well-defined streams would mimic the corresponding rules (NOP) and (WD-NOP).

INTERLEAVING Another useful operator to collect data from more streams is interleaving (also known as *zip*); here, for simplicity, we consider only its binary version: the interleaving $\sigma_1 \parallel \sigma_2$ of two streams σ_1, σ_2 is the stream σ s.t. $\sigma(2i) = \sigma_1(i)$ and $\sigma(2i+1) = \sigma_2(i)$ for all indexes i. In the rest of this section we show that such an operator is not expressible in the calculus by studding the properties of operators on regular streams and their periods. To this aim we introduce the notion $\sigma\hat{}n$, with n natural number, for generalized tail: for all indexes i, $\sigma\hat{}n(i) = \sigma(i+n)$; the property $(\sigma\hat{}n)\hat{}m = \sigma\hat{}m{+}n = \sigma\hat{}n{+}m = (\sigma\hat{}m)\hat{}n$ follows immediately from the definition.

We say that a stream σ is *regular* iff there exist $q \geq 0$, $n > 0$ s.t. $\sigma\hat{}q{+}n = \sigma\hat{}q$. The *period* p of a regular stream is defined by $p = \min\{n > 0 \mid \exists q \geq 0\ \sigma\hat{}q{+}n = \sigma\hat{}q\}$; this is always well-defined since we are considering a total order and the set $\{n > 0 \mid \exists q \geq 0\ \sigma\hat{}q{+}n = \sigma\hat{}q\}$ is not empty by definition of regular stream.

REMARK The definition above captures the intuition that a regular stream "becomes" cyclic after an initial number $q \geq 0$ of elements, and that its period p corresponds to the length of its smallest cycle; for this reason, a regular stream can always be defined by a finite set of equations involving only the stream constructor [15]. The properties that follow concern periods of regular streams, and do not depend on the length q of the initial non-periodic part of a stream; this explains the existential quantification of q in the definition of period, although there is always a least q which correctly individuates where a stream σ "becomes" periodic, that is, $\sigma\hat{}q$ is periodic, and, thus, there exists $n > 0$ s.t. $(\sigma\hat{}q)\hat{}n = \sigma\hat{}q{+}n = \sigma\hat{}q$. Clearly, if q is the length of the non-periodic part, then for all $k \geq 0$, $\sigma\hat{}q{+}k$ is still regular with the same period p by virtue of the identity $\sigma\hat{}q{+}k{+}n = (\sigma\hat{}q{+}n)\hat{}k = (\sigma\hat{}q)\hat{}k = \sigma\hat{}q{+}k$.

The following propositions show that if a regular stream has period p, then for all $n > 0$ the following fact holds: $\exists q\ \sigma\hat{}q{+}n = \sigma\hat{}q$ if and only if there exists $k > 0$ s.t. $n = k \cdot p$. This corresponds to the intuition that if σ is regular with period p, then all cycles in σ have length $k \cdot p$ for $k > 0$.

PROPOSITION 11.1 : If there exist $q \geq 0$, $n > 0$ s.t. $\sigma\hat{}q{+}n = \sigma\hat{}q$, then $\sigma\hat{}q{+}k{\cdot}n = \sigma\hat{}q$ for all $k > 0$.

Proof: The proof proceeds by induction on k.

BASIS: if $k = 1$, then the claim reduces to the hypothesis $\sigma\hat{}q{+}n = \sigma\hat{}q$;

STEP: if $k > 1$ then by inductive hypothesis $\sigma\hat{}q{+}(k-1){\cdot}n = \sigma\hat{}q$, hence $\sigma\hat{}q{+}k{\cdot}n = \sigma\hat{}q{+}(k-1){\cdot}n{+}n = \sigma\hat{}q{+}n = \sigma\hat{}q$.

□

LEMMA 11.2 : If $\sigma\hat{}q{+}n = \sigma\hat{}q$ and $\sigma\hat{}q{+}m = \sigma\hat{}q$, with $q \geq 0$, $m \geq n > 0$, then $\sigma\hat{}q{+}m{-}n = \sigma\hat{}q$.

Proof: If $\sigma\hat{}q{+}n = \sigma\hat{}q$, then $\sigma\hat{}q{+}n{+}m{-}n = \sigma\hat{}q{+}m{-}n$, hence $\sigma\hat{}q{+}m{-}n = \sigma\hat{}q{+}m = \sigma\hat{}q$. □

PROPOSITION 11.3 : The following claim holds for all $n > 0$: if there exists $q \geq 0$ s.t. $\sigma\hat{}^{q+n} = \sigma\hat{}^{q}$, then there exists $k > 0$ s.t. $n = k \cdot p$, with p the period of σ.

Proof: By virtue of the hypothesis $\sigma\hat{}^{q+n} = \sigma\hat{}^{q}$, σ is regular, hence its period p is well-defined; furthermore, by definition, $p \leq n$. If we divide n by p, then, by the properties of the integer division, there exist $k > 0$ and $0 \leq m < p$ s.t. $n = k \cdot p + m$. Then the proof proceeds by proving by induction on k the following claim: for all $k > 0$, if there exists $q \geq 0$ s.t. $\sigma\hat{}^{q+k \cdot p+m} = \sigma\hat{}^{q}$, with $0 \leq m < p$, and p period of σ, then $m = 0$.

BASIS: if $k = 1$, then $\sigma\hat{}^{q+p+m} = \sigma\hat{}^{q}$; furthermore, there exists $r \geq 0$ s.t. $\sigma\hat{}^{r} = \sigma\hat{}^{r+p}$ by definition of period, hence $\sigma\hat{}^{q+r+p+m} = \sigma\hat{}^{q+r} = \sigma\hat{}^{q+r+p}$ and, by definition of period, if $m > 0$ then $p \leq m$, therefore it must be $m = 0$ since $m < p$ by hypothesis;

STEP: if $k > 1$ then $k \cdot p + m \geq p$; by definition of period there exists r s.t. $\sigma\hat{}^{r+p} = \sigma\hat{}^{r}$, hence $\sigma\hat{}^{q+r+p} = \sigma\hat{}^{q+r}$; by hypothesis $\sigma\hat{}^{q+k \cdot p+m} = \sigma\hat{}^{q}$, hence $\sigma\hat{}^{q+r+k \cdot p+m} = \sigma\hat{}^{q+r}$, then by Lemma 11.2, $\sigma\hat{}^{q+r+k \cdot p+m-p} = \sigma\hat{}^{q+r}$, therefore $\sigma\hat{}^{q+r+(k-1) \cdot p+m} = \sigma\hat{}^{q+r}$ and $m = 0$ can be concluded by inductive hypothesis.

□

The next propositions show that the operators of the calculus preserve regular streams and that there exist specific relations between the periods of the operands and of the results of the operations. In the rest of the section we adopt the same syntax of the operators of the calculus for their semantic interpretation.

PROPOSITION 11.4 : The stream constructor and the tail operator preserve regular streams and their period.

Proof: The proof proceeds by cases on the applied operator.

constructor: let σ_1 be regular and $\sigma_2 = m : \sigma_1$ for some number m. For all $q \geq 0, n > 0$ if ${\sigma_1}\hat{}^{q+n} = {\sigma_1}\hat{}^{q}$, then the identities ${\sigma_2}\hat{}^{q+1+n} = {\sigma_1}\hat{}^{q+n} = {\sigma_1}\hat{}^{q} = {\sigma_2}\hat{}^{q+1}$ hold. Symmetrically, if ${\sigma_2}\hat{}^{q+n} = {\sigma_2}\hat{}^{q}$, then the identities ${\sigma_1}\hat{}^{q+n} = {\sigma_2}\hat{}^{q+1+n} = {\sigma_2}\hat{}^{q+1} = {\sigma_1}\hat{}^{q}$ hold. Therefore $\{n > 0 \mid \exists q \geq 0\ {\sigma_1}\hat{}^{q+n} = {\sigma_1}\hat{}^{q}\} = \{n > 0 \mid \exists q \geq 0\ {\sigma_2}\hat{}^{q+n} = {\sigma_2}\hat{}^{q}\}$, hence σ_2 is regular and has the same period as σ_1.

tail: let σ_1 be regular and $\sigma_2 = \sigma_1\hat{}$. For all $q \geq 0, n > 0$ if ${\sigma_1}\hat{}^{q+n} = {\sigma_1}\hat{}^{q}$, then the identities ${\sigma_2}\hat{}^{q+n} = {\sigma_1}\hat{}^{q+1+n} = {\sigma_1}\hat{}^{q+1} = {\sigma_2}\hat{}^{q}$ hold. Symmetrically, if ${\sigma_2}\hat{}^{q+n} = {\sigma_2}\hat{}^{q}$, then the identities ${\sigma_1}\hat{}^{q+1+n} = {\sigma_2}\hat{}^{q+n} = {\sigma_2}\hat{}^{q} = {\sigma_1}\hat{}^{q+1}$ hold. Therefore $\{n > 0 \mid \exists q \geq 0\ {\sigma_1}\hat{}^{q+n} = {\sigma_1}\hat{}^{q}\} = \{n > 0 \mid \exists q \geq 0\ {\sigma_2}\hat{}^{q+n} = {\sigma_2}\hat{}^{q}\}$, hence σ_2 is regular and has the same period as σ_1.

□

Given two integers m and n, we denote with $lcd(m, n)$ the *least common multiple* of m and n, that is, the smallest positive integer s.t. that m and n are divisors of k; therefore, if m and n are divisors of $k > 0$, then $lcd(m, n) \leq k$.

PROPOSITION 11.5 : The binary pointwise operators preserve regular streams. In particular, if the period of σ_1 and σ_2 is p_1 and p_2, respectively, then $\sigma_1 [nop] \sigma_2$ has period p which is a divisor of $lcd(p_1, p_2)$, that is, there exists $k > 0$ s.t. $k \cdot p = lcd(p_1, p_2)$.

Proof: Let σ_1 and σ_2 be regular streams with period p_1 and p_2, respectively, with $lcd(p_1, p_2) = n$; by definition, $p_1, p_2 > 0$, hence $n > 0$. Let k_1 and k_2 be the positive integers s.t. $n = k_i \cdot p_i$ for $i = 1, 2$; by definition of period and Proposition 11.1, there exist $q_1, q_2 \geq 0$ s.t. $\sigma_i{}^{\wedge q_i + k_i \cdot p_i} = \sigma_i{}^{\wedge q_i}$, for $i = 1, 2$, hence $\sigma_i{}^{\wedge q_1+q_2+n} = \sigma_i{}^{\wedge q_1+q_2}$, for $i = 1, 2$. By definition of the binary pointwise operators, $(\sigma_1 [nop] \sigma_2)^{\wedge q_1+q_2+n} = \sigma_1{}^{\wedge q_1+q_2+n} [nop] \sigma_2{}^{\wedge q_1+q_2+n} = \sigma_1{}^{\wedge q_1+q_2} [nop] \sigma_2{}^{\wedge q_1+q_2} = (\sigma_1 [nop] \sigma_2)^{\wedge q_1+q_2}$. This shows that $\sigma_1 [nop] \sigma_2$ is regular; furthermore, since $n > 0$, by Proposition 11.3 there exists $k > 0$ s.t. $k \cdot p = n = lcd(p_1, p_2)$. □

REMARK There exist cases where the period of $\sigma_1 [nop] \sigma_2$ is strictly less than $lcd(p_1, p_2)$, with p_1 and p_2 periods of σ_1 and σ_2, respectively. For instance, let σ_1 and σ_2 be s.t. for all $i \geq 0$, $\sigma_1(2i) = \sigma_2(2i + 1) = m$, $\sigma_1(2i + 1) = \sigma_2(2i) = n$, with $n \neq m$; then, both σ_1 and σ_2 have period 2, while $\sigma_1 [+] \sigma_2$ has period 1, because $(\sigma_1 [+] \sigma_2)(i) = n + m$ for all $i \geq 0$. Analogous examples can be found for the other arithmetic operators.

We finally prove that interleaving preserves regular streams as well, but, except for some particular cases, the period of the result is equal to $2lcd(p_1, p_2)$, with p_1 and p_2 periods of the operands; this different pattern is at the basis of the proof below which shows that interleaving is not expressible in the calculus. Here and in the rest of the paper we use the symbol $\|$ to denote interleaving both at the semantics and syntax level.

LEMMA 11.6 : For all $n \geq 0$, $(\sigma_1 \| \sigma_2)^{\wedge 2n} = \sigma_1{}^{\wedge n} \| \sigma_2{}^{\wedge n}$ and $(\sigma_1 \| \sigma_2)^{\wedge 2n+1} = \sigma_2{}^{\wedge n} \| \sigma_1{}^{\wedge n+1}$.

Proof: For all $i \geq 0$, the following identities hold by definition of generalized tail and interleaving:

- $(\sigma_1 \| \sigma_2)^{\wedge 2n}(2i) = (\sigma_1 \| \sigma_2)(2 \cdot (i + n)) = \sigma_1(i + n) = \sigma_1{}^{\wedge n}(i) = (\sigma_1{}^{\wedge n} \| \sigma_2{}^{\wedge n})(2i)$
- $(\sigma_1 \| \sigma_2)^{\wedge 2n}(2i + 1) = (\sigma_1 \| \sigma_2)(2 \cdot (i + n) + 1) = \sigma_2(i + n) = \sigma_2{}^{\wedge n}(i) = (\sigma_1{}^{\wedge n} \| \sigma_2{}^{\wedge n})(2i + 1)$
- $(\sigma_1 \| \sigma_2)^{\wedge 2n+1}(2i) = (\sigma_1 \| \sigma_2)(2 \cdot (i + n) + 1) = \sigma_2(i + n) = \sigma_2{}^{\wedge n}(i) = (\sigma_2{}^{\wedge n} \| \sigma_1{}^{\wedge n+1})(2i)$
- $(\sigma_1 \| \sigma_2)^{\wedge 2n+1}(2i+1) = (\sigma_1 \| \sigma_2)(2 \cdot (i+n+1)) = \sigma_1(i+n+1) = \sigma_1{}^{\wedge n+1}(i) = (\sigma_2{}^{\wedge n} \| \sigma_1{}^{\wedge n+1})(2i + 1)$

□

LEMMA 11.7 : If $\sigma_1 \parallel \sigma_2$ is regular with period $p = 2n+1$ for some $n \geq 0$, then there exists $q \geq 0$ s.t. $\sigma_2\hat{}^{q+n} = \sigma_1\hat{}^{q}$ and $\sigma_1\hat{}^{q+n+1} = \sigma_2\hat{}^{q}$.

Proof: By definition of period, there exists $q \geq 0$ s.t. $(\sigma_1 \parallel \sigma_2)\hat{}^{q+p} = (\sigma_1 \parallel \sigma_2)\hat{}^{q}$, hence $(\sigma_1 \parallel \sigma_2)\hat{}^{2q+p} = (\sigma_1 \parallel \sigma_2)\hat{}^{2q}$; by the previous identity and Lemma 11.6 $\sigma_2\hat{}^{q+n} \parallel \sigma_1\hat{}^{q+n+1} = (\sigma_1 \parallel \sigma_2)\hat{}^{2q+2n+1} = (\sigma_1 \parallel \sigma_2)\hat{}^{2q} = \sigma_1\hat{}^{q} \parallel \sigma_2\hat{}^{q}$. Hence, for all $i \geq 0$

- $\sigma_2\hat{}^{q+n}(i) = (\sigma_2\hat{}^{q+n} \parallel \sigma_1\hat{}^{q+n+1})(2i) = (\sigma_1\hat{}^{q} \parallel \sigma_2\hat{}^{q})(2i) = \sigma_1\hat{}^{q}(i)$
- $\sigma_1\hat{}^{q+n+1}(i) = (\sigma_2\hat{}^{q+n} \parallel \sigma_1\hat{}^{q+n+1})(2i+1) = (\sigma_1\hat{}^{q} \parallel \sigma_2\hat{}^{q})(2i+1) = \sigma_2\hat{}^{q}(i)$

□

LEMMA 11.8 : Interleaving preserves regular streams and the period of $\sigma_1 \parallel \sigma_2$ is a divisor of $2lcd(p_1, p_2)$, where p_1 and p_2 are the periods of σ_1 and σ_2, respectively.

Proof: Let $lcd(p_1, p_2) = n$; by definition, $p_1, p_2 > 0$, hence $n > 0$. Let k_1 and k_2 be the positive integers s.t. $n = k_i \cdot p_i$ for $i = 1, 2$; by definition of period and Proposition 11.1, there exist $q_1, q_2 \geq 0$ s.t. $\sigma_i\hat{}^{q_i+k_i\cdot p_i} = \sigma_i\hat{}^{q_i}$, for $i = 1, 2$, hence $\sigma_i\hat{}^{q_1+q_2+n} = \sigma_i\hat{}^{q_1+q_2}$, for $i = 1, 2$. By the previous identities and Lemma 11.6, $(\sigma_1 \parallel \sigma_2)\hat{}^{2(q_1+q_2+n)} = \sigma_1\hat{}^{q_1+q_2+n} \parallel \sigma_2\hat{}^{q_1+q_2+n} = \sigma_1\hat{}^{q_1+q_2} \parallel \sigma_2\hat{}^{q_1+q_2} = (\sigma_1 \parallel \sigma_2)\hat{}^{2(q_1+q_2)}$. Therefore $\sigma_1 \parallel \sigma_2$ is regular and, by Proposition 11.3, its period is a divisor of $2n = 2lcd(p_1, p_2)$.

□

LEMMA 11.9 : If there exist $n, q \geq 0$ s.t.

(1) $\sigma_2\hat{}^{q+n} = \sigma_1\hat{}^{q}$ (2) $\sigma_1\hat{}^{q+n+1} = \sigma_2\hat{}^{q}$

then σ_1, σ_2 and $\sigma_1 \parallel \sigma_2$ are all regular with the same period.

Proof: We first show that σ_1 and σ_2 are regular. From (1) we derive $\sigma_2\hat{}^{q+n+n+1} = \sigma_1\hat{}^{q+n+1}$, hence from (2) $\sigma_2\hat{}^{q+2n+1} = \sigma_2\hat{}^{q}$; similarly, from (2) $\sigma_1\hat{}^{q+n+1+n} = \sigma_2\hat{}^{q+n}$, hence from (1) $\sigma_1\hat{}^{q+2n+1} = \sigma_1\hat{}^{q}$, therefore both σ_1 and σ_2 are regular, and by Proposition 11.3, their periods p_1 and p_2 are divisors of $2n+1$, and, hence, are both odd.

We now show that $p_1 = p_2$; let $q_1 \geq 0$, $n_1 > 0$ s.t. $\sigma_1\hat{}^{q_1+n_1} = \sigma_1\hat{}^{q_1}$, hence $\sigma_1\hat{}^{q_1+q+n_1} = \sigma_1\hat{}^{q_1+q}$; from the previous identity and (1) $\sigma_2\hat{}^{q_1+q+n+n_1} = \sigma_1\hat{}^{q_1+q+n_1} = \sigma_1\hat{}^{q_1+q} = \sigma_2\hat{}^{q_1+q+n}$. Analogously, let $q_2 \geq 0$, $n_2 > 0$ s.t. $\sigma_2\hat{}^{q_2+n_2} = \sigma_2\hat{}^{q_2}$, hence $\sigma_2\hat{}^{q_2+q+n_2+2n} = \sigma_2\hat{}^{q_2+q+2n}$; from the previous identity and (1) $\sigma_1\hat{}^{q_2+q+n+n_2} = \sigma_2\hat{}^{q_2+q+2n+n_2} = \sigma_2\hat{}^{q_2+q+2n} = \sigma_1\hat{}^{q_2+q+n}$. Therefore, $\{n > 0 \mid \exists q \geq 0\ \sigma_1\hat{}^{q+n} = \sigma_1\hat{}^{q}\} = \{n > 0 \mid \exists q \geq 0\ \sigma_1\hat{}^{q+n} = \sigma_2\hat{}^{q}\}$, therefore $p_1 = p_2$, by definition.

We know determine the period p of $\sigma_1 \parallel \sigma_2$. By identities (1) and (2) and Lemma 11.6 $(\sigma_1 \parallel \sigma_2)\hat{}^{2q+2n+1} = \sigma_2\hat{}^{q+n} \parallel \sigma_1\hat{}^{q+n+1} = \sigma_1\hat{}^{q} \parallel \sigma_2\hat{}^{q} = (\sigma_1 \parallel \sigma_2)\hat{}^{2q}$,

therefore by Proposition 11.3 p is a divisor of $2n + 1$, hence p is odd. Since $lcd(p_1, p_2) = p_1 = p_2$, by Lemma 11.8, the period p of $\sigma_1 \parallel \sigma_2$ is a divisor of $2p_1 = 2p_2$, hence p is a divisor of $p_1 = p_2$, since p is odd. Furthermore, there exists $m \geq 0$ s.t. $p = 2m + 1$ and, by Lemma 11.7, there exists $r \geq 0$ s.t. ${\sigma_2}^{\hat{}r+m} = {\sigma_1}^{\hat{}r}$ and ${\sigma_1}^{\hat{}r+m+1} = {\sigma_2}^{\hat{}r}$, hence, as already proved above, $p_1 = p_2$ is a divisor of $p = 2m + 1$. Since $p_1 = p_2$ is a divisor of p and p is a divisor of $p_1 = p_2$, we conclude $p_1 = p_2 = p$. □

REMARK Lemma 11.9 characterizes all those cases where the period of $\sigma_1 \parallel \sigma_2$ is not $2lcd(p_1, p_2)$, with p_i period of σ_i for $i = 1, 2$. In fact, there are regular streams satisfying identities (1) and (2) of the lemma for all odd periods; indeed, for any regular stream σ with arbitrary period $p = 2n + 1$, $n \geq 0$, identities (1) and (2) are verified for $\sigma_1 = \sigma$ and $\sigma_2 = \sigma^{\hat{}n+1}$. Examples for $p = 1$ are streams σ_1, σ_2 s.t. $\sigma_1 = \sigma_2 = m{:}\sigma_1$ for all numbers m; more involved examples can be found for $p > 1$. For instance, $\sigma_1 = 1{:}2{:}3{:}\sigma_1$ and $\sigma_2 = 3{:}1{:}2{:}\sigma_2$ if one considers $p = 3$.

The following proposition fully characterizes the periods of interleaving of regular streams.

PROPOSITION 11.10 : Let σ_1, σ_2 be regular with periods p_1, p_2, respectively; if there exist $n, q \geq 0$ s.t. ${\sigma_2}^{\hat{}q+n} = {\sigma_1}^{\hat{}q}$ and ${\sigma_1}^{\hat{}q+n+1} = {\sigma_2}^{\hat{}q}$, then $\sigma_1 \parallel \sigma_2$ has period $p = p_1 = p_2$, otherwise it has period $p = 2lcd(p_1, p_2)$.

Proof: If ${\sigma_2}^{\hat{}q+n} = {\sigma_1}^{\hat{}q}$ and ${\sigma_1}^{\hat{}q+n+1} = {\sigma_2}^{\hat{}q}$ for some $q, n \geq 0$, then we conclude by Lemma 11.9 $p = p_1 = p_2$; otherwise, p must be even by Lemma 11.7, hence there exists $m > 0$ s.t. $p = 2m$. By definition of period, there exists $q \geq 0$ s.t. $(\sigma_1 \parallel \sigma_2)^{\hat{}q+2m} = (\sigma_1 \parallel \sigma_2)^{\hat{}q}$, hence, $(\sigma_1 \parallel \sigma_2)^{\hat{}2q+2m} = (\sigma_1 \parallel \sigma_2)^{\hat{}2q}$. Therefore, by definition of interleaving and Lemma 11.6 the following identities hold for all $i \geq 0$:

- ${\sigma_1}^{\hat{}q+m}(i) = ({\sigma_1}^{\hat{}q+m} \| {\sigma_2}^{\hat{}q+m})(2i) = (\sigma_1 \parallel \sigma_2)^{\hat{}2q+2m}(2i) = (\sigma_1 \parallel \sigma_2)^{\hat{}2q}(2i) = ({\sigma_1}^{\hat{}q} \parallel {\sigma_2}^{\hat{}q})(2i) = {\sigma_1}^{\hat{}q}(i)$
- ${\sigma_2}^{\hat{}q+m}(i) = ({\sigma_1}^{\hat{}q+m} \parallel {\sigma_2}^{\hat{}q+m})(2i + 1) = (\sigma_1 \parallel \sigma_2)^{\hat{}2q+2m}(2i + 1) = (\sigma_1 \parallel \sigma_2)^{\hat{}2q}(2i + 1) = ({\sigma_1}^{\hat{}q} \parallel {\sigma_2}^{\hat{}q})(2i + 1) = {\sigma_2}^{\hat{}q}(i)$

By the identities above and Proposition 11.3, the periods p_1 and p_2 of σ_1 and σ_2 are divisors of $m > 0$, hence $lcd(p_1, p_2) \leq m$, $2lcd(p_1, p_2) \leq 2m = p$. By Lemma 11.8 p is a divisor of $2lcd(p_1, p_2)$, hence, $p \leq 2lcd(p_1, p_2)$, because, by definition, $lcd(p_1, p_2) > 0$; hence we conclude $p = 2lcd(p_1, p_2)$. □

We can now prove the main result of this section.

Let $vars(s)$ be the set of variables contained in the stream value s; if $x \in vars(s)$, we say that s depends on x iff for all streams σ_1, σ_2 and substitutions θ with $vars(s) \subseteq dom(\theta)$, $[\![s]\!]\theta\{x \mapsto \sigma_1\} = [\![s]\!]\theta\{x \mapsto \sigma_2\}$ iff $\sigma_1 = \sigma_2$.

THEOREM 11.11 : Let s be a stream value which depends on $x \in vars(s)$, θ a substitution s.t. $vars(s) \subseteq dom(\theta)$ and $\theta(x')$ regular for all $x' \in vars(s) \setminus \{x\}$;

then there exists $n > 0$ s.t. if σ is regular with period p multiple of n, that is, $p = k \cdot n$ for some $k > 0$, then $[\![s]\!]\theta\{x \mapsto \sigma\}$ is regular with period p' which is a divisor of p.

Proof: The proof proceeds by induction on s.

BASIS: If s is a variable, then it must necessarily be x, otherwise s would not depend on x; therefore the thesis trivially holds for all $n > 0$ since by definition $[\![s]\!]\theta\{x \mapsto \sigma\} = (\theta\{x \mapsto \sigma\})(x) = \sigma$.

STEP: we proceed by case analysis on s.

- if $s = n : s_1$ or $s = s_1\hat{}$, then by definition, if s depends on x, then the same holds for s_1; furthermore, by definition, $[\![n : s_1]\!]\theta = n : [\![s_1]\!]\theta$ and $[\![s_1\hat{}]\!]\theta = ([\![s_1]\!]\theta)\hat{}$. By inductive hypothesis, there exists $n > 0$ s.t. if σ is regular with period $p = k \cdot n$ for some $k > 0$, then $[\![s_1]\!]\theta\{x \mapsto \sigma\}$ is regular with period p' which is a divisor of p. Since $[\![s]\!]\theta' = n : [\![s_1]\!]\theta'$ or $[\![s]\!]\theta' = ([\![s_1]\!]\theta')\hat{}$ then by Proposition 11.4 $[\![s]\!]\theta\{x \mapsto \sigma\}$ is regular with the same period p' of $[\![s_1]\!]\theta'$.
- if $s = s_1\ op\ s_2$, then by definition, if s depends on x, then either s_1 or s_2 depends on x; furthermore, by definition, $[\![s_1\ op\ s_2]\!]\theta = [\![s_1]\!]\theta\ op\ [\![s_2]\!]\theta$.

  If both s_1 and s_2 depend on x, then, by inductive hypothesis, there exist $n_1, n_2 > 0$ s.t. if σ_i is regular with period $p_i = k_i \cdot n_i$ for some $k_i > 0$, then $[\![s_i]\!]\theta\{x \mapsto \sigma_i\}$ is regular with period p'_i which divides p_i, for $i = 1, 2$. Let $n = n_1 \cdot n_2$ and σ a regular stream with period $p = k \cdot n$ for some $k > 0$; then $[\![s_i]\!]\theta\{x \mapsto \sigma\}$ is regular with period p''_i which divides p for $i = 1, 2$. If $\theta' = \theta\{x \mapsto \sigma\}$, then by Proposition 11.5 $[\![s_1\ op\ s_2]\!]\theta' = [\![s_1]\!]\theta'\ op\ [\![s_2]\!]\theta'$ has period p' which divides $lcd(p''_1, p''_2)$; since both p''_1 and p''_2 divide p, then $lcd(p''_1, p''_2)$ divides p because positive integers partially ordered by divisibility are a lattice where lcd is the join. Hence, we can conclude that p' divides p by transitivity of divisibility.

  If only one stream value between s_1 and s_2 depends on x, then, without loss of generality, let us assume that s_2 does not depend on x; hence, by straightforward induction on s_2, by the hypotheses and Proposition 11.4 and Proposition 11.5, one can prove that there exists a regular stream σ_2 s.t. $[\![s_2]\!]\theta\{x \mapsto \sigma_1\} = \sigma_2$ for all streams σ_1. Let p_2 be the period of σ_2; by inductive hypothesis applied to s_1 we know that there exists $n_1 > 0$ s.t. if σ_1 is regular with period $p_1 = k_1 \cdot n_1$ for some $k_1 > 0$, then $[\![s_1]\!]\theta\{x \mapsto \sigma_1\}$ is regular with period p which divides p_1. Let $n = n_1 \cdot p_2$ and σ a regular stream with period $p'_1 = k'_1 \cdot n$ for some $k'_1 > 0$; then $[\![s_1]\!]\theta\{x \mapsto \sigma\}$ is regular with period p' which divides p'_1. If $\theta' = \theta\{x \mapsto \sigma\}$, then by Proposition 11.5 $[\![s_1\ op\ s_2]\!]\theta' = [\![s_1]\!]\theta'\ op [\![s_2]\!]\theta'$ has period p'' which divides $lcd(p', p_2)$; since both p' and p_2 divide p'_1, then $lcd(p', p_2)$ divides p'_1 because positive integers partially ordered by divisibility

are a lattice where *lcd* is the join. Hence, we can conclude that p'' divides p'_1 by transitivity of divisibility.

□

From Theorem 11.11 one can deduce that the interleaving operator cannot be expressed with a stream value, even when operands are restricted to regular streams, as shown by the following corollary.

COROLLARY 11.12 : Let s be a stream value which depends on $x_1, x_2 \in vars(s)$, θ a substitution s.t. $vars(s) \subseteq dom(\theta)$ and $\theta(x')$ regular for all $x' \in vars(s) \setminus \{x_1, x_2\}$. Let f be the binary operator on regular streams defined as follows: $f(\sigma_1, \sigma_2) = [\![s]\!]\theta\{x_1 \mapsto \sigma_1\}\{x_2 \mapsto \sigma_2\}$, for all regular stream σ_1 and σ_2; then f is not the interleaving operator.

Proof: Let us consider a regular stream σ_1 with period p_1; if we apply Theorem 11.11 to variable x_2 and substitution $\theta\{x_1 \mapsto \sigma_1\}$, then we deduce that there exists $n > 0$ s.t. if σ_2 is regular with period p_2 multiple of n, then $[\![s]\!]\theta\{x_1 \mapsto \sigma_1\}\{x_2 \mapsto \sigma_2\}$ is regular with period p which is a divisor of p_2. Therefore, if σ_2 has period $p_2 = (p_1 + 1) \cdot n$, then the period p of $f(\sigma_1, \sigma_2)$ divides p_2; furthermore, by Proposition 11.10 the interleaving of σ_1 and σ_2 has period $p' = 2lcd(p_1, p_2)$ because $p_1 \neq p_2$, therefore p' cannot divide p_2 and, hence, $f(\sigma_1, \sigma_2)$ cannot be the correct result of the interleaving of σ_1 and σ_2. □

DISCUSSION By proving Theorem 11.11 and its corollary, we showed that in our calculus it is not possible to write a function that mimics the behaviour of the interleaving operator. However, one may ask whether it is possible to define a function which exploits regular corecursion to return stream values of different shape, depending on the values of the parameters, as happens with the following definition:

```
interleave(s1,s2) =  s1(0):interleave(s2,s1^)
```

While with the lazy evaluation this function works correctly also for non regular streams, this is not the case for regular corecursion; for instance, if `nat()` returns the stream of natural numbers, as defined in Section 9.2, then in the calculus `interleave(nat(),nat())` diverges simply because in the infinite sequence `nat(),nat()^,nat()^^,...` there is no stream that occurs at least twice.

12

Extended calculus

In this chapter we consider an extension of the calculus with an *interleaving* operator, which gives a stream whose elements are alternatively those of the arguments. This latter operator is interesting because, as shown in Chapter 11, it cannot be derived from the others. After having presented the extended calculus, we discuss some interesting examples involving the interleaving.

In Figure 12.1 and Figure 12.2 we report the extended calculus.

A binary stream operator is now either a pointwise arithmetic operation or the interleaving operator (‖). Correspondingly, the rule for the binary operator has been generalized. Finally, two new rules have been added for the judgment $at_\rho(s, i) = n$, handling the interleaving operator. Rule (AT-‖-EVEN) is used for even indexes, and propagates the evaluation to the left-hand side stream; analogously, for odd indexes, rule (AT-‖-ODD) is applied and the evaluation propagates the evaluation to the right-hand side stream.

We show some examples of usage of the interleaving operator. The following function

```
dup_occ() = 0:1:(dup_occ() || dup_occ())
```

generates the stream which alternates sequences of occurrences of `0` and `1`, with the number of repetitions of the same number duplicated at each step, that is,

```
0:1:0:0:1:1:0:0:0:0: ...
```

As a more involved and general example, the use of the interleaving operator allows the following pattern for generating the infinite sequence of numeric labels obtained by a breadth-first visit of an infinite complete binary tree where the labels of children are defined in terms of that of their parent.

$\overline{fd}$	$::=$	$fd_1 \dots fd_n$	program
fd	$::=$	$f(\overline{x}) = se$	function declaration
e	$::=$	$se \mid ne \mid be$	expression
se	$::=$	$x \mid \texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2 \mid ne\texttt{:}se \mid se\hat{}\ \mid se_1\ op\ se_2 \mid f(\overline{e})$	stream expression
ne	$::=$	$x \mid se(ne) \mid ne_1\ op\ ne_2 \mid 0 \mid 1 \mid 2 \mid \dots$	numeric expression
be	$::=$	$x \mid \texttt{true} \mid \texttt{false} \mid \dots$	boolean expression
op	$::=$	$[nop] \mid \Vert$	binary stream operator
nop	$::=$	$+ \mid - \mid * \mid /$	arithmetic operation

FIGURE 12.1 Extended stream calculus: syntax

c	$::=$	$f(\overline{v})$	(evaluated) call
v	$::=$	$s \mid n \mid b$	value
s	$::=$	$x \mid n{:}s \mid s\hat{} \mid s_1\ op\ s_2$	(open) stream value
i, n	$::=$	$0 \mid 1 \mid 2 \mid \ldots$	index, numeric value
b	$::=$	$\texttt{true} \mid \texttt{false}$	boolean value
τ	$::=$	$c_1 \mapsto x_1 \ \ldots\ c_n \mapsto x_n \quad (n \geq 0)$	call trace
ρ	$::=$	$x_1 \mapsto s_1 \ldots x_n \mapsto s_n \quad (n \geq 0)$	environment

$$\text{(VAL)}\ \frac{}{v, \rho, \tau \Downarrow (v, \rho)}$$

$$\text{(CONS)}\ \frac{ne, \rho, \tau \Downarrow (n, \rho) \quad se, \rho, \tau \Downarrow (s, \rho')}{ne{:}se, \rho, \tau \Downarrow (n{:}s, \rho')} \qquad \text{(TAIL)}\ \frac{se, \rho, \tau \Downarrow (s, \rho')}{se\hat{}, \rho, \tau \Downarrow (s\hat{}, \rho')}$$

$$\text{(OP)}\ \frac{se_1, \rho, \tau \Downarrow (s_1, \rho_1) \quad se_2, \rho, \tau \Downarrow (s_2, \rho_2)}{se_1\ op\ se_2, \rho, \tau \Downarrow (s_1\ op\ s_2, \rho_1 \sqcup \rho_2)}$$

$$\text{(IF-T)}\ \frac{be, \rho, \tau \Downarrow (\texttt{true}, \rho) \quad se_1, \rho, \tau \Downarrow (s, \rho')}{\texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2, \rho, \tau \Downarrow (s, \rho')} \qquad \text{(IF-F)}\ \frac{be, \rho, \tau \Downarrow (\texttt{false}, \rho) \quad se_2, \rho, \tau \Downarrow (s, \rho')}{\texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2, \rho, \tau \Downarrow (s, \rho')}$$

$$\text{(ARGS)}\ \frac{e_i, \rho, \tau \Downarrow (v_i, \rho_i) \quad \forall i \in 1..n \qquad f(\overline{v}), \widehat{\rho}, \tau \Downarrow (s, \rho')}{f(\overline{e}), \rho, \tau \Downarrow (s, \rho')} \quad \begin{array}{l} \overline{e} = e_1, \ldots, e_n \text{ not of shape } \overline{v} \\ \overline{v} = v_1, \ldots, v_n \\ \widehat{\rho} = \bigsqcup_{i \in 1..n} \rho_i \end{array}$$

$$\text{(INVK)}\ \frac{se[\overline{v}/\overline{x}], \rho, \tau\{c \mapsto x\} \Downarrow (s, \rho')}{c, \rho, \tau \Downarrow (x, \rho'\{x \mapsto s\})} \quad \begin{array}{l} c \notin dom(\tau_{\approx\rho}) \\ x \text{ fresh} \\ fbody(f) = (\overline{x}, se) \end{array} \qquad \text{(COREC)}\ \frac{}{c, \rho, \tau \Downarrow (x, \rho)} \quad \tau_{\approx\rho}(c) = x$$

$$\text{(AT)}\ \frac{se, \rho, \tau \Downarrow (s, \rho') \quad ne, \rho, \tau \Downarrow (i, \rho)}{se(ne), \rho, \tau \Downarrow (n, \rho)} \quad at_{\rho'}(s, i) = n$$

$$\text{(AT-VAR)}\ \frac{at_\rho(\rho(x), i) = n'}{at_\rho(x, i) = n'} \qquad \text{(AT-CONS-0)}\ \frac{}{at_\rho(n{:}s, 0) = n} \qquad \text{(AT-CONS-SUCC)}\ \frac{at_\rho(s, i-1) = n'}{at_\rho(n{:}s, i) = n'} \quad i > 0$$

$$\text{(AT-TAIL)}\ \frac{at_\rho(s, i+1) = n}{at_\rho(s\hat{}, i) = n} \qquad \text{(AT-NOP)}\ \frac{at_\rho(s_1, i) = n_1 \quad at_\rho(s_2, i) = n_2}{at_\rho(s_1\ [nop]\ s_2, i) = n_1\ nop\ n_2}$$

$$\text{(AT-}\|\text{-EVEN)}\ \frac{at_\rho(s_1, i) = n}{at_\rho(s_1 \parallel s_2, 2i) = n} \qquad \text{(AT-}\|\text{-ODD)}\ \frac{at_\rho(s_2, i) = n}{at_\rho(s_1 \parallel s_2, 2i+1) = n}$$

FIGURE 12.2 Extended stream calculus: operational semantics

```
bfs() = k:(e1[bfs()]||e2[bfs()])
```

In the pattern, k denotes the numeric constant labeling the root of the tree, while $e_i[\ ]$, for $i = 1, 2$, are stream expressions containing one hole and no occurrences of the interleaving operator; they define the labels of the left and right children, respectively.

A first example of instantiation is defined by $k = 0$, $e_1[\ ] = e_2[\ ] =$ `_[+][1]` to get the non-regular stream *depth* of depths of nodes: $depth(i) = \lfloor \log_2(i+1) \rfloor$ for all $i \geq 0$. Without the interleaving operator such a stream can be defined if one introduces pointwise floor and $\log_2$ as primitive operators of the calculus.

If we take $k = 0$, $e_1[\ ] =$ `_[*][2]` and $e_2[\ ] =$`(_[*][2])[+][1]`, then we obtain the non-regular stream *bin_dec* of decodings of binary natural numbers in increasing order of digits: $bin_dec(i) = (i+1) \mod 2^{\lfloor \log_2(i+1) \rfloor}$ for all $i \geq 0$. Also in this case other primitive pointwise arithmetic operators have to be introduced to define such a stream without the use of the interleaving operator.

As a final example, we can consider $k = 0$, $e_1[\ ] =$`(_[*][10])[+][2]` and $e_2[\ ] =$`(_[*][10])[+][4]` to define the non-regular stream of sorted words built on the two digits 2 and 4:

```
0:2:4:22:24:42:44:...
```

13

Extended well-definedness check

In this chapter we present a well-definedness check that handles the interleaving operator as well. The introduction of such operator adds a non-trivial complexity to the calculus, and a more involved algorithm needs to be devised. After extending the abstract definition with the rules for the interleaving operator, we discuss a new algorithm for checking well-definedness. The algorithm requires two steps and mimics the access to an index of the stream. The main technical result of this chapter is Theorem 13.5, in which we prove that the enhanced well-definedness check is sound and complete with respect to the function *at*.

13.1 Extended definition

First of all, the function $_ \Downarrow \theta$ defined in Chapter 10 is extended to the interleaving operator as follows:

$$(\llbracket s_1 \parallel s_2 \rrbracket \theta)(2i) = \llbracket s_1 \rrbracket \theta(i) \qquad i \geq 0$$
$$(\llbracket s_1 \parallel s_2 \rrbracket \theta)(2i+1) = \llbracket s_2 \rrbracket \theta(i) \qquad i \geq 0$$

The extended well-definedness check mimics the steps for accessing an arbitrary index, as the previous version. However, a non-trivial generalization is needed. Indeed, in this case, checking that a result (ρ, s) is well-defined requires two steps:

1. First, a derivation tree needs to be constructed for a judgment $\mathsf{wd}^{y}_{\rho}(s, \emptyset) \rightsquigarrow C$, where C is a set of *checks*, which are pairs (a, b) of rational numbers.
2. Then, the set of checks generated in the first step is analyzed.

We describe now the two steps in more detail.

The judgment $\mathsf{wd}^{y}_{\rho}(s, \emptyset) \rightsquigarrow C$ is analogous to that defined in Figure 10.1. However, in this case, accessing an index i on s leads to accessing indexes which are of shape $a \cdot i + b$, rather than of shape $i + b$, as in functions *linear* in i. The case without interleaving, considered in previous sections, corresponds to $a = 1$. The well-definedness check performs a symbolic computation using a map, m from variables to pairs (a, b) of rational numbers, with $0 < a \leq 1$. Moreover, the judgment is decorated with the variable y which is the starting point of the check.

$m \quad ::= \quad x_1 \mapsto (a_1, b_1) \ \ldots \ x_n \mapsto (a_n, b_n)$ map from variables to pairs of numbers

$$\text{(MAIN)}\frac{\mathsf{wd}^{y}_{\rho'}(y, \emptyset) \rightsquigarrow C}{wd(\rho, y, v)} \quad \begin{array}{l}\rho' = \rho\{y \mapsto v\}\\ \vdash_{OK} C\end{array}$$

$$\text{(WD-VAR)}\frac{\mathsf{wd}^{y}_{\rho}(\rho(x), m\{x \mapsto (1,0)\}) \rightsquigarrow C}{\mathsf{wd}^{y}_{\rho}(x, m) \rightsquigarrow C} \quad x \notin dom(m)$$

$$\text{(WD-COREC)}\frac{}{\mathsf{wd}^{y}_{\rho}(x, m) \rightsquigarrow \{(a, b)\}} \quad m(x) = (a, b)$$

$$\text{(WD-FV)}\frac{}{\mathsf{wd}^{y}_{\rho}(x, m) \rightsquigarrow \emptyset} \quad x \notin dom(\rho) \qquad \text{(WD-CONS)}\frac{\mathsf{wd}^{y}_{\rho}(s, (id, -1) \cdot m) \rightsquigarrow C}{\mathsf{wd}^{y}_{\rho}(n\colon s, m) \rightsquigarrow C}$$

$$\text{(WD-TAIL)}\frac{\mathsf{wd}^{y}_{\rho}(s, (id, +1) \cdot m) \rightsquigarrow C}{\mathsf{wd}^{y}_{\rho}(s\hat{\ }, m) \rightsquigarrow C} \qquad \text{(WD-NOP)}\frac{\mathsf{wd}^{y}_{\rho}(s_1, m) \rightsquigarrow C_1 \quad \mathsf{wd}^{y}_{\rho}(s_2, m) \rightsquigarrow C_2}{\mathsf{wd}^{y}_{\rho}(s_1[nop]s_2, m) \rightsquigarrow C_1 \cup C_2}$$

$$\text{(WD-}\|\text{)}\frac{\mathsf{wd}^{y}_{\rho}(s_1, (\div 2, \div 2) \cdot m) \rightsquigarrow C_1 \quad \mathsf{wd}^{y}_{\rho}(s_2, (\div 2, -1 \div 2) \cdot m) \rightsquigarrow C_2}{\mathsf{wd}^{y}_{\rho}(s_1 \parallel s_2, m) \rightsquigarrow C_1 \cup C_2}$$

FIGURE 13.1 Extended well-definedness check

The rules are presented in Figure 13.1.

Rules(MAIN) is analogous to that in Figure 10.1, with the additional condition that the checks generated in the premise should be successfully analyzed in the second step, as will be detailed below. Rules (WD-VAR), (WD-CONS), (WD-TAIL), and (WD-NOP) are a generalized version of those in Figure 10.1 where, moreover, checks are collected. Notably, in rule (WD-VAR), a variable defined in the environment, when found the first time, is added in the map, with initial value $(1, 0)$, corresponding to the identity function, and in rules (WD-CONS) and (WD-TAIL) the first component of the pair is left unchanged, while the second one is decremented/incremented by one, respectively. We write $(id, -1) \cdot m$ for the composition of m with the function which associates with each (a, b) the pair $(a, b-1)$, and analogously for $(id, +1) \cdot m$.

In rule (WD-$\|$) the two premises deal with even and odd indexes, respectively; in both cases the first component of the pair is divided by 2, the second component is divided by 2 as well, but for odd indexes it is first decremented by one. As in the rules above, checks are collected, and we use analogous notations.

The key rule is (WD-COREC), handling the case when a cyclic reference associated to a pair (a, b) is found. In this case, a check to be analyzed in the second step is generated.

The second step of the algorithm analyzes the set of checks generated in the first step as described below.

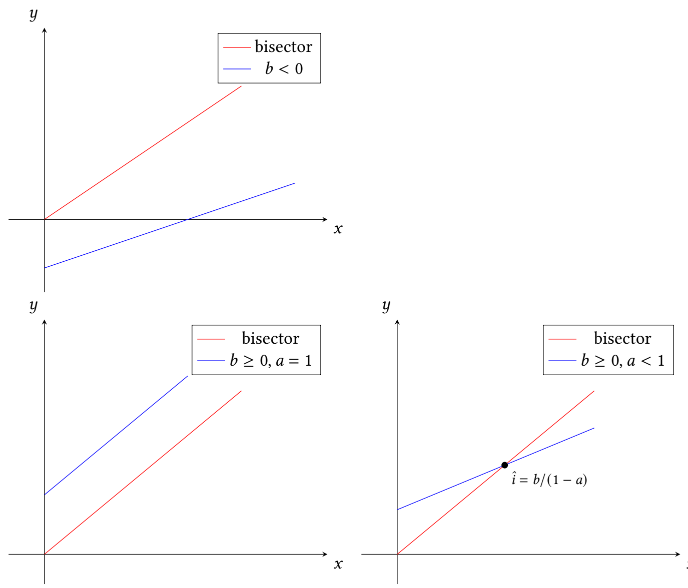

FIGURE 13.2 Possible cases of linear functions

```
INPUT: wd^y_ρ(x_1, m_1) ⇝ {(a_1, b_1)}, ..., wd^y_ρ(x_n, m_n) ⇝ {(a_n, b_n)}
// leaves of the derivation tree
IF b_i < 0 for all h ∈ 1..n SUCCESS
IF a_i = 1 and b_i ≥ 0 for some h ∈ 1..n FAIL
for h in 1..n // a_i < 1 and b_i ≥ 0 for all h ∈ 1..n
î_h ← ⌊b_h/(1 − a_h)⌋ // b_h/(1 − a_h) is the solution of a_h · i + b_h = i
ĵ_h = start(î_h, (a_h, b_h), m_h(y))// starting index needed to obtain i_h
ĵ = max{ĵ_h | h ∈ 1..n, ĵ ≥ 0}
for j in 0..ĵ
 if a cyclic reference for the same variable and index
  is found in the derivation tree of at_ρ(y, j) FAIL
SUCCESS
```

In order to better explain the different cases which may occur when a cyclic reference is found, recall that the pair (a, b) represents the slope and the offset of a linear function. When a cyclic reference is found, the cases which could lead to non-termination are those where $a \cdot i + b$ is greater or equal than i. To see when this happens, we can compare the linear function $a \cdot i + b$ with the identity function (the bisector). Since a turns out to be less or equal than 1 but positive, the slope of the linear function is less or equal than that of the bisector. The possible cases are illustrated in Figure 13.2.

When $b < 0$, $a \cdot i + b$ is strictly less than i for all i, hence we have success. When $a = 1$ and $b \geq 0$, $a \cdot i + b$ is greater or equal than i for all i, hence we have failure. The challenging case is when $b \geq 0$ and $a < 1$. As shown in the picture, in this case $a \cdot i + b$ is greater or equal than i only for a finite set of indexes, namely, those less or equal than the floor of the solution $\hat{i}$ of $a \cdot i + b = i$, that is, the abscissa of the intersection of the two lines. This means that, if we access an index $i \leq \hat{i}$ on some variable x, starting from accessing some index j on the

$$\dfrac{\dfrac{\dfrac{T_1 \quad T_2}{\mathsf{wd}^x_\rho((x\,[+]\,[1]) \;\|\; (x\,[+]\,[1]), \{x \mapsto (1,-1)\}) \rightsquigarrow \{(\frac{1}{2},-\frac{1}{2}),(\frac{1}{2},-1),(1,-1)\}}\,(\text{WD-}\|)}{\mathsf{wd}^x_\rho(0 : ((x\,[+]\,[1]) \;\|\; (x\,[+]\,[1])), \{x \mapsto (1,0)\}) \rightsquigarrow \{(\frac{1}{2},-\frac{1}{2}),(\frac{1}{2},-1),(1,-1)\}}\,(\text{WD-CONS})}{\mathsf{wd}^x_\rho(x,\varnothing) \rightsquigarrow \{(\frac{1}{2},-\frac{1}{2}),(\frac{1}{2},-1),(1,-1)\}}\,(\text{WD-VAR})$$

$$T_1 = \dfrac{\dfrac{}{\mathsf{wd}^x_\rho(x, \{x \mapsto (\frac{1}{2},-\frac{1}{2})\}) \rightsquigarrow \{(\frac{1}{2},-\frac{1}{2})\}}\,(\text{WD-COREC}) \qquad \dfrac{\dots}{\mathsf{wd}^x_\rho([1], \{\dots\}) \rightsquigarrow \{(1,-1)\}}\,(\text{WD-CONS})}{\mathsf{wd}^x_\rho(x\,[+]\,[1], \{x \mapsto (\frac{1}{2},-\frac{1}{2})\}) \rightsquigarrow \{(\frac{1}{2},-\frac{1}{2}),(1,-1)\}}\,(\text{WD-NOP})$$

$$T_2 = \dfrac{\dfrac{}{\mathsf{wd}^x_\rho(x, \{x \mapsto (\frac{1}{2},-1)\}) \rightsquigarrow \{(\frac{1}{2},-1)\}}\,(\text{WD-COREC}) \qquad \dfrac{\dots}{\mathsf{wd}^x_\rho([1], \{\dots\}) \rightsquigarrow \{(1,-1)\}}\,(\text{WD-CONS})}{\mathsf{wd}^x_\rho(x\,[+]\,[1], \{x \mapsto (\frac{1}{2},-1)\}) \rightsquigarrow \{(\frac{1}{2},-1),(1,-1)\}}\,(\text{WD-NOP})$$

FIGURE 13.3 Reduction of $\mathsf{wd}^x_\rho(x,\varnothing) \rightsquigarrow \{(\frac{1}{2},-\frac{1}{2}),(\frac{1}{2},-1),(1,-1)\}$ with $\rho = \{x \mapsto 0 : ((x\,[+]\,[1]) \;\|\; (x\,[+]\,[1]))\}$.

initial variable y, then we could access a greater or equal index on x again.

In this case we compute, for each of such solutions $\hat{i}_h$, the corresponding starting index on y. This can be done as follows:

```
INPUT: î_h, (a_h, b_h), (a, b)// wd^y_ρ(x_h, m_h) ⇝ {(a_h, b_h)}
// î_h solution of a_h · i + b_h = i
// (a, b) = m_h(y) slope and offset from starting point of
   derivation
a' = a/a_h , b' = (b−b_h)/a_h
//slope  and offset from start to first occurrence of x_h
ĵ_h = ⌊(î_h−b')/a'⌋//initial index needed to reach î_h
```

13.2 Examples

We illustrate now the well-definedness check on some examples.

Figure 13.3 shows the derivation tree of $\mathsf{wd}^x_\rho(x,\varnothing) \rightsquigarrow \{(\frac{1}{2},-\frac{1}{2}),(\frac{1}{2},-1),(1,-1)\}$ with $\rho = \{x \mapsto 0 : ((x\,[+]\,[1]) \;\|\; (x\,[+]\,[1]))\}$, which corresponds to the stream `bfs_level()` discussed above. In this first example there is no need of the second step of the checking algorithm because all cyclic references fall within case $b \leq 0$. We left unspecified the reductions of $\mathsf{wd}^x_\rho([1],\{\dots\}) \rightsquigarrow \{(1,-1)\}$ due to space reasons, but it is easy to see that for this particular branch the check is trivial.

Figure 13.4 shows the derivation of $\mathsf{wd}^x_\rho(x,\varnothing) \rightsquigarrow \{(\frac{1}{4},1),(\frac{1}{4},-\frac{1}{2}),(\frac{1}{2},-\frac{3}{2})\}$ with $\rho = \{x \mapsto (x^\wedge \,\|\, x) \;\|\; 0 : x\}$. The represented stream is the one which is constantly equal to 0, but we still deem this example useful to reason on the interleaving operator. In the reduction tree at the top of the figure there are three leaves due to the presence of cyclic references. Two of them generate trivial set of checks (case $b < 0$), while in the leftmost one the pair associated with variable x has shape $a < 0$ and $b > 0$. At this point the second step of

$$\dfrac{\dfrac{T_1 \qquad T_2}{\mathsf{wd}^x_\rho((x^\wedge \parallel x) \;\parallel\; 0 : x, \{x \mapsto (1,0)\}) \rightsquigarrow \{(\frac{1}{4},1),(\frac{1}{4},-\frac{1}{2}),(\frac{1}{2},-\frac{3}{2})\}}\;(\textsc{wd-}\parallel)}{\mathsf{wd}^x_\rho(x,\varnothing) \rightsquigarrow \{(\frac{1}{4},1),(\frac{1}{4},-\frac{1}{2}),(\frac{1}{2},-\frac{3}{2})\}}\;(\textsc{wd-var})$$

$$T_1 = \dfrac{\dfrac{\dfrac{}{\mathsf{wd}^x_\rho(x, \{x \mapsto (\frac{1}{4},1)\}) \rightsquigarrow \{(\frac{1}{4},1)\}}\;(\textsc{wd-corec})}{\mathsf{wd}^x_\rho(x^\wedge, \{x \mapsto (\frac{1}{4},0)\}) \rightsquigarrow \{(\frac{1}{4},1)\}}\;(\textsc{wd-tail}) \qquad \dfrac{}{\mathsf{wd}^x_\rho(x, \{x \mapsto (\frac{1}{4},-\frac{1}{2})\}) \rightsquigarrow \{(\frac{1}{4},-\frac{1}{2})\}}\;(\textsc{wd-corec})}{\mathsf{wd}^x_\rho(x^\wedge \parallel x, \{x \mapsto (\frac{1}{2},0)\}) \rightsquigarrow \{(\frac{1}{4},1),(\frac{1}{4},-\frac{1}{2})\}}\;(\textsc{wd-}\parallel)$$

$$T_2 = \dfrac{\dfrac{}{\mathsf{wd}^x_\rho(x, \{x \mapsto (\frac{1}{2},-\frac{3}{2})\}) \rightsquigarrow \{(\frac{1}{2},-\frac{3}{2})\}}\;(\textsc{wd-corec})}{\mathsf{wd}^x_\rho(0 : x, \{x \mapsto (\frac{1}{2},-\frac{1}{2})\}) \rightsquigarrow \{(\frac{1}{2},-\frac{3}{2})\}}\;(\textsc{wd-cons})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{at_\rho(0 : x, 0) = 0}\;(\textsc{at-cons-0})}{at_\rho((x^\wedge \parallel x) \;\parallel\; 0 : x, 1) = 0}\;(\textsc{at-}\parallel\textsc{-odd})}{at_\rho(x, 1) = 0}\;(\textsc{at-var})}{at_\rho(x^\wedge, 0) = 0}\;(\textsc{at-tail})}{at_\rho(x^\wedge \parallel x, 0) = 0}\;(\textsc{at-}\parallel\textsc{-even})}{at_\rho((x^\wedge \parallel x) \;\parallel\; 0 : x, 0) = 0}\;(\textsc{at-}\parallel\textsc{-even})}{at_\rho(x, 0) = 0}\;(\textsc{at-var}) \qquad \dfrac{\dfrac{\dfrac{}{at_\rho(0 : x, 0) = 0}\;(\textsc{at-cons-0})}{at_\rho((x^\wedge \parallel x) \;\parallel\; 0 : x, 1) = 0}\;(\textsc{at-}\parallel\textsc{-odd})}{at_\rho(x, 1) = 0}\;(\textsc{at-var})$$

FIGURE 13.4 Reduction of $\mathsf{wd}^x_\rho(x,\varnothing) \rightsquigarrow \{(\frac{1}{4},1),(\frac{1}{4},-\frac{1}{2}),(\frac{1}{2},-\frac{3}{2})\}$ with $\rho = \{x \mapsto (x^\wedge \parallel x) \;\parallel\; 0 : x\}$.

$$
\dfrac{
\dfrac{T_1 \qquad T_2}{\mathsf{wd}^x_\rho(y \parallel (y \parallel x\hat{}), \{x \mapsto (1,0)\}) \rightsquigarrow \{(1,-1),(\frac{1}{4},\frac{1}{4})\}}\ (\textsc{wd-}\|)
\qquad
\dfrac{\dfrac{\dfrac{}{\mathsf{wd}^x_\rho(x, \{x \mapsto (1,2)\}) \rightsquigarrow \{(1,2)\}}\ (\textsc{wd-corec})}{\mathsf{wd}^x_\rho(x\hat{}, \{x \mapsto (1,1)\}) \rightsquigarrow \{(1,2)\}}\ (\textsc{wd-tail})}{\mathsf{wd}^x_\rho(x\hat{}\hat{}, \{x \mapsto (1,0)\}) \rightsquigarrow \{(1,2)\}}\ (\textsc{wd-tail})
}{
\dfrac{\mathsf{wd}^x_\rho((y \parallel (y \parallel x\hat{}))\ [+]\ x\hat{}\hat{}, \{x \mapsto (1,0)\}) \rightsquigarrow \{(1,-1),(\frac{1}{4},\frac{1}{4}),(1,2)\}}{\mathsf{wd}^x_\rho(x, \varnothing) \rightsquigarrow \{(1,-1),(\frac{1}{4},\frac{1}{4}),(1,2)\}}\ (\textsc{wd-var})
}\ (\textsc{wd-nop})
$$

$$
T_1 = \dfrac{\dfrac{\dfrac{}{\mathsf{wd}^x_\rho(y, \{x \mapsto (\frac{1}{2},-1), y \mapsto (1,-1)\}) \rightsquigarrow \{(1,-1)\}}\ (\textsc{wd-corec})}{\mathsf{wd}^x_\rho(0 : y, \{x \mapsto (\frac{1}{2},0), y \mapsto (1,0)\}) \rightsquigarrow \{(1,-1)\}}\ (\textsc{wd-cons})}{\mathsf{wd}^x_\rho(y, \{x \mapsto (\frac{1}{2},0)\}) \rightsquigarrow \{(1,-1)\}}\ (\textsc{wd-var})
$$

$$
T_2 = \dfrac{
\dfrac{\dfrac{\dfrac{}{\mathsf{wd}^x_\rho(y, \{x \mapsto (\frac{1}{4},-\frac{5}{4}), y \mapsto (1,-1)\}) \rightsquigarrow \{(1,-1)\}}\ (\textsc{wd-corec})}{\mathsf{wd}^x_\rho(0 : y, \{x \mapsto (\frac{1}{4},-\frac{1}{4}), y \mapsto (1,0)\}) \rightsquigarrow \{(1,-1)\}}\ (\textsc{wd-cons})}{\mathsf{wd}^x_\rho(y, \{x \mapsto (\frac{1}{4},-\frac{1}{4})\}) \rightsquigarrow \{(1,-1)\}}\ (\textsc{wd-var})
\qquad T_3
}{\mathsf{wd}^x_\rho(y \parallel x\hat{}, \{x \mapsto (\frac{1}{2},-\frac{1}{2})\}) \rightsquigarrow \{(\frac{1}{4},\frac{1}{4})\}}\ (\textsc{wd-}\|)
$$

$$
T_3 = \dfrac{\dfrac{}{\mathsf{wd}^x_\rho(x, \{x \mapsto (\frac{1}{4},\frac{1}{4})\}) \rightsquigarrow \{(\frac{1}{4},\frac{1}{4})\}}\ (\textsc{wd-corec})}{\mathsf{wd}^x_\rho(x\hat{}, \{x \mapsto (\frac{1}{4},-\frac{3}{4})\}) \rightsquigarrow \{(\frac{1}{4},\frac{1}{4})\}}\ (\textsc{wd-tail})
$$

FIGURE 13.5 Reduction of $\mathsf{wd}^x_\rho(x, \varnothing) \rightsquigarrow \{(1,-1),(\frac{1}{4},\frac{1}{4}),(1,2)\}$ with $\rho = \{x \mapsto (y \parallel (y \parallel x\hat{}))\ [+]\ x\hat{}\hat{}, y \mapsto 0 : y\}$.

the algorithm is triggered to check the pair $(\frac{1}{4}, 1)$. The calculated solution is $\hat{\imath} = \mathit{floor}(b/(1-a)) = \mathit{floor}(1/(1-\frac{1}{4})) = 1$, which means that $at_\rho(x,0)$ and $at_\rho(x,1)$ need to be calculated, as shown at the bottom of Figure 13.4. The two reduction trees are pretty straightforward but we would like to point out that in this example we need to know the value of $at_\rho(x,1)$ in order to have $at_\rho(x,0)$. This is an interesting fact because this can only happen with the interleaving operator, and would lead to not well-defined streams otherwise.

Figure 13.5 shows the reduction of $\mathsf{wd}^x_\rho(x, \varnothing) \rightsquigarrow \{(1,-1),(\frac{1}{4},\frac{1}{4}),(1,2)\}$ with $\rho = \{x \mapsto (y \parallel (y \parallel x\hat{}))\ [+]\ x\hat{}\hat{}, y \mapsto 0 : y\}$. This last example shows the behaviour of the algorithm when dealing with non-well-defined streams. Indeed, the stream in the environment ρ is of such kind. In the tree at the top of the figure, the derivation starts with the unfolding of variable x. Then there is the application of the rule for numeric pointwise operations that propagates the check to the two subterms. The derivation for the one on the left-hand side, reported in the subtrees T_1 and T_2, terminates with two checks to perform in the second step of the algorithm. For what concerns the subtree for the term $x\hat{}\hat{}$, we have the pair $(1,2)$ added to the set of check. Due to this fact we have failure in the second step of the algorithm since this is the case $a = 1$ and $b > 0$. The stream is indeed not well-defined since an access to the first element leads to non-termination, as shown in Figure 13.6.

$$
\dfrac{
\dfrac{\dfrac{\dfrac{}{at_\rho(0:y,0)=0}\ (\text{AT-CONS-0})}{at_\rho(y,0)=0}\ (\text{AT-VAR})}{at_\rho((y\ \|\ (y\ \|\ x\hat{}\,)),0)=0}\ (\text{AT-}\|\text{-EVEN})
\qquad
\dfrac{\dfrac{\dfrac{\vdots}{at_\rho(x,2)=\,!}\ (\text{AT-VAR})}{at_\rho(x\hat{}\,,1)=\,!}\ (\text{AT-TAIL})}{at_\rho(x\hat{}\hat{}\,,0)=\,!}\ (\text{AT-TAIL})
}{
\dfrac{at_\rho((y\ \|\ (y\ \|\ x\hat{}\,))\ [+]\ x\hat{}\hat{}\,,0)=\,!}{at_\rho(x,0)=\,!}\ (\text{AT-VAR})
}\ (\text{AT-NOP})
$$

FIGURE 13.6 Failing reduction of $at_\rho(x,0)$ with $\rho = \{x \mapsto (y\ \|\ (y\ \|\ x\hat{}\,))\ [+]\ x\hat{}\hat{}\,, y \mapsto 0 : y\}$.

13.3 Soundness and completeness of the extended well-definedness check

We show now that well-definedness of a result is a necessary and sufficient condition for termination of access to an arbitrary index.

In Chapter 10, this result has been proved for the version of the calculus as presented in Chapter 9, that is, without interleaving. We show now that the enhanced definition presented in this chapter is still sound and complete with respect to function *at*. To formally express and prove this statement, we introduce some definitions and notations.

In the following, ρ is a closed environment. First of all, since the numeric value obtained as result is not relevant for the following technical treatment, for simplicity we will write $at_\rho(s,i)$ rather than $at_\rho(s,i)=n$. Analogously, we write $\mathsf{wd}_\rho(s,m)$ rather than $\mathsf{wd}_\rho^y(s,m) \rightsquigarrow C$ when the initial variable and the set of generated checks is not relevant. We call *derivation* an either finite or infinite proof tree.

We write $\mathsf{wd}_\rho(s',m') \vdash \mathsf{wd}_\rho(s,m)$ to mean that $\mathsf{wd}_\rho(s',m')$ is a premise of a (meta-)rule where $\mathsf{wd}_\rho(s,m)$ is the consequence, and $\vdash^\star$ for the reflexive and transitive closure of this relation. We use analogous notations for the judgment $at_\rho(s,i)$. Moreover, $at_\rho(x,i) \vdash^\star at_\rho(s,j)$ means that in the path there are no other nodes of shape $at_\rho(x,_)$.

LEMMA 13.1 : The derivation of $at_\rho(s,j)$ is infinite iff the following condition holds:

(AT-∞) for some x, $\{i_k \mid k \geq 0\}$:
$at_\rho(x,i_0) \vdash^\star at_\rho(s,j)$
$at_\rho(x,i_{k+1}) \vdash^\star at_\rho(\rho(x),i_k) \vdash at_\rho(x,i_k)$ for all $k \geq 0$

LEMMA 13.2 : If $at_\rho(x,i') \vdash^\star at_\rho(s,i)$ then,

for all m, $\mathsf{wd}_\rho(x,m') \vdash^\star \mathsf{wd}_\rho(s,m)$ for some m', and

1. if $x \notin dom(m)$, then $x \notin dom(m')$
2. if $m(x)=(a,b)$, then $m'(x)=(a'\cdot a, b'+b)$, $i' = \lfloor a'\cdot i + b' \rfloor$

LEMMA 13.3 : If $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s, m)$, then

- for each i'', $at_\rho(x, i') \vdash^\star at_\rho(s, i)$ for some i and $i' \geq i''$
- if $m(x) = (a, b)$, $m'(x) = (a' \cdot a, b' + b)$, then $i' = \lfloor a' \cdot i + b' \rfloor$

LEMMA 13.4 :

If $at_\rho(x, i') \vdash^\star at_\rho(s, i)$, and s does not contain the interleaving operator, then $at_\rho(x, i' + b) \vdash^\star at_\rho(s, i + b)$, for each $b \geq 0$.

THEOREM 13.5 : Set $\mathsf{wd}^y_\rho(y, \emptyset) \rightsquigarrow C$.

- If $\vdash_{OK} C$ holds then, for all j, the derivation of $at_\rho(y, j)$ is finite.
- If $\vdash_{OK} C$ fails then, for some j, the derivation of $at_\rho(y, j)$ is infinite.

Proof: We consider the following cases.

- $b < 0$ for all $(a, b) \in C$. We show that, for all j, the derivation of $at_\rho(y, j)$ cannot be infinite. Assume by contradiction an infinite derivation path starting from $at_\rho(y, j)$, then, by Lemma 13.1, the following condition holds:
  (AT-∞) for some x, $\{i_k \mid k \geq 0\}$:
  $at_\rho(x, i_0) \vdash^\star at_\rho(y, j)$
  $at_\rho(x, i_{k+1}) \vdash^\star at_\rho(\rho(x), i_k) \vdash at_\rho(x, i_k)$ for all $k \geq 0$
  Then by Lemma 13.2 we have:

  $\vdash^\star \mathsf{wd}_\rho(x, m) \vdash^\star \mathsf{wd}_\rho(j, \emptyset)$
  with $x \notin dom(m)$

  hence by rule (WD-VAR) $\mathsf{wd}_\rho(\rho(x), m_0) \vdash \mathsf{wd}_\rho(x, m)$, with $m_0 = m\{x \mapsto (1, 0)\}$. Moreover, for all $k \geq 0$, if $m_k(x) = (a_k, b_k)$, by Lemma 13.2 we have:

  $\mathsf{wd}_\rho(x, m_{k+1}) \vdash^\star \mathsf{wd}_\rho(\rho(x_k), m_k) \vdash \mathsf{wd}_\rho(x_k, m_k)$
  with $m_{k+1}(x) = (a_{k+1}, b_{k+1})$, $i_{k+1} = \lfloor a_k \cdot i_k + b_k \rfloor$

  For all $k \geq 0$, since $b_k < 0$, we have $i_{k+1} < i_k$, hence we should have an infinite sequence $\ldots < i_k \ldots < \ldots < i_0$, which is impossible.
- $a = 1$ and $b \geq 0$ for some $(a, b) \in C$. We show that there exists j such that the derivation of $at_\rho(y, j)$ is infinite. Indeed, since $\mathsf{wd}^y_\rho(y, \emptyset) \rightsquigarrow C$, we have that $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(\rho(x), m\{x \mapsto (1, 0)\}) \vdash \mathsf{wd}_\rho(x, m) \vdash^\star \mathsf{wd}_\rho(y, \emptyset)$ with $x \notin dom(m)$, $m'(x) = (1, b)$ with $b \geq 0$. Hence we have that:
  - by Lemma 13.3, $at_\rho(x, i') \vdash^\star at_\rho(\rho(x), i)$ for some i', i such that $i' = \lfloor i + b \rfloor$
  - by rule (AT-VAR), $at_\rho(\rho(x), i) \vdash at_\rho(x, i)$
  - by Lemma 13.3, there exist j and $j' \geq i$ such that $at_\rho(x, j') \vdash^\star at_\rho(y, j)$
  - since $a = 1$, $\rho(x)$ does not contain the interleaving operator, hence by Lemma 13.4, since $j' \geq i$, we also have $at_\rho(x, i'') \vdash^\star at_\rho(\rho(x), j')$, hence altogether

$$at_\rho(x, i'') \vdash^\star at_\rho(\rho(x), j') \vdash at_\rho(x, j') \vdash^\star at_\rho(y, j),$$
$$\text{with } i'' = \lfloor j' + b \rfloor$$

Moreover, again by Lemma 13.4, $at_\rho(x, j' + b + b) \vdash^\star at_\rho(\rho(x), j' + b)$ as well, and it is easy to see that there is an infinite derivation for $at_\rho(y, j)$.

- $a_h < 1$ and $b_h \geq 0$ for all $h \in 1..n$, if $C = \{(a_1, b_1), \ldots, (a_n, b_n)\}$. For all $h \in 1..n$, set $\hat{i}_h$ the solution of $a_h \cdot i + b_h = i$, and $\hat{j}_h = \textsc{start}(\hat{i}_h, (a_h, b_h), m_h(y))$ the starting index needed to obtain $\hat{i}_h$, and $\hat{j} = \max\{\hat{j}_h \mid h \in 1..n, \hat{j} \geq 0\}$. We show that there exists j such that the derivation of $at_\rho(y, j)$ is infinite iff there is a cyclic reference to the same variable and index in the derivation of $at_\rho(y, j)$, and $j = \textsc{start}(i)$ for some $i \leq \hat{i}$.

  With the same reasoning of the first case, we have that, if $at_\rho(y, j)$ has an infinite derivation, then we have:

  $$\mathsf{wd}_\rho(\rho(x), m_0) \vdash \mathsf{wd}_\rho(x, m) \vdash^\star \mathsf{wd}_\rho(j, \emptyset),$$
  $$\text{with } x \notin dom(m),\ m_0 = m\{x \mapsto (1, 0)\}$$
  $$\text{for all } k \geq 0, \text{ if } m_k(x) = (a_k, b_k),$$
  $$\mathsf{wd}_\rho(x, m_{k+1}) \vdash^\star \mathsf{wd}_\rho(\rho(x_k), m_k) \vdash \mathsf{wd}_\rho(x_k, m_k)$$
  $$\text{with } m_{k+1}(x) = (a_{k+1}, b_{k+1}),\ i_{k+1} = \lfloor a_k \cdot i_k + b_k \rfloor$$

  Since, for all $k \geq 0$, $(a_k, b_k) = (a_h, b_h)$ for some $h \in 1..n$, and $a_h < 1$, $b_h \geq 0$, we have that $i_{k+1} \geq i_k$ can only hold for a finite set of indexes, those less or equal than some $\hat{i}_h$. Hence, in the infinite sequence $\ldots, i_{k+1}, i_k, \ldots, i_0$, there is necessarily a repeated index, say, i, and this cyclic reference is found starting from some $j \leq \hat{j}$.

□

14

Equality of streams

In this chapter we consider the problem of the equality of stream values. As discussed in the Introduction, and shown in the operational semantics in Figure 9.2, this issue is relevant not only to provide an equality operator which can be used by the programmer, but also for cycle detection in calls. After presenting the formal definition, we discuss some examples of equal streams. The main technical results of the chapter are:

1. Theorem 14.2, stating that if two streams pass the equality check, then they are equal in the sense that access to an arbitrary index yields the same result.
2. Theorem 14.3, which proves a result of partial completeness of the equality.
3. Theorem 14.4 and Theorem 14.16, that prove the existence an effective algorithm for checking the equality of two streams.

14.1 Formal definition

We start with an example to illustrate the issue of detecting equal calls.

```
ones() = 1:ones()
incr_reg(s) = (s(0)+1) : incr_reg(s^)
```

Intuitively, the result of `incr_reg(ones())` should be the stream consisting of infinite occurrences of number 2. However, it is easy to see that this is not the case if cycle detection is based on mere syntactic equality. Indeed, if `incr_reg` is called on `ones()`, that is, on the result $(x, x \mapsto 1\!:\!x)$, then `incr_reg` is recursively called on $(x\hat{}, x \mapsto 1\!:\!x)$ and cycle detection fails because x and $x\hat{}$ are not syntactically equal, even though they denote the same stream. This leads to non-termination, since rule (COREC) will never be applied.

The first step towards a more expressive definition is obtained by considering equality in the free theory of regular terms, as in [3]. This can be done by a coinductive definition[1] which computes the unfolding of variables by looking up their associated values in the environment. This allows us to identify results returned by `ones()` and `altOnes()`, where `altOnes()= 1:1:altOnes()`. Indeed, in the environment of shape $\{x \mapsto 1\!:\!x, y \mapsto 1\!:\!1\!:\!y\}$ resulting from

1 As further discussed at the end of this section, for regular terms the coinductive definition can be turned into an equivalent inductive and algorithmic one.

s	$::=$	$x \mid n\!:\!s \mid s\hat{} \mid s_1\ op\ s_2$	(open) stream value
op	$::=$	$[nop] \mid \|\|$	binary stream operator
ρ	$::=$	$x_1 \mapsto s_1 \ldots x_n \mapsto s_n \quad (n \geq 0)$	environment

$$\text{(VAR-L)}\ \frac{\rho(x) \approx_\rho s}{x \approx_\rho s} \qquad \text{(VAR-R)}\ \frac{s \approx_\rho \rho(x)}{s \approx_\rho x} \qquad \text{(VAR)}\ \frac{}{x \approx_\rho x}$$

$$\text{(CONS)}\ \frac{s_1 \approx_\rho s_2}{n\!:\!s_1 \approx_\rho n\!:\!s_2} \qquad \text{(OP)}\ \frac{s_1 \approx_\rho s_1' \quad s_2 \approx_\rho s_2'}{s_1\ op\ s_2 \approx_\rho s_1'\ op\ s_2'}$$

$$\text{(TAIL-L)}\ \frac{s' \approx_\rho s_2}{s_1\hat{} \approx_\rho s_2}\ \mathsf{Tail}_\rho(s_1) = s' \qquad \text{(TAIL-R)}\ \frac{s_1 \approx_\rho s'}{s_1 \approx_\rho s_2\hat{}}\ \mathsf{Tail}_\rho(s_2) = s'$$

$$\text{(TAIL-CONS)}\ \frac{}{\mathsf{Tail}_\rho(n : s) = s} \qquad \text{(VAR)}\ \frac{\mathsf{Tail}_\rho(\rho(x)) = s}{\mathsf{Tail}_\rho(x) = s} \qquad \text{(TAIL)}\ \frac{\mathsf{Tail}_\rho(s) = s' \quad \mathsf{Tail}_\rho(s') = s''}{\mathsf{Tail}_\rho(s\hat{}) = s''}$$

$$\text{(NOP)}\ \frac{}{\mathsf{Tail}_\rho(s_1 [nop] s_2) = s_1\hat{}[nop] s_2\hat{}} \qquad \text{(}\|\text{)}\ \frac{}{\mathsf{Tail}_\rho(s_1 \| s_2) = s_2 \| s_1\hat{}}$$

FIGURE 14.1 Equality check

the evaluation, one can check by unfolding that the values associated with x and y correspond to the same regular term. However, if one allows other operators in the equational systems, then the equational theory is no longer free, therefore equality of regular terms fails to identify the results of `ones()` and `ones()^` as in the example above.

In order to deal with the tail operator, our solution is based on the key idea that the equality check performs a partial symbolic evaluation of the tail. For instance, with this solution the example `incr_reg(ones())` is correctly handled, since the symbolic evaluation of the tail of $(x, x \mapsto 1\!:\!x)$ returns $(x, x \mapsto 1\!:\!x)$.

The equality check is formalized by the judgment $s_1 \approx_\rho s_2$, coinductively defined in Figure 14.1.

In rules (VAR-L) and (VAR-R), a variable defined in the environment is equal to a stream value if the same holds for its associated stream value. In rule (VAR), a variable is equal to itself.

In rule (CONS), prepending the same element to equal streams gives equal streams.

In rule (OP), two streams defined using the same binary operation are equal if their arguments are respectively equal.

The most interesting rules are those handling the case when one of the two sides of the equality is of shape $s\hat{}$, which are the only ones applicable when the other side is not a variable. Indeed, in such case an attempt is made at computing (a stream value equal to) the tail of s, through the auxiliary function Tail_ρ defined in the bottom part of the figure. Function Tail_ρ is inductively

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x \approx_\rho y\hat{}\hat{}}\ (\text{VAR-L})}{1:x \approx_\rho 1:y\hat{}\hat{}}\ (\text{CONS}) \quad \mathsf{Tail}_\rho(y\hat{}) = 1:y\hat{}\hat{}}{1:x \approx_\rho y\hat{}\hat{}}\ (\text{TAIL-R})}{x \approx_\rho y\hat{}\hat{}}\ (\text{VAR-L})$$

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Tail}_\rho(2:3:1:y\hat{}\hat{}) = 3:1:y\hat{}\hat{}}\ (\text{CONS})}{\mathsf{Tail}_\rho(y) = 3:1:y\hat{}\hat{}}\ (\text{VAR}) \quad \dfrac{}{\mathsf{Tail}_\rho(3:1:y\hat{}\hat{}) = 1:y\hat{}\hat{}}\ (\text{CONS})}{\mathsf{Tail}_\rho(y\hat{}) = 1:y\hat{}\hat{}}\ (\text{TAIL})$$

FIGURE 14.2 $x \approx_\rho y\hat{}\hat{}$ with $\rho = \{x \mapsto 1{:}x, y \mapsto 2:3:1:y\hat{}\hat{}\}$

defined, and the base cases are: the constructor, rule (TAIL-CONS), where the result is simply the tail stream, the arithmetic binary operation, rule (TAIL-NOP), where the tail operator is propagated to substreams, and the interleaving operator, (TAIL-||), where the elements of the tail are alternatively those of the second argument and the tail of the first one. In the other cases, the function is propagated as expected. In particular, in rule (TAIL), to obtain the result we need two subsequent applications of Tail_ρ, the former getting the tail of the argument.

The judgment $s_1 \approx_\rho s_2$ is defined by interpreting the rules in Figure 14.1 *coinductively*, that is, infinite derivations are allowed. Such coinductive definition is the most natural and abstract for infinite objects such as streams, and is convenient for all the proofs present in this chapter. However, such approach does not directly lead to an algorithm, and some additional problems need to be addressed in order to prove that the coinductive check is an effective procedure; we discuss all the details about this matter in Section 14.3.

We now illustrate how the equality check works by some examples. The first one, in Figure 14.2, shows the regular derivation of the equality $x \approx_\rho y\hat{}\hat{}$ in the environment $\rho = \{x \mapsto 1{:}x, y \mapsto 2:3:1:y\hat{}\hat{}\}$.

At the start of the derivation, one stream is a variable, while the other one is of shape $y\hat{}\hat{}$. In this derivation, we applied rule (VAR-L) first, and then rule (TAIL-R) to get a stream value equal to $y\hat{}\hat{}$; we could have applied the rules in the other order as well. The derivation of $\mathsf{Tail}_\rho(y\hat{}) = 1:y\hat{}\hat{}$ is shown in the bottom part of the figure. The result is computed by applying twice the tail operator, and unfolding y. Then, in the main derivation tree, rule (CONS) is applied since the first element of both streams is 1. After that, it is easy to see that there is a regular infinite derivation, denoted by the dots.

In the second example we show how the equality check deals with non-regular streams. Figure 14.3 shows the regular derivation of $x \approx_\rho y$ with $\rho = \{x \mapsto 1{:}(x\ [+]\ x), y \mapsto 1:((1:y)\ [+]\ (1:y))\hat{}\}$.

Here, both x and y denote the stream of all powers of 2. The derivation

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\vdots}{x \approx_\rho y \quad \mathsf{Tail}_\rho(1 : y) = y}\,(\textsc{var-l})
}{x \approx_\rho (1 : y)\hat{}}\,(\textsc{tail-r})
\quad \cdots
}{x \mathbin{[+]} x \approx_\rho (1 : y)\hat{} \mathbin{[+]} (1 : y)\hat{}}\,(\textsc{op})
\quad
\dfrac{}{\mathsf{Tail}_\rho((1 : y) \mathbin{[+]} (1 : y)) = (1 : y)\hat{} \mathbin{[+]} (1 : y)\hat{}}\,(\textsc{tail-nop})
}{x \mathbin{[+]} x \approx_\rho ((1 : y) \mathbin{[+]} (1 : y))\hat{}}\,(\textsc{tail-r})
}{1\mathbin{:}(x \mathbin{[+]} x) \approx_\rho 1 : ((1 : y) \mathbin{[+]} (1 : y))\hat{}}\,(\textsc{cons})
}{1\mathbin{:}(x \mathbin{[+]} x) \approx_\rho y}\,(\textsc{var-r})
}{x \approx_\rho y}\,(\textsc{var-l})
$$

FIGURE 14.3 $x \approx_\rho y$ with $\rho = \{x \mapsto 1\mathbin{:}(x \mathbin{[+]} x), y \mapsto 1 : ((1 : y) \mathbin{[+]} (1 : y))\hat{}\}$

starts with the unfolding of variables x and y, followed by the application of rule (CONS). Then, the tail of the second component is computed, with the reduction tree shown as additional premise in rule (TAIL-R).

After that, the derivation continues by distributing the equality check to the substreams x and $(1 : y)\hat{}$. At this point, it is easy to see that there is a regular infinite derivation, after another application of rule (TAIL-R). For space reasons, we have omitted the derivation for the right premise of rule (OP) which is equal to the left one.

14.2 Soundness and (relative) completeness of the equality check

In this section we prove two important results regarding the equality check. First we prove that the check is sound (Theorem 14.2), which means that if we derive that two streams are equal, then they are equal in the sense that access to an arbitrary index will give the same result. The second result of this section (Theorem 14.3) proves that the equality check is complete if we restrict to regular streams.

SOUNDNESS The soundness of the equality check (Theorem 14.2) relies on the soundness of the Tail judgment (Theorem 14.1). To the aim of the proof of Theorem 14.1 we will write "of shape tail or variable" to indicate streams of the form $s\hat{}$ or x, respectively. For Theorem 14.2, we introduce the notation $at_\rho(s, i) \overset{k}{=} n$, meaning that the proof tree of $at_\rho(s, i + 1) = n$ has depth k.

THEOREM 14.1 : If $\mathsf{Tail}_\rho(s) = s'$ then, for all $i \geq 0$, $at_\rho(s\hat{}, i) \overset{k}{=} n$ implies $at_\rho(s', i) \overset{k'}{=} n$ with $k' \leq k$; furthermore, if s' is of shape tail or variable, then $k' < k$.

Proof: By induction on the rules defining $\mathsf{Tail}_\rho(s) = s'$.

(TAIL-CONS) This is an axiom with conclusion $\mathsf{Tail}_\rho(n\!:s) = s$. We have to show that, for all $i \geq 0$, $at_\rho((n\!:s)\hat{}, i) \overset{k}{=} n$ implies $at_\rho(s, i) \overset{k'}{=} n$ with $k' \leq k$ and $k' < k$ if s is of shape tail or variable; $at_\rho((n\!:s)\hat{}, i) \overset{k}{=} n$ is necessarily derived by rule (AT-TAIL) from $at_\rho(n\!:s, i+1) \overset{k-1}{=} n$ which, in turn, is necessarily derived by rule (AT-CONS-SUCC) from $at_\rho(s, i) \overset{k-2}{=} n$; since $k-2 < k$ the thesis trivially holds.

(VAR) $\mathsf{Tail}_\rho(\rho(x)) = s$ is the premise and $\mathsf{Tail}_\rho(x) = s$ the conclusion. We have to show that, for all $i \geq 0$, $at_\rho(x\hat{}, i) \overset{k}{=} n$ implies $at_\rho(s, i) \overset{k'}{=} n$ with $k' \leq k$ and $k' < k$ if s is of shape tail or variable. Since $at_\rho(x\hat{}, i) \overset{k}{=} n$ is necessarily derived by rule (AT-TAIL) from $at_\rho(x, i+1) \overset{k-1}{=} n$ which, in turn, is necessarily derived by rule (AT-VAR) from $at_\rho(\rho(x), i+1) \overset{k-2}{=} n$, by applying rule (AT-TAIL) $at_\rho(\rho(x)\hat{}, i) \overset{k-1}{=} n$ is derivable; therefore, by inductive hypothesis we have $at_\rho(s, i) \overset{k'}{=} n$ with $k' \leq k-1$; since $k-1 < k$ the thesis trivially holds.

(TAIL) $\mathsf{Tail}_\rho(s) = s', \mathsf{Tail}_\rho(s') = s''$ are the premises, and $\mathsf{Tail}_\rho(s\hat{}) = s''$ the conclusion. We have to show that, for all $i \geq 0$, $at_\rho(s\hat{}\hat{}, i) \overset{k}{=} n$ implies $at_\rho(s'', i) \overset{k''}{=} n$ with $k'' \leq k$ and $k'' < k$ if s'' is of shape tail or variable. Since $at_\rho(s\hat{}\hat{}, i) \overset{k}{=} n$ is necessarily derived by rule (AT-TAIL) from $at_\rho(s\hat{}, i+1) \overset{k-1}{=} n$, by inductive hypothesis on the first premise we have $at_\rho(s', i+1) \overset{k'}{=} n$, with $k' \leq k-1$, and, hence, by applying rule (AT-TAIL) $at_\rho(s'\hat{}, i) \overset{k'+1}{=} n$ is derivable with $k'+1 \leq k$. By inductive hypothesis on the second premise, we have $at_\rho(s'', i) \overset{k''}{=} n$, with $k'' \leq k'+1 \leq k$; furthermore, if s'' is of shape tail or variable, then, again by inductive hypothesis on the second premise, $k'' < k'+1 \leq k$, hence $k'' < k$.

(NOP) This is an axiom with conclusion $\mathsf{Tail}_\rho(s_1[nop]s_2) = s_1\hat{}[nop]s_2\hat{}$. Since in this case $s_1\hat{}[nop]s_2\hat{}$ cannot be of shape tail or variable, we only have to show that, for all $i \geq 0$, $at_\rho((s_1[nop]s_2)\hat{}, i) \overset{k}{=} n$ implies $at_\rho(s_1\hat{}[nop]s_2\hat{}, i) \overset{k'}{=} n$ with $k' \leq k$. Since $at_\rho((s_1[nop]s_2)\hat{}, i) \overset{k}{=} n$ is necessarily derived by rule (AT-TAIL) from $at_\rho(s_1[nop]s_2, i{+}1) \overset{k-1}{=} n$ which, in turn, is necessarily derived by rule (AT-NOP) from $at_\rho(s_1, i+1) \overset{k_1}{=} n_1$ and $at_\rho(s_2, i+1) \overset{k_2}{=} n_2$, with $n_1+n_2 = n$, $\max(k_1, k_2) = k-2$, by applying rule (AT-TAIL) $at_\rho(s_1\hat{}, i) \overset{k_1+1}{=} n_1$ and $at_\rho(s_2\hat{}, i) \overset{k_2+1}{=} n_2$ are derivable, and, hence, by applying rule (AT-NOP) $at_\rho(s_1\hat{}[nop]s_2\hat{}, i) \overset{k'}{=} n$ is derivable, with $k' = \max(k_1+1, k_2+1) + 1 = \max(k_1, k_2) + 2 = k$.

($\|$) This is an axiom with conclusion $\mathsf{Tail}_\rho(s_1 \parallel s_2) = s_2 \parallel s_1\hat{}$. Since in this case $s_2 \parallel s_1\hat{}$ cannot be of shape tail or variable, we only have to

show that, for all $i \geq 0$, $at_\rho((s_1 \parallel s_2)\hat{}, i) \stackrel{k}{=} n$ implies $at_\rho(s_2 \parallel s_1\hat{}, i) \stackrel{k'}{=} n$ with $k' \leq k$. We have that $at_\rho((s_1 \parallel s_2)\hat{}, i) \stackrel{k}{=} n$ is necessarily derived by rule (AT-TAIL) from $at_\rho(s_1 \parallel s_2, i+1) \stackrel{k-1}{=} n$; the proof proceeds by case analysis on the parity of i:

i **even:** $at_\rho(s_1 \parallel s_2, i+1) \stackrel{k-1}{=} n$ is necessarily derived by rule (AT-‖-ODD) from $at_\rho(s_2, \frac{i}{2}) \stackrel{k-2}{=} n$; hence, by applying rule (AT-‖-EVEN) we have the thesis $at_\rho(s_2 \parallel s_1\hat{}, i) \stackrel{k'}{=} n$ with $k' = k - 1 < k$.

i **odd:** $at_\rho(s_1 \parallel s_2, i+1) \stackrel{k-1}{=} n$ is necessarily derived by rule (AT-‖-EVEN) from $at_\rho(s_1, \frac{i+1}{2}) \stackrel{k-2}{=} n$; hence, by applying rule (AT-TAIL) $at_\rho(s_1\hat{}, \frac{i-1}{2}) \stackrel{k-1}{=} n$ and by applying rule (AT-‖-ODD) we get the thesis $at_\rho(s_2 \parallel s_1\hat{}, i) \stackrel{k'}{=} n$ with $k' = k$.

□

THEOREM 14.2 (Soundness of equality): For all $i \geq 0$, if $at_\rho(s_0, i) = n$, $at_\rho(s'_0, i) = n'$, and $s_0 \approx_\rho s'_0$, then $n = n'$.

Proof: Assume that $at_\rho(s_0, i) \stackrel{k}{=} n$, $at_\rho(s'_0, i) \stackrel{k'}{=} n'$. The proof is by Noetherian induction on the pairs (k, k'), with the componentwise order, that is, $(k_1, k_2) \leq (k'_1, k'_2)$ iff $k_1 \leq k'_1$ and $k_2 \leq k'_2$.

BASE We consider the pair $(0, 0)$, that is, the case when the two judgments are derived by the unique axiom (AT-CONS-O), hence, for $i = 0$. We have $at_\rho(n\colon s, 0) = n$, and $at_\rho(n'\colon s', 0) = n'$. Moreover, we have $n\colon s \approx_\rho n'\colon s'$, which has been necessarily derived by rule (CONS), hence $n = n'$.

INDUCTIVE STEP We consider pairs (k, k') where either $k > 0$ or $k' > 0$, and proceed by case analysis on the rule applied for the root.

(VAR-L) We have $x \approx_\rho s'_0$ and $\rho(x) \approx_\rho s'_0$. Moreover, we have $at_\rho(x, i) \stackrel{k}{=} n$, $at_\rho(s'_0, i) \stackrel{k'}{=} n'$ and, since the former judgment has been necessarily derived by rule (AT-VAR), $at_\rho(\rho(x), i) \stackrel{k-1}{=} n$. We can apply the inductive hypothesis and get the thesis.

(VAR-R) This case is symmetrical to the one above.

(VAR) We have $x \approx_\rho x$. We conclude by the fact that, by definition, the judgment $at_\rho(s, i) = n$ is deterministic, therefore $at_\rho(x, i) = n_1$ and $at_\rho(x, i) = n_2$ implies $n_1 = n_2$.

(CONS) We have $m\colon s \approx_\rho m\colon s'$ and $s \approx_\rho s'$. Moreover, we have $at_\rho(m\colon s, i) \stackrel{k}{=} n$, $at_\rho(m\colon s', i) \stackrel{k'}{=} n'$ and, since $k > 0$ or $k' > 0$, we have that $i > 0$ and that these judgments have been necessarily derived by rule (AT-CONS-SUCC); therefore, $at_\rho(s, i-1) \stackrel{k-1}{=} n$,

$at_\rho(s', i-1) \overset{k'-1}{=} n'$. We can now apply the inductive hypothesis and get the thesis.

(OP) By cases on the binary operator *op*.

(NOP): We have $s_1\ [nop]\ s_2 \approx_\rho s'_1\ [nop]\ s'_2$, $s_1 \approx_\rho s'_1$ and $s_2 \approx_\rho s'_2$. Moreover, we have $at_\rho(s_1\ [nop]\ s_2, i) \overset{k}{=} n$, $at_\rho(s'_1\ [nop]\ s'_2, i) \overset{k'}{=} n'$ and, since these judgments have been necessarily derived by rule (AT-NOP), $at_\rho(s_1, i) \overset{k_1}{=} l$, $at_\rho(s_2, i) \overset{k_2}{=} m$, $at_\rho(s'_1, i) \overset{k'_1}{=} l'$, $at_\rho(s'_2, i) \overset{k'_2}{=} m'$, $n = l\ nop\ m$, $n' = l'\ nop\ m'$, $k_1, k_2 < k$, $k'_1, k'_2 < k'$. We can apply the inductive hypothesis and get the thesis.

(∥): By cases on the parity of i.

i **even:** We have $s_1 \parallel s_2 \approx_\rho s'_1 \parallel s'_2$ and $s_1 \approx_\rho s'_1$. Moreover, we have $at_\rho(s_1 \parallel s_2, i) \overset{k}{=} n$, $at_\rho(s'_1 \parallel s'_2, i) \overset{k'}{=} n'$ and, since these judgments have been necessarily derived by rule (AT-∥-EVEN), $at_\rho(s_1, \frac{i}{2}) \overset{k-1}{=} n$, $at_\rho(s'_1, \frac{i}{2}) \overset{k'-1}{=} n'$. We apply the inductive hypothesis and get the thesis.

i **odd:** This case is symmetrical to the one above.

(TAIL-R) We have $s_0 \approx_\rho s_1\hat{}$ and $s_0 \approx_\rho s'_1$, where $\mathsf{Tail}_\rho(s_1) = s'_1$. Furthermore, we have $at_\rho(s_0, i) \overset{k_1}{=} n$, and, by Theorem 14.1, $at_\rho(s_1\hat{}, i) \overset{k_2}{=} n'$ implies $at_\rho(s'_1, i) \overset{k'_2}{=} n'$.

If $k'_2 < k_2$, then we can directly conclude by inductive hypothesis; otherwise, by Theorem 14.1, $k'_2 = k_2$ (hence $at_\rho(s'_1, i) \overset{k_2}{=} n'$) and s'_1 is **not** of shape tail or variable. In this case the proof proceeds by showing that further backward steps of the derivation can be obtained only by applying specific rules, to show that the inductive hypothesis can be applied. Because s'_1 is not of shape tail or variable, the only applicable rules for $s_0 \approx_\rho s'_1$ are (TAIL-L), (VAR-L), (CONS) and (OP). Except for (TAIL-L), for the other rules the derivation steps that make the inductive hypothesis applicable are those already proved for the corresponding cases (VAR-L), (CONS) and (OP), by virtue of the equality $k'_2 = k_2$. For instance, if s_0 is a variable x (rule (VAR-L)), then we have $\rho(x) \approx_\rho s'_1$; furthermore $at_\rho(\rho(x), i) \overset{k_1-1}{=} n$, because $at_\rho(x, i) \overset{k_1}{=} n$ is necessarily derived by (AT-VAR). We can conclude by inductive hypothesis since $(k_1 - 1, k'_2) < (k_1, k'_2) = (k_1, k_2)$.

In case of (TAIL-L) we proceeds similarity as done for (TAIL-R): We have $s_2\hat{} \approx_\rho s'_1$ and $s'_2 \approx_\rho s'_1$, where $\mathsf{Tail}_\rho(s_2) = s'_2$. Furthermore, $at_\rho(s'_1, i) \overset{k'_2}{=} n'$, with $k'_2 = k_2$, and, by Theorem 14.1, $at_\rho(s_1\hat{}, i) \overset{k_1}{=} n'$ implies $at_\rho(s'_1, i) \overset{k'_1}{=} n'$. If $k'_1 < k_1$, then we can directly conclude by inductive hypothesis; otherwise, by Theorem 14.1, $k'_1 = k_1$ (hence $at_\rho(s'_2, i) \overset{k_1}{=} n'$) and s'_2 is **not** of shape tail or variable. If both s'_1 and

$$
\dfrac{
  \dfrac{
    \dfrac{\dfrac{\text{stuck}}{0\!:\!x \approx_\rho (0\!:\!x) \parallel x}\ (??)}{0\!:\!x \approx_\rho x}\ (\text{VAR-R})
    \quad
    \dfrac{\dfrac{}{x \approx_\rho x}\ (\text{VAR})}{x \approx_\rho (0\!:\!x)\hat{}}\ (\text{TAIL-R})
  }{(0\!:\!x) \parallel x \approx_\rho x \parallel (0\!:\!x)\hat{}}\ (\text{OP})
  \quad
  \dfrac{\mathsf{Tail}_\rho((0\!:\!x) \parallel x) = x \parallel (0\!:\!x)\hat{}}{\mathsf{Tail}_\rho(x) = x \parallel (0\!:\!x)\hat{}}\ (\text{TAIL-VAR})
}{
  \dfrac{(0\!:\!x) \parallel x \approx_\rho x\hat{}}{x \approx_\rho x\hat{}}\ (\text{VAR-L})
}\ (\text{TAIL-R})
$$

Note: the premise $\mathsf{Tail}_\rho((0\!:\!x) \parallel x) = x \parallel (0\!:\!x)\hat{}$ is labelled (TAIL-$\parallel$) and has no premises shown.

FIGURE 14.4 $x \approx_\rho x\hat{}$ with $\rho = \{x \mapsto (0\!:\!x) \parallel x\}$

s_2' are **not** of shape tail or variable $s_2' \approx_\rho s_1'$ can only be derived with rules (CONS) and (OP); by virtue of the qualities $k_1' = k_1$ and $k_2' = k_2$ the inductive hypothesis can be made applicable as already proved for the corresponding cases.

(TAIL-L) This case is symmetrical to the one above.

□

Note that the claim of Theorem 14.2 is rather general because it covers also the case where the stream expressions s_0 and s_0' are only partially well-defined. Consider for instance $0\!:\!x$ and $0\!:\!y$ in the environment $\rho = x \mapsto x, y \mapsto y$.

PARTIAL COMPLETENESS Completeness of the equality check roughly means that we should be able to prove that two streams are equal by using the rules in Figure 14.1 whenever they "semantically" correspond to the same stream. In our calculus, this result does not hold in general since, for instance, if we consider the equality $x \approx_\rho x\hat{}$ in the environment $\rho = \{x \mapsto (0\!:\!x) \parallel x\}$ we have a stuck derivation (see Figure 14.4) even if x and $x\hat{}$ both represent the stream constantly equal to 0 (for space reasons we have omitted the derivation for $\mathsf{Tail}_\rho(0\!:\!x) = x$). It is easy to see that a similar stuck path in the derivation is obtained if rule (TAIL-R) is applied before (VAR-L).

The following theorem proves a partial result of completeness by restricting the calculus to only stream built by using variables, the constructor and the tail operators. This shows that for functions as `incr_reg` defined at the beginning of this chapter cycle detection always succeeds.

THEOREM 14.3 (Relative completeness of equality): Let s_0, s_0' and ρ be s.t. they can only contain variables, the constructor and the tail operators. If for all $i \geq 0$ $at_\rho(s_0, i) = at_\rho(s_0', i)$, then $s_0 \approx_\rho s_0'$.

Proof: The proof proceeds by conduction on the rules defining $s_0 \approx_\rho s_0'$ and by case analysis on s_0 and s_0'.

CASE $s_0 = x$ By hypothesis, for all $i \geq 0$ $at_\rho(x, i) = at_\rho(s_0', i)$ and necessarily (AT-VAR) has been used, therefore, for all $i \geq 0$ $at_\rho(\rho(x), i) = at_\rho(s_0', i)$, hence we conclude by rule (VAR-L).

CASE $s'_0 = x$ Symmetric to the previous one.

CASE $s_0 = s_1\hat{}$ By hypothesis, for all $i \geq 0$ $at_\rho(s_1\hat{}, i) = at_\rho(s'_0, i)$; necessarily (AT-TAIL) has been used, therefore, for all $i \geq 0$ $at_\rho(s_1, i+1) = at_\rho(s'_0, i)$, therefore by Theorem 14.4 there exists s' s.t. $\mathsf{Tail}_\rho(s_1) = s'$, and by Theorem 14.1 for all $i \geq 0$, $at_\rho(s_1\hat{}, i) = n$ implies $at_\rho(s', i) = n$, hence, for all $i \geq 0$ $at_\rho(s', i) = at_\rho(s'_0, i)$, hence we can conclude by (TAIL-L).

CASE $s'_0 = s'_1\hat{}$ Symmetric to the previous one.

CASE $s_0 = n : s_1$, $s'_0 = n' : s'_1$ By hypothesis, for all $i \geq 0$ $at_\rho(n : s_1, i) = at_\rho(n' : s'_1, i)$; necessarily (AT-CONS-0) (when $i = 0$) and (AT-CONS-SUCC) (when $i > 0$) have been used, therefore, $n = n'$ and for all $i \geq 0$ $at_\rho(s_1, i) = at_\rho(s'_1, i)$, hence we conclude by rule (CONS).

□

14.3 Towards an algorithm for equality

In this section our aim is to show that for well-defined streams, derivations for the coinductive inference system in Figure 14.1 are always regular and the tail can always be computed symbolically. This is an important step to show that an algorithm for equality can be driven by the rules in Figure 14.1. To do so we need to address two separate issues: for well-defined streams prove that (1) function Tail always terminates and (2) all derivation trees for $s_0 \approx_\rho s'_0$ only involve a finite set of pairs $s_1 \approx_\rho s_2$.

The following theorem proves the first part of our thesis.

THEOREM 14.4 : For all $i \geq 0$, if $at_\rho(s_0, i) = n$, then there exists s' s.t. $\mathsf{Tail}_\rho(s_0) = s'$.

Proof: Assume that $at_\rho(s_0, i) \overset{k}{=} n$. The proof is by induction on k and case analysis on the shape of s_0.

CASES $n : s$, s_1 *nop* s_2, $s_1 \parallel s_2$ In these cases the thesis trivially holds because the claim can be obtained from the corresponding axioms which are always applicable.

CASE x By hypothesis, we have that $at_\rho(x, i) \overset{k}{=} n$; by rule (AT-VAR) $at_\rho(\rho(x), i) \overset{k-1}{=} n$ is necessarily derived and, by inductive hypothesis, there exists s' s.t. $\mathsf{Tail}_\rho(\rho(x)) = s'$, and from this we have the thesis by applying rule (VAR).

CASE $s\hat{}$ By hypothesis, we have that $at_\rho(s\hat{}, i) \overset{k}{=} n$; by rule (AT-TAIL) $at_\rho(s, i+1) \overset{k-1}{=} n$ is necessarily derived and, by inductive hypothesis, there exists s' s.t. $\mathsf{Tail}_\rho(s) = s'$ and $at_\rho(s', i) \overset{k'}{=} n$ by Theorem 14.1.

If $k' < k$, then by inductive hypothesis there exists s'' s.t. $\mathsf{Tail}_\rho(s') = s''$ and we can conclude the thesis by applying rule (TAIL). Otherwise,

$$\text{(VAR-MAXTAILS-AX)}\frac{}{\mathsf{V} \vdash \mathsf{max_tails}_\rho(x) = 0} \quad \begin{array}{l} x \in \mathsf{V} \text{ or} \\ x \notin dom(\rho) \end{array}$$

$$\text{(VAR-MAXTAILS)}\frac{\mathsf{V} \cup \{x\} \vdash \mathsf{max_tails}_\rho(\rho(x)) = k}{\mathsf{V} \vdash \mathsf{max_tails}_\rho(x) = k} \quad x \notin \mathsf{V}$$

$$\text{(OP-MAXTAILS)}\frac{\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_1) = k_1 \quad \mathsf{V} \vdash \mathsf{max_tails}_\rho(s_2) = k_2}{\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_1\ op\ s_2) = \max(k_1, k_2)}$$

$$\text{(CONS-MAXTAILS)}\frac{\mathsf{V} \vdash \mathsf{max_tails}_\rho(s) = k}{\mathsf{V} \vdash \mathsf{max_tails}_\rho(n : s) = k} \qquad \text{(TAIL-MAXTAILS)}\frac{\mathsf{V} \vdash \mathsf{max_tails}_\rho(s) = k}{\mathsf{V} \vdash \mathsf{max_tails}_\rho(s\hat{}) = k + 1}$$

$$\text{(VAR-BINOPS-AX)}\frac{}{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(x) = 0} \quad \begin{array}{l} x \in \mathsf{V} \text{ or} \\ x \notin dom(\rho) \end{array}$$

$$\text{(VAR-BINOPS)}\frac{\mathsf{V} \cup \{x\} \vdash \mathsf{bin_ops}_\rho(\rho(x)) = k}{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(x) = k} \quad x \notin \mathsf{V}$$

$$\text{(OP-BINOPS)}\frac{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_1) = k_1 \quad \mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_2) = k_2}{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_1\ op\ s_2) = k_1 + k_2 + 1}$$

$$\text{(CONS-BINOPS)}\frac{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s) = k}{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(n : s) = k + 1} \qquad \text{(TAIL-BINOPS)}\frac{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s) = k}{\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s\hat{}) = k}$$

FIGURE 14.5 Auxiliary functions

by Theorem 14.1 s' is **not** of shape tail or variable, hence, as already shown above for those cases, there exists s'' s.t. $\mathsf{Tail}_\rho(s') = s''$ since the corresponding axiom can be always applied, therefore also when $k' \nless k$ the thesis follows by applying rule (TAIL).

□

The second step for our aim is to prove that all derivation trees for $s_0 \approx_\rho s'_0$ involve only a finite set of pairs $s_1 \approx_\rho s_2$. In this way, we are guaranteed to never incur into non-termination.

Figure 14.5 shows the definitions of two functions we will use for the proof.

Function max_tails computes the number of tail operators in the definition of a stream, while bin_ops does the same with the number of binary operators. Both functions keep track of already processed variables with a set of variables V to terminate the derivation with the axioms (VAR-MAXTAILS-AX) and (VAR-BINOPS-AX).

The following lemmas are needed for the proof of Theorem 14.16, and require

$\mathsf{Tail}_\rho(s_0) = s'$ to be generalized to $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_0) = s'$, with V set of variables.

$$\text{(V-TAIL-CONS)}\frac{}{\mathsf{V} \vdash \mathsf{Tail}_\rho(n : s) = s} \qquad \text{(V-TAIL)}\frac{\mathsf{V} \vdash \mathsf{Tail}_\rho(s) = s' \quad \mathsf{V} \vdash \mathsf{Tail}_\rho(s') = s''}{\mathsf{V} \vdash \mathsf{Tail}_\rho(s\hat{}) = s''}$$

$$\text{(V-VAR-AX)}\frac{}{\mathsf{V} \vdash \mathsf{Tail}_\rho(x) = x}\ x \in \mathsf{V} \qquad \text{(V-VAR)}\frac{\mathsf{V} \cup \{x\} \vdash \mathsf{Tail}_\rho(\rho(x)) = s}{\mathsf{V} \vdash \mathsf{Tail}_\rho(x) = s}\ x \notin \mathsf{V}$$

$$\text{(V-NOP)}\frac{}{\mathsf{V} \vdash \mathsf{Tail}_\rho(s_1[nop]s_2) = s_1\hat{}[nop]s_2\hat{}} \qquad \text{(V-}\|\text{)}\frac{}{\mathsf{V} \vdash \mathsf{Tail}_\rho(s_1 \parallel s_2) = s_2 \parallel s_1\hat{}}$$

LEMMA 14.5 : If $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_0) = s'$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_0) = k$, then $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s') = k'$ and $k' \leq k + 1$.

Proof: By induction on the rules defining $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_0) = s'$.

(V-TAILS-CONS) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(n : s) = s$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(n{:}s) = k$. To derive this latter hypothesis we must have applied rule (CONS-MAXTAILS), so we also know that $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s) = k$. The thesis immediately follows from this fact.

(V-TAIL) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(s\hat{}) = s''$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s\hat{}) = k$. To derive this latter hypothesis we must have applied rule (TAIL-MAXTAILS), so we also know that $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s) = k - 1$; furthermore, by rule (V-TAIL), $\mathsf{V} \vdash \mathsf{Tail}_\rho(s) = s'$, therefore by inductive hypothesis, we know that $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s') = k'$ with $k' \leq k$. Again, by rule (V-TAIL), $\mathsf{V} \vdash \mathsf{Tail}_\rho(s') = s''$, therefore by inductive hypothesis $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s'') = k''$ with $k'' \leq k' + 1 \leq k + 1$, hence we get the thesis.

(V-VAR-AX) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(x) = x$ with $x \in \mathsf{V}$ and by (VAR-MAXTAILS-AX) $\mathsf{V} \vdash \mathsf{max_tails}_\rho(x) = 0$, hence the thesis is immediate.

(V-VAR) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(x) = s$ with $x \notin \mathsf{V}$ and $x \in dom(\rho)$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(x) = k$. To derive this latter hypothesis we must have applied rule (VAR-MAXTAILS), so we also know that $\mathsf{V} \cup \{x\} \vdash \mathsf{max_tails}_\rho(\rho(x)) = k$. By rule (V-VAR) $\mathsf{V} \cup \{x\} \vdash \mathsf{Tail}_\rho(\rho(x)) = s$ therefore by inductive hypothesis, we have $\mathsf{V} \cup \{x\} \vdash \mathsf{max_tails}_\rho(s) = k'$ with $k' \leq k + 1$, and, thus, the thesis.

(V-NOP) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_1[nop]s_2) = s_1\hat{}[nop]s_2\hat{}$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_1[nop]s_2) = k$. To derive this latter hypothesis we must have applied rule (OP-MAXTAILS), so we also know that $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_1) = k_1$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_2) = k_2$ with $\max(k_1, k_2) = k$. By rules (TAIL-MAXTAILS) and (OP-MAXTAILS) we have $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_1\hat{}[nop]s_2\hat{}) =$

$\max(k_1+1, k_2+1) \leq \max(k_1, k_2)+1 \leq k+1$, hence the thesis.

(V-‖) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_1 \parallel s_2) = s_2 \parallel s_1\hat{}$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_1 \parallel s_2) = k$. To derive this latter hypothesis we must have applied rule (OP-MAXTAILS), so we also know that $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_1) = k_1$ and $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_2) = k_2$ with $\max(k_1, k_2) = k$. By rules (TAIL-MAXTAILS) and (OP-MAXTAILS) we have $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s_2 \parallel s_1\hat{}) = \max(k_2, k_1+1) \leq \max(k_1, k_2)+1 \leq k+1$, hence the thesis.

□

LEMMA 14.6 : If $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_0) = s'$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_0) = k$, then $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s') = k'$ and $k' \leq k$.

Proof: By induction on the rules defining $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_0) = s'$.

(V-TAILS-CONS) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(n : s) = s$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(n : s) = k$. To derive this latter hypothesis we must have applied rule (CONS-BINOPS), so we also know that $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s) = k-1$. The thesis immediately follows from this fact.

(V-TAIL) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(s\hat{}) = s''$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s\hat{}) = k$. To derive this latter hypothesis we must have applied rule (TAIL-BINOPS), so we also know that $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s) = k$; furthermore, by rule (V-TAIL), $\mathsf{V} \vdash \mathsf{Tail}_\rho(s) = s'$, therefore by inductive hypothesis, we know that $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s') = k'$ with $k' \leq k$. Again, by rule (V-TAIL), $\mathsf{V} \vdash \mathsf{Tail}_\rho(s') = s''$, therefore by inductive hypothesis $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s'') = k''$ with $k'' \leq k' \leq k$, hence we get the thesis.

(V-VAR-AX) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(x) = x$ with $x \in \mathsf{V}$ and by (VAR-BINOPS-AX) $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(x) = 0$, hence the thesis is immediate.

(V-VAR) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(x) = s$ with $x \notin \mathsf{V}$ and $x \in dom(\rho)$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(x) = k$. To derive this latter hypothesis we must have applied rule (VAR-BINOPS), so we also know that $\mathsf{V} \cup \{x\} \vdash \mathsf{bin_ops}_\rho(\rho(x)) = k$. By rule (V-VAR) $\mathsf{V} \cup \{x\} \vdash \mathsf{Tail}_\rho(\rho(x)) = s$ therefore by inductive hypothesis, we have $\mathsf{V} \cup \{x\} \vdash \mathsf{bin_ops}_\rho(s) = k'$ with $k' \leq k$, and, thus, the thesis.

(V-NOP) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_1[nop]s_2) = s_1\hat{}[nop]s_2\hat{}$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_1[nop]s_2) = k$. To derive this latter hypothesis we must have applied rule (OP-BINOPS), so we also know that $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_1) = k_1$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_2) = k_2$ with $k = k_1+k_2+1$. By rules (TAIL-BINOPS) and (OP-BINOPS) we have $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_1\hat{}[nop]s_2\hat{}) = k_1+k_2 \leq k_1+k_2+1 = k$, hence the thesis.

(V-‖) By hypothesis, we have $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_1 \parallel s_2) = s_2 \parallel s_1\hat{}$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_1 \parallel s_2) = k$. To derive this latter hypothesis we must have applied rule (OP-BINOPS), so we also know that $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_1) = k_1$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_2) = k_2$ with $k = k_1+k_2+1$. By rules (TAIL-BINOPS) and

(OP-BINOPS) we have $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s_2 \parallel s_1\hat{}) = k_1 + k_2 \leq k_1 + k_2 + 1 = k$, hence the thesis.

□

LEMMA 14.7 : If $\mathsf{Tail}_\rho(x) = s'$ then rule (VAR) cannot be applied for variable x in the derivation of $\mathsf{Tail}_\rho(\rho(x)) = s'$.

Proof: We first observe that for any term s_0 there exists a unique applicable rule for deriving $\mathsf{Tail}_\rho(s_0) = s'$, therefore there cannot exist two different derivations for $\mathsf{Tail}_\rho(s_0) = s'$. From this and the fact that derivations must be finite (because the definition of $\mathsf{Tail}_\rho(s_0) = s'$ is inductive) we can deduce that, for any path in the derivation of $\mathsf{Tail}_\rho(\rho(x)) = s'$, rule (VAR) cannot be applied for the same variable x, otherwise the path would not be finite. □

LEMMA 14.8 : Let V be a set of variables; if $\mathsf{Tail}_\rho(s_0) = s'$ is derivable by applying rule (VAR) only for variables not in V, then $\mathsf{V} \vdash \mathsf{Tail}_\rho(s_0) = s'$ is derivable.

Proof: The proof proceeds by induction on the rules for $\mathsf{Tail}_\rho(s_0) = s'$; the only non trivial case is for (VAR), that is, when $s_0 = x$. By Lemma 14.7 rule (VAR) cannot be applied for variable x in the derivation of $\mathsf{Tail}_\rho(\rho(x)) = s'$, therefore by inductive hypothesis $\mathsf{V} \cup \{x\} \vdash \mathsf{Tail}_\rho(s_0) = s'$ is derivable, and, hence, we can conclude $\mathsf{V} \vdash \mathsf{Tail}_\rho(x) = s'$ by rule (V-VAR), because by hypothesis we have $x \notin \mathsf{V}$. □

COROLLARY 14.9 : If $\mathsf{Tail}_\rho(s_0) = s'$ then $\emptyset \vdash \mathsf{Tail}_\rho(s_0) = s'$.

Proof: A direct consequence of Lemma 14.8. □

COROLLARY 14.10 : If $\mathsf{Tail}_\rho(s_0) = s'$ and $\emptyset \vdash \mathsf{max_tails}_\rho(s_0) = k$, then $\emptyset \vdash \mathsf{max_tails}_\rho(s') = k'$ and $k' \leq k + 1$.

Proof: Directly from Lemma 14.5 and Lemma 14.8. □

COROLLARY 14.11 : If $\mathsf{Tail}_\rho(s_0) = s'$ and $\emptyset \vdash \mathsf{bin_ops}_\rho(s_0) = k$, then $\emptyset \vdash \mathsf{bin_ops}_\rho(s') = k'$ and $k' \leq k$.

Proof: Directly from Lemma 14.6 and Lemma 14.8. □

If $\emptyset \vdash \mathsf{max_tails}_\rho(s) = k_1$ and $\emptyset \vdash \mathsf{bin_ops}_\rho(s) = k_2$, then we use the notation $\mathsf{max_tails}_\rho(s)$ and $\mathsf{bin_ops}_\rho(s)$ to denote k_1 and k_2, respectively, coherently with the fact that the two judgments define a total function from stream values to natural numbers for any set of variables V. This follows directly from the fact that, by definition, for any V and s, there is always a unique applicable rule for $\mathsf{V} \vdash \mathsf{max_tails}_\rho(s) = _$ and $\mathsf{V} \vdash \mathsf{bin_ops}_\rho(s) = _$.

DEFINITION 14.12 : For any s_1, s_2, ρ, the measure of $s_1 \approx_\rho s_2$, denoted by $\mu_\rho(s_1, s_2)$, is defined as follows:

$\mu_\rho(s_1, s_2) = (k_1, k_2)$ where
$k_1 = \max(\{\mathsf{max_tails}_\rho(s_i) \mid i = 1, 2\} \cup \{\mathsf{max_tails}_\rho(\rho(x)) \mid x \in dom(\rho)\})$
$k_2 = \max(\{\mathsf{bin_ops}_\rho(s_i) \mid i = 1, 2\} \cup \{\mathsf{bin_ops}_\rho(\rho(x)) \mid x \in dom(\rho)\})$

The following theorem shows that the measure of each node $s_1 \approx_\rho s_2$ in a possibly infinite derivation for $s_0 \approx_\rho s_0'$ is bounded by the measure of the root $s_0 \approx_\rho s_0'$, where component-wise order is considered: $(k_1, k_2) \leq (k_1', k_2')$ iff $k_1 \leq k_1'$ and $k_2 \leq k_2'$.

THEOREM 14.13 : If $s_0 \approx_\rho s_0'$, then $\mu_\rho(s_1, s_2) \leq \mu_\rho(s_0, s_0')$ for all $s_1 \approx_\rho s_2$ in the derivation of $s_0 \approx_\rho s_0'$.

Proof: By induction on the length of the path in the derivation from the root $s_0 \approx_\rho s_0'$ to the node $s_1 \approx_\rho s_2$ and by case analysis on the applied rule for the root. For rules (TAIL-L) and (TAIL-R) the claims on $\mathsf{Tail}_\rho(s) = s'$ in Corollary 14.10 and Corollary 14.11 are exploited. □

The following lemma and theorem prove that the set of nodes $s_0 \approx_\rho s_0'$ with measure bounded by a constant is always finite, under the assumption that s_0 and s_0' are built over a fixed finite set of variables.

LEMMA 14.14 : Let V be a fixed finite set of variables, ρ an environment, and k_1, k_2 two natural numbers; then the set $\{s \mid (\mathsf{max_tails}_\rho(s), \mathsf{bin_ops}_\rho(s)) = (k_1, k_2), s \text{ built on } \mathsf{V}\}$ is finite.

Proof: By general induction over (k_1, k_2).

- $(k_1, k_2) = (0, 0)$: if s contains at least a tail operator, then by definition $\mathsf{max_tails}_\rho(s) > 0$, whereas if it contains at least another operator, then by definition $\mathsf{bin_ops}_\rho(s) > 0$, therefore s can only be a variable and the thesis follows from the hypothesis on V.
- $(k_1, k_2) > (0, 0)$: the proof for the inductive step proceeds by case analysis on the shape of s.
  - variable: the set of terms s.t. s is a variable must be finite, again by hypothesis on V;
  - tail: let us consider the set $S_1 = \{s \mid (\mathsf{max_tails}_\rho(s), \mathsf{bin_ops}_\rho(s)) = (k_1, k_2), s = s_0\hat{\ }, s \text{ built on } \mathsf{V}\}$; by definition, $\mathsf{max_tails}_\rho(s_0) = k_1 - 1$ and $\mathsf{bin_ops}_\rho(s_0) = k_2$, therefore by generalized inductive hypothesis we deduce that the set of terms $\{s_0 \mid (\mathsf{max_tails}_\rho(s_0), \mathsf{bin_ops}_\rho(s_0)) = (k_1 - 1, k_2), s_0 \text{ built on } \mathsf{V}\}$ is finite, therefore S_1 is finite as well;
  - constructor: let us consider the set $S_2 = \{s \mid (\mathsf{max_tails}_\rho(s), \mathsf{bin_ops}_\rho(s)) = (k_1, k_2), s = n : s_0, s \text{ built on } \mathsf{V}\}$; by definition, $\mathsf{max_tails}_\rho(s_0) = k_1$ and $\mathsf{bin_ops}_\rho(s_0) = k_2 - 1$, therefore we deduce that S_2 is finite similarly as done for the previous case;

– other binary operators: let us consider the set $S_3 = \{s \mid (\mathsf{max_tails}_\rho(s), \mathsf{bin_ops}_\rho(s)) = (k_1, k_2), s = s_1\ op\ s_2, s \text{ built on } \mathsf{V}\}$; by definition, $\mathsf{max_tails}_\rho(s_i) = k_1^i \leq k_1$ and $\mathsf{bin_ops}_\rho(s_i) = k_2^i < k_2$, $i = 1, 2$, therefore by generalized inductive hypothesis we deduce that the sets of terms $\{s_i \mid (\mathsf{max_tails}_\rho(s_i), \mathsf{bin_ops}_\rho(s_i)) = (k_1^i, k_2^i), s_i \text{ built on } \mathsf{V}\}$, are finite for all $(k_1^i, k_2^i) < (k_1, k_2)$, $i = 1, 2$. Since all possible pairs (k_1^i, k_2^i) are finite because bounded by (k_1, k_2) we deduce that the set of all possible subterms s_1, s_2 of s is finite, therefore S_3 is finite as well.

We have partitioned the set $\{s \mid (\mathsf{max_tails}_\rho(s), \mathsf{bin_ops}_\rho(s)) = (k_1, k_2), s \text{ built on } \mathsf{V}\}$ into four sets depending on the shape of s and proved that all such sets are finite, therefore we can conclude the claim.

□

THEOREM 14.15 : Let V be a fixed finite set of variables, ρ an environment, and k_1, k_2 two natural numbers; then the set $\{s_1 \approx_\rho s_2 \mid \mu_\rho(s_1, s_2) \leq (k_1, k_2), s_1, s_2 \text{ built on } \mathsf{V}\}$ is finite.

Proof: A direct consequence of Lemma 14.14, and of the facts that V, the domain of ρ and the set $\{(k_1', k_2') \mid (k_1', k_2') \leq (k_1, k_2)\}$ are finite, and the finite union of finite sets is finite. □

THEOREM 14.16 : Any derivation for $s_0 \approx_\rho s_0'$ involves only a finite set of pairs $s_1 \approx_\rho s_2$.

Proof: A direct consequence of Theorem 14.13 and Theorem 14.15. □

15

Related and future work

In the second part of the thesis, we focused on the paradigmatic example of streams to enhance regular corecursion in order to overcome the limitations imposed by regular terms. Moreover, we provided a well-definedness procedure to guarantee safe access to streams, as well as an algorithm to check stream equality. Proposals in the existing literature to tackle these problems are outlined in the following, together with other papers related to streams.

STREAM CALCULI In the context of streams, our main sources of inspiration for the operators considered in the calculus and some examples were the papers of Rutten [28, 27], where a coinductive calculus of streams of real numbers is defined, and Hinze [20], where a calculus of generic streams is defined in a constructive way and implemented in Haskell. The work presented in the thesis differs from all the approaches mentioned above since, in our case, the aim is to provide an *operational* semantics, designed to directly lead to an implementation. That is, even though streams are infinite objects (terms where the constructor is the only operator, defined coinductively), evaluation handles their *finite* representations, and is defined by an *inductive* inference system.

BEYOND REGULAR TERMS The main disadvantage of regular corecursion with respect to lazy evaluation is that only regular structures are supported. For instance, given the function definition `from(n)= n : from(n+1)`, a call like, e.g., `member(10,from(0))` will not terminate in our calculus, since with call-by-value semantics the subterm `from(0)` should be evaluated. There have been a few proposals to allow the manipulation of infinite non-regular values. Notably, in the context of logic programming, *structural resolution* [25, 24] (a.k.a. S-resolution) is a proposed generalization for cases when formulas computable at infinity are not regular; infinite derivations that cannot be built in finite time are generated lazily, and only partial answers are shown. In particular, recent results [12] investigate how it is possible to integrate co-LP cycle detection into S-resolution, by proposing a comprehensive theory to provide operational semantics that go beyond loop detection.

Another approach is the work by Courcelle [15], introducing algebraic trees and equations as generalizations of regular ones. Such proposals share, even though with different techniques and in a different context, our aim of extending regular corecursion; on the other hand, the fact that corecursion is *checked* is, in our knowledge, a novelty of our work.

WELL-DEFINEDNESS OF STREAMS The problem of ensuring well-defined corecursive definitions has been also considered in the context of type theory and proof assistants. We have shown in Chapter 10 that simple guarded definitions [14] do not work properly in case values are allowed to contain non constuctors as the tail operator; a more complex approach based on a type system has been proposed by Sacchini [29] for an extension of the calculus of constructions which is more expressive than that considered here; however, as opposed to what happens with the judgment *wd* defined in Chapter 10, corecursive calls to the result of an application of tail are never well-typed even in case of well-defined streams, as happens for the definition of `fib` as given in Section 9.2.

D'Angelo et al. presented LOLA [19], a specification language for runtime monitoring that manipulates streams. The general idea behind the framework is to generate a set of output streams, starting from a given set of input streams. Input streams describe the values of the system under observation and output streams represent errors or reports of the monitor. Stream expressions are constructed starting from constants, stream variables, function symbols and, in particular, an operator to define a stream by starting from another one shifted by an offset; in this case, a default value must be specified because the offset might lead past the end or before the beginning of the stream. There are a few differences with respect to our approach, especially due to the fact that, while we allow streams to be infinite, in LOLA only finite ones are considered. However, there are some similarities with our work. In this framework, well-definedness is checked by relying on a dependency graph, which keeps track of relations between the processed streams. The vertices of this graph are the streams, while the edges represent the dependencies between them. Each edge is weighted with a value ω to point the fact that a stream depends on another one shifted by ω positions. Then, the well-definedness constraint is that each closed-walk inside the graph must have a total weight different from 0. These syntactic constraints appear to be very similar to the approach we used for predicate *wd*, in which we impose the number of occurrences of the constructor operator to be greater than the number of occurrences of the tail operator.

FUTURE WORK A first objective for future work is to enrich the language by adding more data types and constructs, e.g., an `if-then-else` with a stream of booleans as condition and two streams in the branches. Additional interesting operators on streams could be added, such as the filtering ones, which would greatly increase the expressive power of the calculus. Each new operator should be thoroughly studied, in order to be dealt with by the well-definedness and equality checks, as done for the interleaving operator in Chapter 12 and Chapter 13. Another interesting direction is the design of a static type system to filter out early errors. Notably, our well-definedness check takes place at runtime. The introduction of a static check would filter out certain non-well-defined streams beforehand, thus reducing the overhead on the check at runtime.

A possible direction to optimize the semantics could be to handle regular streams in a special way. Indeed, such kind of streams can be finitely represented by their first few digits and period, and with just this information we can compute functions like *at* in constant time. To achieve this goal, the semantics should be able to identify regular streams and treat them differently, for example by applying special rules designed for optimizing performance.

Conclusion

16

Conclusion

In this thesis we introduced two extensions of the framework of regular corecursion in orthogonal directions.

In the first part we presented coFJ, a Java-like calculus which supports *flexible* regular corecursion, by equipping methods with a codefinition to be evaluated when a cyclic call is found.

The key contributions of this part are the following:

- The design of a novel programming style, smoothly incorporating support for cyclic data structures and coinductive reasoning in the object-oriented paradigm.
- The first, in our knowledge, operational model supporting detection of function/method cyclic calls which is shown to be *sound* with respect to an abstract semantics.
- The evidence, thanks to the intermediate semantics, that the issues of providing a finite representation for (some) infinite objects, and the one of detecting cyclic calls, are independent.

In the second part of the thesis we presented a stream calculus which enhances regular corecursion *beyond regular terms*.

The key contributions of this part are the following:

- The design of a language semantics where equations representing infinite terms can contain other operators besides the constructor. In this way, we can define also non-regular streams (like the one of natural numbers) without loosing the possibility of inspecting a regular stream in its entirety.
- The formalization of a well-definedness check for streams, a sound procedure to filter out ill-formed streams at runtime. Thanks to this, we achieve a convenient trade-off between expressive power and reliability.
- The formalization of an equality check on streams, useful to achieve two important objectives: (1) to provide an equality operator which can be used by the programmer and (2) to detect equal calls in more cases, so to improve cycle detection.

16.1 Future work

We already discussed future work for both parts of the thesis in Chapter 8 and Chapter 15. Here we just report some keypoints:

PART I

- Provide a fully-fledged implementation of `coFJ`.
- Guarantee type soundness, statically ensuring that an undetermined value never occurs as receiver of field access or method invocation.
- Integrate regular corecursion with the notion of mutable state.

PART II

- Investigate additional operators to further increase the expressive power of the language.
- Design a static type system to prevent runtime errors such as the non-well-definedness of a stream.
- Reason on optimizations of the semantics when dealing with regular streams.

Finally, a natural goal for future work is to integrate the two objectives of the thesis, e.g., by enhancing the stream calculus in Part II with *flexible* corecursive definitions, as done with `coFJ` in Part I. This extension would give users the possibility to define specific behaviour when a cycle is detected. For instance, assuming to also allow boolean results for functions, we could define the predicate `allPos`, checking that all the elements of a stream are positive, specifying as result `true` when a cycle is detected; in this way, e.g., `allPos(one_two())` would return the correct result.

Bibliography

[1] Peter Aczel. An Introduction to Inductive Definitions. In: *Handbook of Mathematical Logic*. Edited by Jon Barwise. Vol. 90. Studies in Logic and the Foundations of Mathematics. Elsevier, 1977, pp. 739 –782. Cited on p. 15.

[2] Davide Ancona. Regular corecursion in Prolog. In: *Computer Languages, Systems & Structures* 39.4 (2013), pp. 142–162. Cited on p. 61.

[3] Davide Ancona, Pietro Barbieri, Francesco Dagnino, and Elena Zucca. Sound Regular Corecursion in coFJ. In: *ECOOP'20 - Object-Oriented Programming*. Edited by Robert Hirschfeld and Tobias Pape. Vol. 166. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 1:1–1:28. DOI: 10.4230/LIPIcs.ECOOP.2020.1. Cited on pp. 11, 37, 107.

[4] Davide Ancona, Pietro Barbieri, and Elena Zucca. Enhanced Regular Corecursion for Data Streams. In: *ICTCS'21 - Italian Conf. on Theoretical Computer Science*. 2021. Cited on p. 11.

[5] Davide Ancona, Pietro Barbieri, and Elena Zucca. Enhancing expressivity of checked corecursive streams. In: *Functional and Logic Programming (FLOPS 2022)*. 2022. Cited on p. 11.

[6] Davide Ancona, Francesco Dagnino, and Elena Zucca. Extending Coinductive Logic Programming with Co-Facts. In: *First Workshop on Coalgebra, Horn Clause Logic Programming and Types, CoALP-Ty'16*. Edited by Ekaterina Komendantskaya and John Power. Vol. 258. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2017, pp. 1–18. DOI: 10.4204/EPTCS.258.1. Cited on pp. 29, 61, 62.

[7] Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing Inference Systems by Coaxioms. In: *26th European Symposium on Programming, ESOP 2017*. Edited by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 29–55. DOI: 10.1007/978-3-662-54434-1_2. Cited on pp. 15, 16, 37.

[8] Davide Ancona and Agostino Dovier. A Theoretical Perspective of Coinductive Logic Programming. In: *Fundamenta Informaticae* 140.3-4 (2015), pp. 221–246. Cited on pp. 7, 61.

[9] Davide Ancona and Elena Zucca. Corecursive Featherweight Java. In: *FTfJP'12 - Formal Techniques for Java-like Programs*. ACM Press, 2012, pp. 3–10. Cited on pp. 9, 61, 62.

[10] Davide Ancona and Elena Zucca. Safe corecursion in coFJ. In: *FTfJP'13 - Formal Techniques for Java-like Programs*. ACM Press, 2013, p. 2. Cited on pp. 61, 62.

[11] Pietro Barbieri, Francesco Dagnino, and Elena Zucca. An inductive abstract semantics for coFJ. In: *FTfJP'20 - Formal Techniques for Java-like Programs.* ACM Press, 2020. Cited on p. 11.

[12] Henning Basold, Ekaterina Komendantskaya, and Yue Li. Coinduction in Uniform: Foundations for Corecursive Proof Search with Horn Clauses. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings.* 2019, pp. 783–813. Cited on p. 123.

[13] Luca Ciccone, Francesco Dagnino, and Elena Zucca. Flexible Coinduction in Agda. In: *International Conference on Interactive Theorem Proving, ITP 2021.* Edited by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 13:1–13:19. DOI: 10.4230/LIPIcs.ITP.2021.13. Cited on pp. 7, 8.

[14] Thierry Coquand. Infinite Objects in Type Theory. In: *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers.* 1993, pp. 62–78. DOI: 10.1007/3-540-58085-9_72. Cited on pp. 6, 70, 76, 124.

[15] Bruno Courcelle. Fundamental Properties of Infinite Trees. In: *Theoretical Computer Science* 25 (1983), pp. 95–169. DOI: 10.1016/0304-3975(83)90059-2. Cited on pp. 85, 123.

[16] Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. In: *Logical Methods in Computer Science* 15.1 (2019). DOI: 10.23638/LMCS-15(1:26)2019. Cited on pp. 15, 16, 37.

[17] Francesco Dagnino. *Flexible Coinduction.* PhD thesis. DIBRIS, University of Genova, 2021. Cited on pp. 15, 16, 44.

[18] Francesco Dagnino, Davide Ancona, and Elena Zucca. Flexible coinductive logic programming. In: *Theory and Practice of Logic Programming* 20.6 (2020). Issue for ICLP 2020, pp. 818–833. DOI: 10.1017/S147106842000023X. Cited on pp. 8, 61, 62.

[19] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime Monitoring of Synchronous Systems. In: *12th International Symposium on Temporal Representation and Reasoning (TIME 2005).* 2005, pp. 166–174. DOI: 10.1109/TIME.2005.26. Cited on p. 124.

[20] Ralf Hinze. Concrete Stream Calculus: An Extended Study. In: *Journal of Functional Programming* 20.5–6 (2010), 463–535. DOI: 10.1017/S0956796810000213. Cited on p. 123.

[21] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In: *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999.* ACM Press, 1999, pp. 132–146. DOI: 10.1145/320384.320395. Cited on pp. 15, 17.

[22] Jean-Baptiste Jeannin and Dexter Kozen. Computing with Capsules. In: *Journal of Automata, Languages and Combinatorics* 17.2-4 (2012), pp. 185–204. DOI: 10.25596/jalc-2012-185. Cited on pp. 9, 29, 61, 62, 69.

[23] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Functional Programming with Regular Coinductive Types. In: *Fundamenta Informaticae* 150 (2017), pp. 347–377. Cited on pp. 9, 29, 61.

[24] Ekaterina Komendantskaya, Patricia Johann, and Martin Schmidt. A Productivity Checker for Logic Programming. In: *Logic-Based Program Synthesis and Transformation - LOPSTR 2016, Revised Selected Papers*. Edited by Manuel V. Hermenegildo and Pedro López-García. Vol. 10184. Lecture Notes in Computer Science. Springer, 2016, pp. 168–186. DOI: 10.1007/978-3-319-63139-4_10. Cited on p. 123.

[25] Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic logic programming: from Semantics to Implementation. In: *J. Log. Comput.* 26.2 (2016), pp. 745–783. DOI: 10.1093/logcom/exu026. Cited on p. 123.

[26] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. In: *Information and Computation* 207.2 (2009), pp. 284–304. DOI: 10.1016/j.ic.2007.12.004. Cited on p. 15.

[27] Jan Rutten. *The Method of Coalgebra: exercises in coinduction*. C, 2019. Cited on p. 123.

[28] Jan J. M. M. Rutten. A coinductive calculus of streams. In: *Mathematical Structures in Computer Science* 15.1 (2005), pp. 93–147. DOI: 10.1017/S0960129504004517. Cited on p. 123.

[29] Jorge Luis Sacchini. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In: *Symposium on Logic in Computer Science, LICS 2013*. 2013, pp. 233–242. DOI: 10.1109/LICS.2013.29. Cited on p. 124.

[30] Luke Simon. *Extending logic programming with coinduction*. PhD thesis. University of Texas at Dallas, 2006. Cited on pp. 7, 29, 61.

[31] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-Logic Programming: Extending Logic Programming with Coinduction. In: *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*. Edited by Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki. Vol. 4596. Lecture Notes in Computer Science. Springer, 2007, pp. 472–483. DOI: 10.1007/978-3-540-73420-8_42. Cited on pp. 7, 61.