University of Milan

Department of Computer Science and Technology

Program Transformations in the Delay Monad

A Case Study for Coinduction via Copatterns and Sized Types

Supervisor

Prof. Alberto Momigliano

A thesis submitted by

Edoardo Marangoni, num. 973597

for the degree of

Master of Science

Academic Year 2022-2023

"...I can hardly understand, for instance, how a young man can decide to ride over to the next village without being afraid that, quite apart from accidents, even the span of a normal life that passes happily may be totally insufficient for such a ride."

Franz Kafka

NO GENERATIVE ARTIFICIAL INTELLIGENCE WAS USED IN THIS WORK.

Content

CHAPTER I

Introduction

Our objective is to define an *operational semantics* for an *imperative language* targeting an adequate *monad* to model the desired *effects* and operate *transformations* on program sources, all in a *dependently typed proof assistant*. This work, in a nutshell, is a case study to analyse how all these techniques work when put together.

In this section we will introduce some conventions used throughout the work (Section 1.1). Then, we will explain what it means to define the *semantics* of a language and why such an effort is useful (Section 1.2). We will then turn to a bird-eye view of Agda (Section 1.3), which is the tool we used to mechanize all the definitions and proofs in the thesis.

1.1 Notational conventions

Inference rules

We will make use of inference rules in the form

$$\frac{A}{\mathrm{c}}$$

where A is the set of *antecedents* and c is the *conclusion*. Intuitively, such a rule means that if the *judgments* in A are true, then c is true. If $A = \emptyset$, the rule $\frac{\emptyset}{c}$ identifies an axiom.

Code snippets

We will make great use of code snippets. Snippets coming from different sources are indentified by different colours.

```
This code is from Agda's standard library
```

snippet 1.1.1 from Agda's stdlib

```
This code has no relevant source
snippet 1.1.2
```

```
This code is part of the thesis
see code  snippet 1.1.3
```

1.2 Semantics

Semantics, in general, is a tool that has the objective to assign meaning to the execution of programs in a certain programming language. In the literature, many formal model of semantics appear: *denotational semantics*, *operational semantics*, *axiomatic semantics*, *action semantics*, *algebraic semantics*, *functorial semantics* and many more [1].

We will explore **operational** semantics in particular. This formalism, which appeared for the first time in the definition of the semantics of Algol 68; in general, operational semantics express the meaning of a program in a way that directly reflects the execution of the program in a reference system. The formalism of operational semantics has multiple flavours itself.

Small step operational semantics, introduced by Plotkin in [2] and also known as *structural* (or *structured*) operational semantics, is expressed inductively and, as the name suggests, structurally, as a sequence of finite steps. For example, consider the language composed of natural numbers n and fully-parenthesized sums $e := (e_1 + e_2)$, where the result of the computation (the *values*) are natural numbers (the result of the sum).

We can express the rules of the small-step semantics of our toy language in the form of inference rules as shown in Semantics 1. Notice that the + operator is *overloaded*, as it appears both as a function between expressions of the language and natural numbers. In words, for example, term $((1 + 1) + 1)$ would step to $((2) + 1)$, then $(2 + 1)$ and finally 3:

$$((1 + 1) + 1) \underset{\text{rule 5}}{\rightarrow} ((2) + 1) \underset{\text{rule 2}}{\rightarrow} (2 + 1) \underset{\text{rule 5}}{\rightarrow} 3$$

$$\frac{e_1 \to e_{1'}}{(e_1 + e_2) \to (e_{1'} + e_2)} \quad (1) \qquad \frac{e_1 \to n_1}{(e_1 + e_2) \to (n_1 + e_2)} \quad (2)$$

$$\frac{e_2 \to e_{2'}}{(n_1 + e_2) \to (n_1 + e_{2'})} \quad (3) \qquad \frac{e_2 \to n_2}{(n_1 + e_2) \to (n_1 + n_2)} \quad (4)$$

$$\frac{n_1 + n_2 \equiv n}{(n_1 + n_2) \to n} \quad (5)$$

Semantics 1: Small-step rules for sums

Big step operational semantics, introduced by Kahn in [3] and also known as *natural* operational semantics, puts in relation the final result of the evaluation of a program term and the term itself (without intermediate steps). Taking again the previous example, we can express the big-step semantics of this language as shown in Semantics 2 and the computation of $((1 + 1) + 1)$ expressed with such rules would be simply $((1 + 1) + 1) \Rightarrow 3$.

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n_1 + n_2 \equiv n}{(e_1 + e_2) \Rightarrow n}$$

Semantics 2: Big-step semantics for sums

The two semantics have different advantages and disadvantages. Leroy, in [4], claims that small-step semantics is more expressive and preferred for some objectives such as proving the soundness of a type system, while big-step semantics is to be preferred for some other ends, such as proving the correctness of program transformations.

1.2.1 Program transformations

Program transformations are the core of our work. A program transformation is an operation that changes in some way an input program in some source language in another program in a target language. Examples of program transformations are the static analysis of the source code such as *constant folding*, *dead code elimination*, *register allocation*, *liveness analysis* and many more [5]. The kind of transformations just cited are *source to source*, that is, the transformation is a function from the input language to a program in the same language.

Another important example of program transformation are *compilers* for which, in general, we often take the correctness for granted: in this case the transformation outputs, generally, a result in a different language, for example assembly code for an input in the C language; this kind of transformations, for which the output language is different from the input one, are known as *source to target*.

Consider the LLVM compiler infrastructure [6]: it is composed of hundreds of thousands of lines of code and performs various transformations from its own intermediate representation, starting with a translation of the program in SSA form, continuing with tens of (optional) optimizing transformations and concluding with a last translation into a target language. In principle, we do not have a formal assurance – a **proof** – that no transformation ever changes the meaning of the input program, and the user has to trust the programmers and the community (altough efforts have been put to verify at least parts of LLVM: [7], [8], [9]).

Having a formal statement that proves that the transformations operated on a program do not change the semantics of the source language is obviously a much desired feature, and many efforts in the literature have been in this direction. One of the most notable ones is CompCert [10], which has the objective of providing a formalized backend for (almost all of) the ISO C standard by providing a compiler where the majority of transformations (all, if we do not consider lexical analysis and printing to ASM as transformations) are either programmed in Caml or programmed and proved in Coq [11].

To this end, having a formal definition of the semantics of a language is the first step to prove the correctness of the transformations operated on programs in that language; in general, the idea is to prove that the transformation does not change the result of the execution of the transformed program. This means that the *observable* behaviour of the program is not changed by the transformations: for textbook examples, these behaviours are often the termination, non-termination or crash of the program when executed. In realistic languages, observable behaviours can be also inputs and outputs.

It is clear how finding the suitable definition of the semantics for the job is an important task but, as noted in [4] and [12], it can be fairly difficult. The reason for this difficulty is that we ideally want a representation that is able to capture every detail of the semantics of the program but also be lightweight enough to allow proofs and definitions to be streamlined.

As stated in [12], we could consider expressing the semantics of a language as an inductive relation, either small-step or big-step, showing how and when the evaluation of a program converges to a result. With this mechanism, we either lose the explicit meaning of diverging (non-terminating) and failing programs or we shall add new rules for both diverging and failing programs, inducing a multiplication of rules that can be unreasonable for large languages.

Furthermore, a functional definition (an interpreter) of the semantics expressed in this fashion should have type

```
eval(Program, Context) -> Fails or Diverges or Converges
```

But such an interpreter, clearly, is impossible to implement in a total constructive language, as this amounts to solving the halting problem.

As suggested by Danielsson in [12], we explore the implementation and use of a functional semantics targeting the coinductively-defined `Delay` monad (which will be studied in details in later chapters) to represent non-termination, failure and termination in a concise fashion. In this way, the semantics is an interpreter and its type signature does not imply that we have to solve the halting problem.

Now that we have an intuition of what our goal is, we move to the explaination of the tool we used, Agda.

1.3 Agda

Agda is a programming language and a proof assistant. Its development goes back to 1999 where a first version was written by Catarina Coquand [13]; in 2005 Ulf Norell worked on a redesign [14], which laid the foundations for what Agda is today. In this section, we begin introducting what proof assistants are and what their objectives are, and after that we will introduce specific details of Agda.

1.3.1 Proof assistants

This introduction to proof assistants follows [15]. As the name suggests, a proof assistant has the role of providing aid to the user in the context of *proofs*, so that a user can set up a mathematical theory, define properties and do logical reasoning with them. A mathematical proof can in principle be reduced to a sequence of small steps each of which can be verified simply and irrefutably. The role of a proof is twofold: one is to *convince* the reader that the statement the proof is about is correct, and the other is to *explain* why the statement is correct.

A mechanzed tool can be helpful to verify that each small step in a proof is correct, thus convince the reader that the whole proof is correct, and one role of a proof assistant is exactly that: a *proof checker*. Of course, the proof checker itself must be reliable and convincing: to this end, one may have an independent description of the logic underlying the tool, that is the set of axioms and inference rules (the *kernel*) that are implemented in the checker. Also, the correctness of the checker itself can be verified as well, proving that the checker can verify a theorem φ if and only if φ is derivable from the independent kernel.

The Curry-Howard isomorphism

The relation between logic and computer science is deep and has a long history, and a full account of the historical events that occurred in the literature is not the objective of this work: we choose, instead, to report the fundamental discoveries and inventions. The work of Alonzo Church in the 1930s led to the invention of the λ-calculus, a formal system able to express computations and functions, while later extensions added a type system. Almost two decades after the inventions of Church, Haskell Curry noticed that the rules forming types in the λ-calculus can be seen as logical rules [16]; finally, William Howard realized in [17] that intuitionistic natural deduction can be seen as a typed variant of the λ-calculus.

Intuitionistic (or constructive) logic, as intended in the Brouwer–Heyting–Kolmogorov interpretation (see [18], [19], [20]), postulates that each proof must contain a *witness*: for example, a proof of $P \wedge Q$ is a pair $< a, b >$ where a is a proof of P and b is a proof of Q; a proof of $P \Rightarrow Q$ is a function f that converts a proof of P into a proof of Q and so on: we can see already here a connection between a formal interpretation of logic and the usual type system and programming languages we use daily.

These ideas together led to the **Curry-Howard correspondence** (or Curry-Howard isomorphism), which essentially says that a proposition is identified with the type (which we can se as a collection) of all its proofs, and a type is identified with the proposition there exists a term of that type (so that each of its terms is in turn a proof of the corresponding proposition). This, in concrete, leads to correspondence shown in Table 3.

By the time the correspondence appeared formally, many advancements were already available in the world of proof assistants: for example, the Automath proof checker introduced by de Bruijn in 1967 [21] included many notions that reappeared later in the literature such as *dependent types*.

Martin-Löf type theory

In 1972, Per Martin-Löf extended the ideas in the Curry-Howard isomorphism introducing a new *type theory* known as intuitionistic type theory (or constructive type theory or, simply, Martin-Löf type theory, MLTT). This theory internalizes the concepts of intuitionistic (or constructive) logic as intended in the Brouwer–Heyting–Kolmogorov interpretation. Many extensions and versions have been proposed in the literature: the first version was shown to be inconsistent by Girard, and later revision were made consistent (removing the *impredicativity* that caused the inconsisency) and

Logic	Type Theory
proposition	type
predicate	dependent type
proof	term/program
true	unit type
false	empty type
conjunction	product type
disjunction	sum type
implication	function type
negation	function type into empty type
universal quantification	dependent product type
existential quantification	dependent sum type
equality	identity type

Table 3: Curry-Howard correspondence between logic and type theory

introduced inductive and universe types. Every flavour of MLTT has dependent types which, as shown in Table 3, are used to build types that are equivalent to universal and existential quantifiers in predicate logic.

MLTT introduced many concepts and it is, as of today, the backbone of many proof assistants, but it is not the only type theory available. Another example is the Calculus of Construction (and its variants) proposed by Coquand [22], which is the theory underlying the Coq proof assistant. Another is the more recent *homotopy type theory* [23].

Termination and consistency

The logical system on which the proof assistant lies must respect strict constraints. Perhaps one the most important is that every term – which, as explained above, is also a proof – must be provably terminating. This is necessary to keep the consistency of the system itself and avoid proofs of $\bot$, the type that has no constructor and thus cannot, by definition, be possibly built, which in turn means that it cannot be proven.

This reasoning is also important in a setting where types as well depend on arbitrary terms: for example, what would be the type of a type depending on the value of an infinite loop?

Allowing non-terminating terms and non-well-founded recursive definitions, a proof of $\bot$ can be immediate: for example, by defining a term $x := x + 1$ one can easily come up with a proof of $0 = 1$. This requirement (together with the requirement of productivity for coinductive definitions, as we will see in later chapters), is shared between any kind of proof assistant. We will examine these concepts in more details in Chapter 2.

We conclude this section introducing proof assistants describing where Agda sits in relation to what we just discussed. Agda is a proof assistant and a programming language (at the time of writing, it even compiles to Javascript) based on a flavour of MLTT, where termination (and productivity) of definitions is enforced for the reasons cited above. We proceed, now, describing in details its inner workings, starting from a lightweight introduction to its syntax.

1.3.2 Syntax

The first thing to highlight in relation to the syntax of Agda is that every character including unicode codepoints and "," is a valid identifier, except "(" and ")". The character "_" has a special meaning and allows the definition of mixfix operators which can be used in multiple ways, as shown in snippet 1.3.1. For example, `3::2::1::[]` is lexed as an identifier, and we must use spaces to make Agda parse it as we may intend it (a list). These are both valid identifiers as well `this+is*a-valid[identifier]`, `this,as⇒well`.

```
(if_then_else_) x y z
if x then y else z
(if x then_else_) y z
(if_then y else z) x
(if x then _ else z) y
```

snippet 1.3.1

1.3.3 Type system

As anticipated, Agda is based on a flavour of Martin-Löf type theory and as every MLTT it has dependent types. As explained above, this allows the user to embed semantic informations about the programs at the type level and represent logical statements in a computer program. In this subsection, we briefly describe how types can be defined in Agda.

Data types

One of the simplest datatypes is that of Boolean values. Agda's standard library defines them as shown in snippet 1.3.2.

```
data Bool : Set where
  false : Bool
  true : Bool
```

snippet 1.3.2 from Agda's stdlib see code

The `data` keyword is used to introduce new datatypes in the program. The type system allows for more complex definitions, as prescribed by the logical system it is based on. For example, we can define polymorphic lists as follows:

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : (x : A) (xs : List A) → List A
```

snippet 1.3.3

In this example, where the `List` type is parameterized by the type `A`, we can already see a glimpse of the power of Agda's type system, which will also be explored in more depth when examining the definition of functions.

Levels

The fundamental type in Agda is `Set`, which we used in the previous examples without giving a detailed description. `Set` is the sort of *small* types [24], but not every type in Agda is a `Set`, however; to avoid paradoxes similar to that of Russel, Agda uses universe levels and provides an infinite number of them.

We thus have that `Set` is not of type `Set`, instead it is $\texttt{Set} : \texttt{Set}_1$ and, in turn, it is $\texttt{Set}_1 : \texttt{Set}_2, \ldots : \texttt{Set}_n$, where the subscript `n` is its **level**. In principle, we have that `Set` is implicitly `Set 0`. A type whose elements are types themselves is called a *sort* or *universe* [24].

It is interesting to underline that Agda's type system allows *universe polymorphism*, allowing the user to parameterize or index definitions on the unverse level as well, as shown in snippet 1.3.4.

```
data List {a} (A : Set a) : Set a where
  []  : List A
  _∷_ : (x : A) (xs : List A) → List A
```

snippet 1.3.4 from Agda's stdlib see code

Records

Another example of instrumentation Agda proposes to define datatypes are **records**.

From Agda's documentation: *"Records are types for grouping values together. They generalise the dependent product type by providing named fields and (optional) further components."* [24].

```
record Pair {a b} (A : Set a) (B : Set b) : Set (a ⊔ b) where
  field
    fst : A
    snd : B
```

snippet 1.3.5

An example of record is shown in snippet 1.3.5, defining the type for pairs with type polymorphism and universe polymorphism. This definition automatically inserts in scope three new functions: one to create a `Pair` and two to access its fields: we will examine this briefly, after having introduced the concept of *functions* in Agda.

1.3.4 Functions

From the syntactic point of view function definitions are syntactically similar to those in Haskell, following an equational style defining *clauses*. The similarity with Haskell stops here, as typing rules in Agda are not similar to those in Haskell, which uses a completely different type system.

```
not : Bool → Bool
not false = true
not true = false
```

snippet 1.3.6

```
id : ∀ {A : Set} → A → A
id a = a
```

snippet 1.3.7

By the Curry-Howard isomorphism, types are univocally related to propositions and function definitions are univocally linked to proofs. We can see this in snippet 1.3.7 where, in code, we define a polymorphic function that for any parameter A outputs a result of type A; its definition is just returning the parameter. We can interpret this in logic as follows: `∀ {A : Set} → A → A` *is* the proposition $\forall A : \text{Set}, A \implies A$, while `id a = a` *is* the proof, in λ-calculus, $\lambda x.x$.

We now explore in more depth the use of Agda as a proof assistant. A part of the usefulness of Agda is its interaction with the user through *holes*, which indicate a term that the programmer does not have (conceptually) available yet; Agda aids the programmer showing graphically the type of the hole: we demonstrate both this aspect and the capacities of Agda in the definition of proofs with an example.

Suppose we want to encode in Agda the following logical statement:

$$\forall b : \text{Bool}, b \vee \text{false} \equiv \text{false}$$

In Agda, we can represent this statement as follows:

```
v-identityʳ : ∀ (b : Bool) → b v false ≡ b
v-identityʳ false = refl
v-identityʳ true = refl
```

snippet 1.3.8

The previous example also shows *the proof* for the statement: with pattern matching on the value of `b` we can prove this simply by using the reflexivity of the built-in propositional equality. To give a slightly more involved example to show other uses of Agda we define the function in snippet 1.3.9 that appends two `List`s (see the definition of `List`s in snippet 1.3.3).

```
_++_ : List A → List A → List A
[]       ++ ys = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)
```

snippet 1.3.9 from Agda's stdlib see code

Suppose, now, that we want to prove the following statement:

$$\forall l : \text{List}, [] ++ l \equiv l$$

that is, that the empty list `[]` is the (right) identity of the append operator. In Agda:

```
++-identityʳ : ∀ {A : Set} (l : List A) → l ++ [] ≡ l
++-identityʳ [] = refl
++-identityʳ (x ∷ l) rewrite (++-identityʳ l) = refl
```

snippet 1.3.10

Snippet 1.3.10 shows the use of another tool offered by Agda: the `rewrite`. This keyword allows the programmer to extend Agda's evaluation relation with new computation rules [24]. In practice, this means that given an evidence that $x \equiv y$, `rewrite` rules allow to change evidences involving y to evidences using x. Consider the previous example: we had to prove that `(x ∷ l) ++ [] ≡ x ∷ l`. By the definition of _++_ in snippet 1.3.9, the term `(x ∷ l) ++ []` is *normalized* to `x ∷ (l ++ [])`, which means that we must show

```
x ∷ (l ++ []) ≡ x ∷ l
```

Instructing Agda to rewrite `++-identityʳ l`, which is a proof of `l ++ [] ≡ l`, means syntactically changing the occurrences of `l ++ []` with occurrences of `l`, which leaves us to prove that `x ∷ l ≡ x ∷ l`, which is easily done by reflexivity.

Copatterns

Going back to the example shown in snippet 1.3.5, the record definition automatically defines a constructor

```
Pair : ∀ {a b} (A : Set a) (B : Set b) → Set (a ⊔ b)
```

and two projection functions

```
Pair.fst : ∀ {a b} {A : Set a} {B : Set b} → Pair A B → A
Pair.snd : ∀ {a b} {A : Set a} {B : Set b} → Pair A B → B
```

Elements of `Pair` can be constructed using the extended notation or using *copatterns*, as shown in snippet 1.3.11.

```
-- Extended notation
p34 : Pair ℕ ℕ
p34 = record {fst = 3; snd = 4} --
-- Copatterns
-- Prefix notation
p34 : Pair ℕ ℕ
Pair.fst p34 = 3
Pair.snd p34 = 4
-- Postfix notation
p34 : Pair ℕ ℕ
p34 .Pair.fst  = 3
p34 .Pair.snd  = 4
```

snippet 1.3.11

Dot patterns

A dot pattern (also called inaccessible pattern) can be used when the only type-correct value of the argument is determined by the patterns given for the other arguments. A dot pattern is not matched against to determine the result of a function call. Instead it serves as checked documentation of the only possible value at the respective position, as determined by the other patterns. The syntax for a dot pattern is `.t`.

As an example, consider the datatype `Square` defined as follows

```
data Square : ℕ → Set where
  sq : (m : ℕ) → Square (m * m)
```

snippet 1.3.12

Suppose we want to define a function `root : (n : ℕ) → Square n → ℕ` that takes as its arguments a number n and a proof that it is a square, and returns the square root of that number. We can do so as follows:

```
root : (n : ℕ) → Square n → ℕ
root .(m * m) (sq m) = m
```

snippet 1.3.13

CHAPTER 2

Recursive datatypes and proofs

All throughout this work we make use of the mathematical technique called *coinduction*. It is far from easy to come up with an intuitive and contained explaination for this technique, as coinduction is a pervasive topic in computer science and mathematics and can be explained with different flavours and intuitions: in category theory as coalgebras, in automata theory and formal languages as a tool to compare infinite languages and automata execution, in real analysis as greatest fixed points, in computer science as infinite datatypes and corecursion and much more.

We examine this last possibility, as our use of coinduction is "limited" to coinductive datatypes and proofs by corecursion. We begin introducing induction both as a mathematical tool and in computer science, in particular in Agda. We then move to coinduction and its mathematical intepretation; we conclude this chapter with an explaination of how recursive definitions are handled in proof assistants such as Agda.

2.1 Induction

The easiest and most intuitive inductive datatype is that of natural numbers. In Agda, one may represent them as shown in snippet 2.1.1.

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

snippet 2.1.1

A useful interpretation of inductive datatypes is to imagine concrete instances as trees reflecting the structure of the constructors, as shown in Figure 4; of course, this interpretation is not limited to *degenerate* trees and it can be used to represent any kind of inductive structure such as lists (which shall be binary trees), trees themselves and so on.

$$\begin{array}{c} \text{zero} \\ \downarrow \\ \text{succ zero} \\ \downarrow \\ \text{succ succ zero} \\ \downarrow \\ \dots \end{array}$$

Figure 4: Structure of a natural number as tree of constructors

Our interest is not limited to the definition of inductive datatypes, we also want to prove their properties: the correct mathematical tool to do so is the *principle of induction*, which has been implicitly used, in history, since the medieval times of the Islamic golden age, even if some works, such as [25], claim that Plato's Parmenide contained implicit inductive proofs already. Its modern version, paired with an explicit formal meaning, goes back to the foundational works of Boole, De Morgan and Peano in the 19th century.

Suppose we want to prove a property P of natural numbers. This property can be, for example, Theorem 2.1.1.

Theorem 2.1.1 For all natural numbers n, the sum of the first n powers of 2 is $2^n - 1$, or

$$\forall n, \sum_{i=0}^{n-1} 2^i \equiv 2^n - 1$$

For the sake of the explaination, we give a proof of Theorem 2.1.1 in a discursive manner, so that we are able to delve into each step.

A proof by induction begins by proving a base case (typically 0 or 1, but it is not necessarily always the case); we choose to prove it for $n = 0$: the sum of the first 0 powers of 2 is 0, and $2^0 - 1 = 1 - 1 = 0$, therefore the case base is proved.

The power of the induction principle shows up here. The prover now assumes that the principle holds for every number up to n (this is called the *inductive hypothesis*) and using this fact, which in this case is that $\sum_{i=0}^{n-1} 2^i = 2^n$, the prover shows that the statement holds for $n + 1$ as well:

$$\sum_{i=0}^{n} 2^i = \sum_{i=0}^{n-1} 2^i + 2^n = 2^n + 2^n - 1 = 2^{n+1} - 1$$

The principle of induction is not limited to natural numbers. Every recursive type has an *elimination principle* which prescribes how to use terms of such type that entails a **structural recursive definition** and a **principle of structural induction**. This, in turn, implies that there exists a well-founded relation inducing a well-order on the terms of the type: a well-founded relation assures that, given a concrete instance of an inductive term, analysing its constructor tree we will eventually reach a base case (`zero` in the case of natural numbers, `nil` in the case of lists, the root node in case of trees). Such a relation implies that a proof that examines a term in a descending manner will eventually terminate.

There are many cases in which, however, we might want to express theorems about structures that are not well-founded. A simple example of this is the following: consider the infinite sequence of natural numbers

$$s \stackrel{\text{def}}{=} 0, 1, 2, 3, 4, \ldots$$

The sequence s certainly is a mathematical object that we can show theorems about: for example, we might want to show that there is no element that is greater than any other, but how are we to define such an object using induction? An idea might be that of using lists as defined in snippet 2.1.2.

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

snippet 2.1.2

However, concrete terms which we can actually build up cannot be infinite; instead, they must be a finite sequence of applications of constructors. In other words, the tree of constructors of a concrete list we can come up with is necessarily of bounded (finite) height.

We could try to trick Agda to define a potentially infinite sequence such as s as shown in snippet 2.1.3. Then, we could represent s as `infinite-list 0`.

```
 infinite-list : Nat → List Nat
 infinite-list n = n ∷ (infinite-list (n + 1))
-- Termination checking failed for the following functions:
--   infinite-list
-- Problematic calls:
--   infinite-list (n + 1)
--     (at /home/ecmm/t.agda:28,25-38)
```

snippet 2.1.3

It turns out, however, that such a definition is not acceptable for Agda's termination checker. One may argue that this is Agda's fault and that, for example, Haskell may be completely fine with such a definition (and it is indeed, as it employs a completely different strategy with regard to termination checking). In the end, such a definition is indeed *recursive*.

2.2 Coinduction

However, we must notice that a "fully constructed" infinite list such as s does not have a base case and a possible inductive definition cannot be well-founded. It turns out, then, that it is induction itself that can be inadequate to reason about some infinite structures. It is important to remark, however, that in general it is not problematic to reason about infinite structures, and it is not infinity per sé that makes induction an inadequate tool.

What induction does is build potentially infinite objects starting from constructors. **Coinduction**, on the other hand, allows us to reason about potentially infinite objects by describing how to observe (or destruct) them, as opposed to how to build them. Following the previous analogy where inductive datatypes were seen as constructor trees of finite height and functions or inductive proofs operated on the nodes of this tree, we can see coinduction as a means to operate on a tree of potentially infinite height by defining how to extract data from each level of the tree.

While induction has an intuitive meaning and can be explained easily, coinduction is arguably less intuitive and requires more background to grok and, as anticipated in the introduction of this chapter, formal explainations draw inspiration from various and etherogeneous fields of mathematics and computer science.

In this work we do not have the objective to give a formal and thorough explaination of coinduction (which can be found, for example, in works such as [26] and [27]); instead, we will give a contained description of the relation between induction and coinduction, then move to a more formal description using fixed points, with the only objective of providing an intuition of what is the theoretical background of coinduction.

Take again the example of lists as prescribed in snippet 2.1.2. We can express such a definition using inference rules as shown in Table 5.

$$\frac{A : \text{Type}}{\text{nil} : \text{List}\ A}\ \textit{nil} \qquad\qquad \frac{A : \text{Type} \quad xs : \text{List}\ A \quad x : A}{\text{cons}\ x\ xs : \text{List}\ A}\ \textit{cons}$$

Table 5: Inference rules for the polymorphic List type

This inference rules are *satisfied* by some set of values. Suppose that the type A, interpreted as a *set*, is $A := \{x, y, z\}$; an example of a set satisfying the rules in Table 5 is

$$S = \{\text{nil}, \text{cons}\ x\ \text{nil}, \text{cons}\ y\ \text{nil}, \text{cons}\ z\ \text{nil}, \text{cons}\ x\ (\text{cons}\ x\ \text{nil}), \text{cons}\ y\ (\text{cons}\ x\ \text{nil}), \ldots\}$$

The set S is exactly the inductive interpretation of the inference rules: in S there are those elements that follow the rules and those elements only. Among all the sets that satisfy the rules, S is the **smallest** one; however, it is not the only possibility. We could take, for example, the **biggest** set that follows that prescription and has every possible element of the universe in it: of course, such a set, say B, also follows the inference rules in Table 5. The set B is the coinductive interpretation of the inference rules above.

Consider now dropping the rule *nil* from Table 5. The set S, the inductive interpretation, would be the empty set, as no base case is satisfied and there is no "starting point" to build new lists. On the other hand the set B would still contain lots of lists, in particular infinite lists.

2.2.1 Induction and coinduction as fixed points

The mathematical explaination is largely inspired by [28], [29] and [4] and follows a "bottom-up" style of exposition: we start with concepts that have no apparent connection with (co-)induction, and reveal near the end how a specific interpretation of the matter exposed can give an intuition of what coinduction is (as well as a formal definition in a specific field of mathematics).

Let U be a set such that there exists a binary relation $\leq \subseteq U \times U$ that is reflexive, antisymmetric and transitive; we call $< U, \leq >$ a partially ordered set. Note that we concede the possibility for two elements of U to be incomparable without being the same. Formally:

Definition 2.2.1 (Partially ordered set) Let U be a set. U is called a partially ordered set if there exists a relation $\leq \subseteq U \times U$ such that for any $a, b, c \in U$ $\leq$ is

1. **reflexive**: $a \leq a$
2. **antisymmetric**: $a \leq b \land b \leq a \Rightarrow a = b$
3. **transitive**: $a \leq b \land b \leq c \Rightarrow a \leq c$

We call the pair $< U, \leq >$ a partial order.

An example of a partially ordered set is the power set 2^X of any set X with $\leq$ being the usual notion of inclusion, as shown in Figure 6. This partially ordered set is in fact a **lattice** as it has a least element (the empty set $\emptyset$) and a greatest element (the entire set

$U = \{a, b, c\}$). Furthermore, the absence of paths between the sets $\{a, b\}$ and $\{a, c\}$ is an exemplification of the fact that two elements of U may be incomparable.

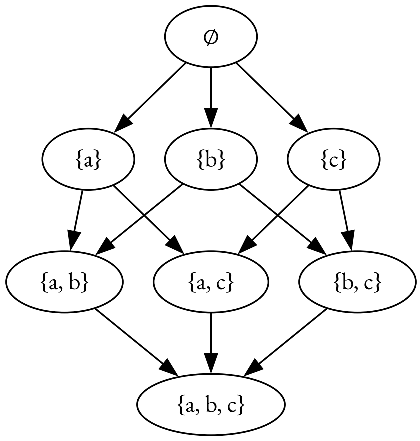

Figure 6: Hasse diagram for the partially ordered set created by the inclusion relation (represented by the arrows) on a set $U = \{a, b, c\}$

Definition 2.2.2 (Lattice) A lattice is a partial order $< U, \leq >$ for which every pair of elements has a greatest lower bound and least upper bound, that is

$$\forall a \in U, \forall b \in U,\ \exists s \in U, \exists i \in U,\ \sup(\{a, b\}) = s \wedge \inf(\{a, b\}) = i$$

We also give a specific infix notation for the sup and inf when applied to binary sets:

$$a \wedge b \stackrel{\text{def}}{=} \sup(\{a, b\}) \text{ and } a \vee b \stackrel{\text{def}}{=} \inf(\{a, b\})$$

which we name meet and join, respectively.

Lattices are defined *complete* if for every subset $L \subseteq U$ there are two elements $\sup(L)$ and $\inf(L)$ such that the first is the smallest element greater than or equal to all elements in L, while the second is the greatest element less than or equal to all elements in L. Formally:

Definition 2.2.3 (Complete lattice) A complete lattice is a lattice for which every subset of the carrier set has a greatest lower bound and least upper bound, that is

$$\forall L \subseteq U,\ \exists s \in U, \exists i \in U,\ \sup(L) = s \wedge \inf(L) = i$$

Complete lattices always have a bottom element

$$\bot \stackrel{\text{def}}{=} \inf(\emptyset)$$

and a top element

$$\top \stackrel{\text{def}}{=} \sup(U)$$

We also give a characterization of a specific kind of functions on U in Definition 2.2.4.

Definition 2.2.4 (Monotone function on complete lattices) Let $< U, \leq >$ be a complete lattice. A function $f : U \rightarrow U$ is monotone if it preserves the partial order:

$$\forall a \in U, \forall b \in U,\ a \leq b \Rightarrow f(a) \leq f(b)$$

We write $[U \rightarrow U]$ to denote the set of all monotone functions on $< U, \leq >$.

Definition 2.2.5 Let $< U, \leq >$ be a complete lattice; let X be a subset of U and let $f \in [U \rightarrow U]$. Then we say that

1. X is ***f*-closed** if $f(X) \subseteq X$, that is, the output set is included in the input set;
2. X is ***f*-consistent** if $X \subseteq f(X)$, that is, the input set is included in the output set; and
3. X is a *fixed point* of f if $f(X) = X$.

For example (taken from [29]), consider the following function on $U = \{a, b, c\}$:

$$\begin{array}{cc} e_1(\emptyset) = \{c\} & e_1(\{a,b\}) = \{c\} \\ e_1(\{a\}) = \{c\} & e_1(\{a,c\}) = \{b,c\} \\ e_1(\{b\}) = \{c\} & e_1(\{b,c\}) = \{a,b,c\} \\ e_1(\{c\}) = \{b,c\} & e_1(\{a,b,c\}) = \{a,b,c\} \end{array}$$

There is only one e_1-closed set, $\{a, b, c\}$, and four e_1-consistent sets, $\emptyset, \{c\}, \{b, c\}, \{a, b, c\}$.

Theorem 2.2.1 (Knaster-Tarski) Let U be a complete lattice and let $f \in [U \rightarrow U]$. The set of fixed points of f, which we define $\text{fix}(f)$, is a complete lattice itself. In particular

1. the least fixed point of f (noted μF), which is the bottom element of $\text{fix}(f)$, is the intersection of all f-closed sets.
2. the greatest fixed point of f (noted νF), which is the top element of $\text{fix}(f)$, is the union of all f-consistent sets.

Proof 2.2.1 Omitted.

From the example above, we have that $\mu e_1 = \nu e_1 = \{a, b, c\}$.

Corollary 2.2.1

1. **Principle of induction**: if X is f-closed, then $\mu f \subseteq X$;
2. **Principle of coinduction**: if X is f-consistent, then $X \subseteq \nu f$.

Now that all the mathematical framework is in place, we can make a concrete example.

$$\frac{}{\varepsilon}\ nil \qquad \frac{l}{x :: l}\ cons$$

Table 7: Inference rules for the untyped List type

Consider the rules in Table 7, a semplifications of rules in Table 5: we drop the polymorphism and leave implicit the part of the "is a list" part of the judgment; we also consider x in the cons rule to be any element in the universe. We will show that we can build a complete lattice from the rules in Table 7 and then show that the sets S and B that we defined above are respectively the least fixed point and greatest fixed point of a specific function f on the complete lattice. We can interpret each rule in Table 7 as an *inference system* over a set U of judgements. In this case, we have

$$U \stackrel{\text{def}}{=} \{j_1, j_2, j_3\}$$

where, for ease of exposure, we set $j_1 \stackrel{\text{def}}{=} \varepsilon$, $j_2 \stackrel{\text{def}}{=} l$ and $j_3 \stackrel{\text{def}}{=} x :: l$.

The pair $< 2^U, \subseteq >$ is a complete lattice, and has the structure in Figure 8.

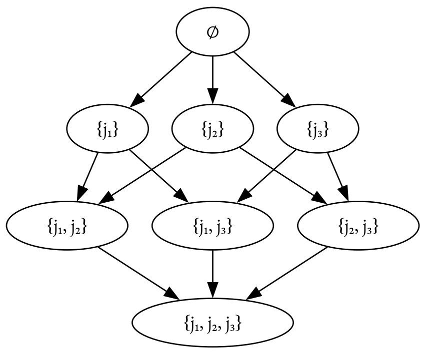

Figure 8: Hasse diagram for the partially ordered set created by the inclusion relation (represented by the arrows) on the set $U = \{j_1, j_2, j_3\}$

We now define another binary relation, $\varphi : 2^U \times U$, that embodies the rules themselves:

$$\varphi \stackrel{\text{def}}{=} \{(\emptyset, j_1), (j_2, j_3)\}$$

Intuitively, if we have a rule $\frac{A}{c}$, then $(A, c) \in \varphi$. We now define a function $f : 2^U \to 2^U$ that represent a single step of derivation starting from a set S of premises using the rules in φ:

$$f(L) \stackrel{\text{def}}{=} \{j_1\} \cup \{c \in U \mid \exists A \subseteq L, (A, c) \in \varphi\}$$

Going back to the expanded notation for rules[1] and adding a special notation for $x :: \varepsilon \stackrel{\text{def}}{=} [x]$, we have, for example,

$$f(\emptyset) = \{\varepsilon\}$$
$$f(\{\varepsilon\}) = \{\varepsilon, [x]\}$$
$$f(\{[x]\}) = \{\varepsilon, x :: [x]\}$$
$$f(\{\varepsilon, [x]\}) = \{\varepsilon, [x], x :: [x]\}$$

These sets are all f-consistents, as we always have $L \subseteq f(L)$. Furthermore, the function f is clearly monotone: $L \subseteq L'$ implies that from L' we will be able to derive at least the same conclusions that we can derive from L, thus $f(L) \subseteq f(L')$.

From Theorem 2.2.1, we know that f has a least fixed point and a greatest fixed point, that is

$$\mu f = \bigcap \{L \mid f(L) \subseteq L\}$$
$$\nu f = \bigcup \{L \mid L \subseteq f(L)\}$$

The first set, the smallest f-closed is the set of terms that can be inductively generated from the rules, while the second set, the largest f-consistent is the set of terms that can be coinductively generated from the rules.

2.3 Recursion in Agda

We have already shown an example of inductive definition and proof by induction in Section 2.1. We continue our exposition of inductive and coinductive datatypes and proofs, taking advantage of the effort to introduce Agda and the practical infrastructure it provides to work with inductive and coinductive proofs and definitions.

2.3.1 Termination

To this end, there are many aspects to take into account. The first is that in Agda *"not all recursive functions are permitted - Agda accepts only these recursive schemas that it can mechanically prove terminating"* [24]. It is important to underline that this is a desired condition and not an hindrance, as it is necessary to keep the consistency of the system, as we explained in Paragraph 1.3.1.3.

We inspect these aspects gradually as we define types and proofs by recursion. The first datatype defined by recursion is that of natural numbers in snippet 2.1.1. Of course, we can also define functions, as shown in snippet 2.3.1, and define properties about such declarations leveraging the dependent type system of Agda.

[1] We do this as we technically do not have $(\{j_1\}, \{j_3\}) \in \wp$, although it is clearly something expressed in the rules.

```
_+_ : Nat → Nat → Nat
zero + n₂ =  n₂
suc n₁ + n₂ = suc (n₁ + n₂)
```

snippet 2.3.1

```
+-idᵣ  : ∀ (n : Nat) → n + zero ≡ n
+-idᵣ zero = refl
+-idᵣ (suc n)
 rewrite (+-idᵣ n) = refl
```

snippet 2.3.2

```
+-sucᵣ : ∀ (n₁ n₂ : Nat)
  → n₁ + suc n₂ ≡ suc (n₁ + n₂)
+-sucᵣ zero n₂ = refl
+-sucᵣ (suc n₁) n₂
 rewrite (+-sucᵣ n₁ n₂) = refl
```

snippet 2.3.3

```
+-comm : ∀ (n₁ n₂ : Nat) → n₁ + n₂ ≡ n₂ + n₁
+-comm zero n₂ rewrite (+-idᵣ n₂) = refl
+-comm (suc n₁) n₂ rewrite (+-sucᵣ n₂ n₁) = cong suc (+-comm n₁ n₂)
```

snippet 2.3.4

Although daunting at first, Agda is a very powerful system. In snippet 2.3.4 we expressed the commutativity of the sum of naturals in a handful of lines: of course, this is something that is well understood and fairly basic, but all the infrastructure assures us that if the definition is accepted, there's no possibility that our proof is wrong[2].

```
monus : Nat → Nat → Nat
monus zero _ = zero
monus (suc x) zero = suc x
monus (suc x) (suc y) = monus x y
```

snippet 2.3.5

```
div : Nat → Nat → Nat
div zero _ = zero
div (suc x) y =
  suc (div (monus x y) y)
```

snippet 2.3.6

However, even if Agda is "a powerful hammer", it comes with its limitations, which we begin to investigate with the example of integer division. The definition in snippet 2.3.6 defines integer division as repeated subtraction: it is acceptable to intuitively say that it is a terminating definition, however Agda's termination checker does not agree and groans:

```
Termination checking failed for the following functions: div
```

Agda's termination checker employs a syntactical analysis to prove the termination of a definition; this means that each recursive call must follow a strict schema: in prac-

[2] Assuming there are no inconsistencies in Agda itself.

tice, this means that the only argument that are allowed in recursive calls are immediate subexpressions or general (but strict) subexpressions [24].

With this limitations, the checker is not able to capture relevant semantic informations in our definition of `div` such as the fact that `monus` decreases in the first parameter, thus making our definition of `div` unacceptable.

Sizes for induction

To overcome the limitations of syntactic termination checking, many authors studied the possibility of using types themselves to allow a more powerful termination checker. Abel, drawing from earlier works such as [30], [31] and [32], proposes in [33] a solution that involves a particular idea, *sizes*. We will see that sizes have applications in coinductive definitions as well, but for now we start by giving an intuition of what sizes are in the context of inductive datatypes (such as natural numbers) and recursive functions (such as `div`).

In the inductive case, the sized approach is conceptually simple: we attach a *size* i to every inductive type D yielding a type D^i, and we check that the size is diminishing in recursive calls [33]. To give a practical understanding of what sizes are, consider again Figure 4. Say that T is the tree representing the structure of a number n, where each node is a constructor: a tree for n will have $n + 1$ nodes, thus the height of the tree T is $n + 1$. In this context, we can understand the concept of size as an upper bound on the height of the tree, therefore a valid size for the tree T (and for n) shall be any size greater than or equal to $n + 1$.

In Agda, sizes are represented as a built-in type `Size`. We will proceed in our discussion gradually, and we start now by defining naturals with a notion of size attached to them, as shown in snippet 2.3.7. Agda, beyond the `Size` type, offers the user other primitives. One of these is the `↑ _`operator, which has type `↑ _ : Size → Size` and is used to compute the successor of a given size i; for any size i it is $i <\uparrow i$.

```
data SizedNat : Size → Set where
  zero : ∀ (i : Size) → SizedNat (↑ i)
  succ : ∀ (i : Size) → SizedNat (i) → SizedNat (↑ i)
```

snippet 2.3.7

Let us examine the definition in snippet 2.3.7 in details. We define `SizedNat` as a type indexed by `Size` with two constructors: `zero` and, as expected, `succ`. As anticipated, we want sizes to be an *upper bound* on the height of the constructor tree, so it is natural that the constructor `zero`, given any size i, constructs a tree with one node only (the constructor `zero`) that has height 1 and is upper bounded by $i + 1$ for any i; the same applies to the constructor `succ`, that for any size i and any other natural that has the upper-bounded height of i builds a constructor tree with one node added (the `succ` constructor) that has height at most $i + 1$.

We consider now the example in snippet 2.3.8, that sheds light on why sizes are an upper bound.

```
monus : ∀ (i : Size) (x : SizedNat i) (y : SizedNat ∞) → SizedNat i
monus .(↑ i) (zero i) y = zero i
monus .(↑ i) (succ i x) (zero .∞) = succ i x
monus .(↑ i) (succ i x) (succ .∞ y) = monus i x y
```

snippet 2.3.8

Snippet 2.3.8 defines the usual *monus* function, also noted as $\dot{-}$ in the literature and already shown (in an unsized version) in snippet 2.3.5 (this definition also uses *dot patterns* – see Paragraph 1.3.4.2).

The first thing to comment on is the size ∞, which indicates an upper bound for terms whose height is unknown. In fact, in this case we don't know what is the size of y: what we care about is that, intuitively, for any x and y, it must be $x \dot{-} y \leq x$. Before, we could not express this property of monus in a way that made it available to the termination checker (which could then use it to prove termination): now, this property is implicitly expressed in the type itself.

We can now define division of natural as repeated subtraction in a way that satisfies Agda's termination checker, as shown in snippet 2.3.9.

```
div : ∀ (i : Size) → (x : SizedNat i) → (y : SizedNat ∞) → SizedNat i
div .(↑ i) (zero i) y = zero i
div .(↑ i) (succ i x) y = succ i (div i (monus i x y) y)
```

snippet 2.3.9

In all the examples we proposed we always made sizes explicit, however Agda's termination checker and type system are mature enough to solve the system of equations and find the correct sizes even if left implicit in the declaration of functions.

2.3.2 Productivity

We said, above, that termination of recursive definition is necessary to keep consistency of the system. When it comes to coinduction and corecursive definitions, another criterion, that of **productivity**, is necessary. In short, productivity means that the corecursive function allows new piece of the output to be visible in finite time [34]. Concretely, using a syntax criterion to enforce productivity, Agda requires that the definition of a corecursive function is such that every recursive call is immediately "under" a (co-)constructor.

The classical example of coinductive datatypes is that of *streams*, which in Agda is implemented as shown in snippet 2.3.10.

```
record Stream (A : Set a) : Set a where
  coinductive
  constructor _∷_
  field
    head : A
    tail : Stream A
```

snippet 2.3.10

This definition is a record paired with the `coindutive` keyword; we can thus understand the fields `head` and `tail` as dual to constructors in inductive definitions, embodying the observational nature of coinductive datatypes. We believe that instead of trying to describe in details every choice of the instrumentation for coinduction offered by Agda, it is better to show the behaviour of the `Stream` datatype with an example.

```
countFrom : Nat → NatStream
head (countFrom x) = x
tail (countFrom x) = countFrom (x + 1)
```

snippet 2.3.11

```
countFrom-at-1 : head (tail (countFrom 0)) ≡ 1
countFrom-at-1 = refl
```

snippet 2.3.12

In snippet 2.3.11 we already see the use of another technique offered by Agda, that is *copatterns*, which, as explained in the documentation, *"[allow] to define the fields of a record as separate declarations, in the same way that we would give different cases for a function"* [24], which where originally thought as a tool in the context of coinductive definitions [35], then adapted to general usage.

The meaning of `countFrom` is given in the example `countFrom-at-1` (snippet 2.3.12): the normalization of its type is

```
head (tail (countfrom 0)) ⇒ head (countFrom (0 + 1)) ⇒ 0 + 1 ⇒ 1
```

and, in words, `countFrom` is an infinite stream starting at some number n that for each observation - the application of a sequence of `tails` followed by a `head` - increments its value depending on the number of `tail` calls in the observation; in other words, it is a representation of the infinite sequence of numbers s we described earlier.

Coinduction, together with copatterns, allows us to write corecursive definitions such as snippet 2.3.13.

```
repeat : Nat → NatStream
head (repeat x) = x
tail (repeat x) = repeat x
```

snippet 2.3.13

As before, not every definition is accepted, even if it may be conceptually fine (in this case depending on what is `F`).

```
repeatF : (NatStream → NatStream) → Nat → NatStream
head (repeatF _ x) = x
tail (repeatF F x) = F (repeatF F x)
```

snippet 2.3.14

The function in snippet 2.3.14 cannot be accepted, as the productivity checker cannot make assumptions on what `F` does to the `NatStream` in input, and groans again:

```
Termination checking failed for the following functions: repeatF
```

Sizes for coinduction

The usefulness of sizes is not limited to prove recursive definitions terminating, in fact, they can be used in the definition of coinductive types.

```
data Stream (A : Set a) (i : Size) : Set a where
  _∷_ : A → Thunk (Stream A) i → Stream A i
```

snippet 2.3.15 from Agda's stdlib see code

We show, in snippet 2.3.15, how Agda's standard library implements *sized* streams at the time of writing; we shall examine it in details in order to introduce all the concepts concerning the use of sizes in the (again, at the time of writing) idiomatic way. The first thing to notice is that `Stream` is not a `record` anymore and does not mention the coinductivity of the type: it is declared as an usual inductive datatype with a constructor `_::_` and is parameterized by a type `A` and a size `i`.

This constructor, which resembles the shape of the `cons` (or precisely `_::_`) constructor of finite lists, takes a term of type `A` as its "head" and a term of type `Thunk (Stream A) i` as its "tail". Of course, in order to understand what this means it is necessary to inspect what `Thunks` are.

```
record Thunk {ℓ} (F : SizedSet ℓ) (i : Size) : Set ℓ where
  coinductive
  field force : {j : Size< i} → F j
```

snippet 2.3.16 from Agda's stdlib see code

Snippet 2.3.16 shows the definition of `Thunk` as it is done in Agda's standard library at the time of writing.

```
SizedSet : (ℓ : Level) → Set (suc ℓ)
SizedSet ℓ = Size → Set ℓ
```

snippet 2.3.17 from Agda's stdlib see code

`Thunks` are parameterized by a level (see Chapter 1.3.3.2), a `SizedSet F` of that level and a `Size`. `SizedSet` is a type that characterizes, as it suggests, the set that are paired with sizes, and its definition is shown in snippet 2.3.17. A `Thunk` has no constructor and only has a field `force` that, given a size `j` of type `Size< i`, that is a size strictly less than `i`, returns an instance of the type `F`.

In words, a `Thunk` is a way to abstract away the coinductive features of a type, embodying its observational nature: taking the definition of the sized `Stream` datatype using `Thunk`, we can define a stream as shown in snippet 2.3.18: to create the stream repeating the term `a` indefinitely, we define it using the constructor `_::_`: the "head" is indeed `a`, while the "tail" is an instance of a `Thunk` as prescribed by the anonymous λ with a postfix projection of the `force` copattern.

Excluded the aspect of tracking sizes, this methodology is exactly the same as that used in eager languages to make computations lazy, simply delaying them with a function call that is executed when needed.

```
repeat : A → Stream A i
repeat a = a ∷ λ where .force → repeat a
-- The same as
-- repeat' : ∀ {i} (n : ℕ) → Stream ℕ i
-- repeat' {i} n = n ∷ xs
--   where
--     xs : Thunk (Stream ℕ) i
--     force xs = repeat' n
-- or, in postfix, xs .force = repeat' n
```

snippet 2.3.18 from Agda's stdlib see code

Snippet 2.3.15 shows the implementation of the `repeat` function as done in Agda's standard library. We can compare this definition with that in snippet 2.3.13, which used copatterns. Copatterns are used in snippet 2.3.15 as well, but are hidden in syntactic sugar and are not relative to the `Stream` itself anymore but to `Thunk`, as explained above.

While for inductive types the size was an upper bound on the height of the constructor tree of a term of that type, for coinductive types sizes represent a lower bound on the *depth* of the potentially infinite tree of coconstructors. Each instance of a coinductive datatype will always have arbitrary (∞) size, but in order to provide well-formed definitions we reason with approximations, that is streams that have a depth i for some arbitrary i [33].

Intuitively, the size of a coinductive datatypes gives a lower bound on the number of times the term can be observed in a productive manner (that is, yielding a result in finite time), it is therefore reasonable that `force`-ing a `Thunk` (thus observing the next piece of the potentially infinite tree) produces a result which has a size j that is striclty smaller than the size i we started with.

When we tried to define the function in snippet 2.3.14, Agda's productivity checker could not accept the definition because it was unaware of what the function F did to its input: was F to observe parts of the stream in input, was it to increase the stream adding coconstructors to its coconstructors tree (thus increasing its then unknown size), or was it to leave the stream untouched? We could not know. Now, with the help of sizes, we can impose restrictions on F such that we surely know that F might increase the

```
repeatF : ∀ {i} (n : ℕ) (F : ∀ {i} → Stream ℕ i → Stream ℕ i)
            → Stream ℕ i
repeatF {i} n F = n ∷ λ where .force {j} → F {j} (repeatF n F)
```

snippet 2.3.19

stream or leave it untouched, but it can't make observations in such a way that leaves the stream with less observations "available", as shown in snippet 2.3.19.

2.3.3 Final considerations on sizes

Sizes give the programmer the ease to write recursive and corecursive functions (thus, in a dependently typed environment such as Agda, also proofs) without the troubles of syntactic termination and productivity checks.

Sizes, however, are not a complete solution to every problem: Agda's issues page on GitHub, at the time of writing, includes 12 issues where the use of sizes makes Agda inconsistent; 7 of these were solved, while 5 are not and one in particular, which allows a proof of ⊥, is marked as being put in the *icebox*, that is, it is an *"Issue [that] there are no plans to fix for upcoming releases."* [36].

There is no official statement regarding the future of sizes in Agda; however, it seems that much effort is being put in the implementation of a *cubical* version of Agda [37], which draws inspiration from [38] and of course [23].

```
record T i : Set₁ where
  coinductive
  field force : (j : Size< i) → Set
open T

data Embed : ∀ i → T i → Set where
  abs : {A : T ∞} → A .force ∞ → Embed ∞ A

app : {A : T ∞} → Embed ∞ A → A .force ∞
app (abs x) = x

Fix' : Size → (Set → Set) → Set
Fix' i F = F (Embed i λ{ .force j → Fix' j F})

data ⊥ : Set where

Omega : Set
Omega = Fix' ∞ (λ A → A → ⊥)

self : Omega
self x = app x x

loop : ⊥
loop = self (abs self)
```

snippet 2.3.20

CHAPTER 3

The delay monad

In this chapter we introduce the concept of monad and then describe a particular kind of monad, the *delay monad*, which will be used troughout the work.

3.1 Monads

In 1989, Eugenio Moggi published a paper [39] in which the term *monad*, which was already used in the context of mathematics and, in particular, category theory, was given meaning in the context of functional programming. Explaining monads is, arguably, one the most discussed topics in the pedagogy of computer science, and tons of articles, blog posts and books try to explain the concept of monad in various ways.

A monad is a datatype equipped with (at least) two functions, `bind` (often `_>>=_`) and `unit`; in general, we can see monads as a structure used to combine computations. One of the most common instance of monad is the `Maybe` monad, which we now present to investigate what monads are: in Agda, the `Maybe` monad is composed of a datatype

```
data Maybe {a} (A : Set a) : Set a where
  just : A → Maybe A
  nothing : Maybe A
```

snippet 3.1.1 from Agda's stdlib

(where `{a}` is the *level*, see Subsection 1.3.3) and two functions representing its monadic features:

```
unit : A → Maybe A
unit = just
_>>=_ : Maybe A → (A → Maybe B) → Maybe B
nothing >>= f = nothing
just a  >>= f = f a
```

snippet 3.1.2 from Agda's stdlib

The `Maybe` monad is a structure that represents how to deal with computations that may result in a value but may also result in nothing; in general, the line of reasoning for monads is exactly this, they are a tool used to model some behaviour of the execution, which is also called **effect**. In the context of programming monads are also “computation builders”.

Consider snippet 3.1.3: this example, even if simple, is a practical application of the line of reasoning a programmer applies when using monads. In this example, we want to simply increment an integer variable which might be, for some reason, unavailable. The `_>>=_` function encapsulates the reasoning that the programmer should make explicit, perhaps matching on the value of `x`, in a compositional and reusable fashion.

```
h : Maybe ℕ → Maybe ℕ
h x = x >>= λ v → just (v + 1)
```

snippet 3.1.3

The underlying idea of monads in the context of computer science, as explained by Moggi in [39], is to describe “notions of computations” that may have consequences comparable to *side effects* in pure functional languages.

3.1.1 Formal definition

We will now give a formal definition of what monads are. They’re usually understood in the context of category theory and in particular *Kleisli triples*; here, we give a minimal definition following [40].

Definition 3.1.1 (Monad) Let A, B and C be types. A monad M is defined as the triple (`M`, `unit`, `_>>=_`) where `M` is a monadic constructor; `unit : A → M A` represents the identity function and `_>>=_ : M A → (A → M B) → M B` is used for monadic composition.

The triple must satisfy the following laws.

1. (**left identity**) For every `x : A` and `f : A → M B`, `unit x >>= f` ≡ `f x`;
2. (**right identity**) For every `mx : M A`, `mx >>= unit` ≡ `mx`; and
3. (**associativity**) For every `mx : M A`, `f : A → M B` and `g : B → M C`,

   `(mx >>= f) >>= g` ≡ `mx >>= (λ my → f my >>= g)`

3.2 The Delay monad

In 2005, Venanzio Capretta introduced the `Delay` monad to represent recursive (thus potentially infinite) computations in a coinductive (and monadic) fashion [41]. As described in [42], the `Delay` type is used to represent computations whose result may be available with some *delay* or never be returned at all: the `Delay` type has two constructors; one, `now`, contains the result of the computation. The second, `later`, embodies one "step" of delay and, of course, an infinite (coinductive) sequence of `later` indicates a non-terminating computation, practically making non-termination an effect.

In Agda, the `Delay` type is defined as follows (using *sizes* and *levels*, see Subsection 2.3.2.1):

```
data Delay {ℓ} (A : Set ℓ) (i : Size) : Set ℓ where
  now   : A → Delay A i
  later : Thunk (Delay A) i → Delay A i
```

snippet 3.2.1 from Agda's stdlib

Paired with the following `bind` function (`return`, or `unit`, is `now`).

```
bind : ∀ {i} → Delay A i → (A → Delay B i) → Delay B i
bind (now a)   f = f a
bind (later d) f = later λ where .force → bind (d .force) f
```

snippet 3.2.2 from Agda's stdlib

In words, what `bind` does, is this: given a `Delay A i x`, it checks whether `x` contains an immediate result (i.e., `x ≡ now a`) and, if so, it applies the function `f`; if, otherwise, `x` is a step of delay, (i.e., `x ≡ later d`), `bind` delays the computation by wrapping the observation of `d` (represented as `d .force`) in the `later` constructor. This is the only possibile definition: for example, `bind' (later d) f = bind' (d .force) f` would not pass the termination and productivity checker; in fact, take the `never` term as shown in snippet 3.2.3: of course, `bind' never f` would never terminate.

```
never : ∀ {i} → Delay A i
never = later λ where .force → never
```

snippet 3.2.3 from Agda's stdlib

We might however argue that `bind` as well never terminates, in fact `never` *never yields a value* by definition; this is correct, but the two views on non-termination are radically different. The detail is that `bind'` observes the whole of `never` immediately, while `bind` leaves to the observer the job of actually inspecting what the result of `bind x f` *is*, and this is the utility of the `Delay` datatype and its monadic features.

3.3 Bisimilarity

Consider the following snippet.

```
comp-a : ∀ {i} → Delay ℤ i
comp-a = now 0ℤ
```

see code snippet 3.3.1

The term represents in snippet 3.3.1 a computation converging to the value `0` immediately, as no `later` appears in its definition.

```
comp-b : ∀ {i} → Delay ℤ i
comp-b = later λ where .force → now 0ℤ
```

see code snippet 3.3.2

The term above represent the same converging computation, albeit in a different number of steps. There are situations in which we want to consider equal computations that result in the same outcome, be it a concrete value (or failure) or a diverging computation. We cannot use Agda's propositional equality, as the two terms *are not the same*:

```
comp-a≡comp-b : comp-a ≡ comp-b
comp-a≡comp-b = refl
-- ^ now 0ℤ ≠ later (λ { .force → now 0ℤ }) of type Delay ℤ ∞
```

see code snippet 3.3.3

We thus define an equivalence relation on `Delay` known as **weak bisimilarity**. In words, weak bisimilarity relates two computations such that either both diverge or both converge to the same value, independent of the number of steps taken[3].

Definition 3.3.1 (Weak bisimilarity) Let a_1 and a_2 be two terms of type A. Then, weak bisimilarity of terms of type `Delay A` is defined by the following inference rules.

$$\frac{a_1 \equiv a_2}{\text{now } a_1 \approx \text{now } a_2}\ \text{now} \qquad \frac{\text{force } x_1 \approx \text{force } x_2}{\text{later } x_1 \approx \text{later } x_2}\ \text{later}$$

$$\frac{\text{force } x_1 \approx x_2}{\text{later } x_1 \approx x_2}\ \text{later}_l \qquad \frac{x_1 \approx \text{force } x_2}{x_1 \approx \text{later } x_2}\ \text{later}_r$$

The implementation in Agda of Definition 3.3.1 follows the rules above but uses sizes to deal with coinductive definitions (see Subsection 2.3.2.1) and retraces the definition of *strong* bisimilarity as implemented in Agda's standard library at the time of writing:

[3] **Strong** bisimilarity, on the other hand, requires both computation to converge to the same value in the same number of steps; it is easy to show that strong bisimilarity implies weak bisimilarity.

the difference with the rules shown in Definition 3.3.1 is that in the latter the inference rules imply that propositional equality is the only kind of relation allowed for two terms to be weakly bisimilar at the level of non-delayed terms, while this definition allows terms of two potentially different "end" types to be bisimilar as long as they are related by some relation R.

```
data WeakBisim {a b r} {A : Set a} {B : Set b} (R : A → B → Set r) i :
           (xs : Delay A ∞) (ys : Delay B ∞) → Set (a ⊔ b ⊔ r) where
  now    : ∀ {x y} → R x y → WeakBisim R i (now x) (now y)
  later  : ∀ {xs ys} → Thunk^R (WeakBisim R) i xs ys
           → WeakBisim R i (later xs) (later ys)
  laterₗ : ∀ {xs ys} → WeakBisim R i (force xs) ys
           → WeakBisim R i (later xs) ys
  laterᵣ : ∀ {xs ys} → WeakBisim R i xs (force ys)
           → WeakBisim R i xs (later ys)
```

see code snippet 3.3.4

Propositional equality is still the most frequently used relation, so we define a special notation for this specialization, which resembles that of the inference rules:

```
infix 1 _⊢_≈_
_⊢_≈_ : ∀ i → Delay A ∞ → Delay A ∞ → Set ℓ
_⊢_≈_ = WeakBisim _≡_
```

see code snippet 3.3.5

We also show that weak bisimilarity as we defined it is an equivalence relation. When expressing this theorem in Agda, it is also necessary to make the relation `R` we abstract over be an equivalence relation, as shown in Theorem 3.3.1; as shown in [12], the transitivity proof is not claimed to be size preserving.

Theorem 3.3.1 (Weak bisimilarity is an equivalence relation)

```
reflexive : ∀ {i} (r-refl : Reflexive R)  → Reflexive (WeakBisim R i)
symmetric : ∀ {i} (r-sym : Sym P Q) → Sym (WeakBisim P i) (WeakBisim Q i)
transitive : ∀ {i} (r-trans : Trans P Q R) → Trans (WeakBisim P ∞) (WeakBisim Q ∞)
(WeakBisim R i)
```

see code see proof a.1.1 snippet 3.3.6

Theorem 3.3.2 states that `Delay` is a monad up to weak bisimilarity.

Theorem 3.3.2 (`Delay` is a monad) The triple (`Delay`, `now`, `bind`) is a monad and respects monad laws up to weak bisimilarity. In Agda:

```
left-identity : ∀ {i} (x : A) (f : A → Delay B i) → (now x) >>= f ≡ f x
right-identity : ∀ {i} (x : Delay A ∞) → i ⊢ x >>= now ≈ x
associativity : ∀ {i} {x : Delay A ∞} {f : A → Delay B ∞}
  {g : B → Delay C ∞} → i ⊢ (x >>= f) >>= g ≈ x >>= λ y → (f y >>= g)
```

see code see proof a.1.2 snippet 3.3.7

3.4 Convergence, divergence and failure

Using the relation of weak bisimilarity, we want to define a characterization of computations, which we will use later when expressing theorems regarding the semantics of the language we will consider.

The `Delay` monads allows us to model the effect of non-termination, but, other than modeling converging computations, we also want to model the behaviour of computations that terminate but in a wrong way, which we name *failing*. We model this effect with the aid of the `Maybe` monad, creating a new monad that combines the two behaviours: we baptize this new monad `FailingDelay`.

This monad does not have a specific datatype (as it is the combination of two existing monads), so we directly show the definition of `bind` in Agda (snippet 3.4.1).

```
bind : ∀ {i} (d : Delay {ℓ} (Maybe A) i) (f : (A → Delay {ℓ′} (Maybe B) i))
       → Delay {ℓ′} (Maybe B) i
bind (now (just x)) f = f x
bind (now nothing) f = now nothing
bind (later x) f = later (λ where .force → bind (x .force) f)
```

see code snippet 3.4.1

Having a monad that deals with the three effects (if we consider convergence one) we want to model, we now define types for these three states. The first we consider is termination (or convergence); in words, we define a computation to converge when there exists a term `v` such that the computation is (weakly) bisimilar to it (see Definition 3.4.1).

Definition 3.4.1 (Converging computation)

```
_⇓_ : ∀ (x : Delay (Maybe A) ∞) (v : A) → Set ℓ
x ⇓ v = ∞ ⊢ x ≋ (now (just v))

_⇓ : ∀ (x : Delay (Maybe A) ∞) → Set ℓ
x ⇓ = ∃ λ v → ∞ ⊢ x ≋ (now (just v))
```

see code snippet 3.4.2

We then define a computation to diverge when it is bisimilar to an infinite chain of `later`, which we named `never` in snippet 3.2.3 (see Definition 3.4.2).

Definition 3.4.2 (Diverging computation)

```
_⇑ : ∀ (x : Delay (Maybe A) ∞) → Set ℓ
x ⇑ = ∞ ⊢ x ≋ never
```

see code snippet 3.4.3

The third and last possibility is for a computation to fail: such a computation converges but to no value (see Definition 3.4.3).

Definition 3.4.3 (Failing computation)

```
_↯ : ∀ (x : Delay (Maybe A) ∞) → Set ℓ
x ↯ = ∞ ⊢ x ≋ now nothing
```

see code snippet 3.4.4

We can already say that a computation, in the semantics we will define later, will not show any other kind of behaviour, therefore Postulate 3.4.1 seems clearly true; in a constructive environment like Agda we can, however, only postulate it, as a proof would essentially be a solution to the halting problem.

Postulate 3.4.1

```
three-states : ∀ {a} {A : Set a} {x : Delay (Maybe A) ∞}
             → XOr (x ⇓) (XOr (x ⇑) (x ↯))
```

see code snippet 3.4.5

CHAPTER 4

The Imp programming language

In this chapter we will go over the implementation of a simple – but Turing complete – imperative language called **Imp**, as described in [43]. After defining its syntax, we will give rules for its semantics and show its implementation in Agda. After this introductory work, we will discuss our implementation of transformations on Imp programs.

4.1 Introduction

4.1.1 Syntax

The syntax of the Imp language can be described in a handful of EBNF rules, as shown in Grammar 1.

aexp	$\rightarrow$	n	*integer constant*
	$\mid$	id	*identifiers*
	$\mid$	$a_1 + a_2$	*sum*
bexp	$\rightarrow$	b	*boolean constant*
	$\mid$	$a_1 < a_2$	*integer inequality*
	$\mid$	$\neg b$	*boolean negation*
	$\mid$	$b_1 \wedge b_2$	*logical and*
command	$\rightarrow$	skip	*nop*
	$\mid$	id $\leftarrow$ **aexp**	*assignment*
	$\mid$	$c_1 ; c_2$	*sequence*
	$\mid$	if **bexp** then c_1 else c_2	*conditional*
	$\mid$	while **bexp** do c	*loop*

Grammar 1: **Imp**

The syntactic elements of this language are *commands*, *arithmetic expressions*, *boolean expressions* and *identifiers*. Given its simple nature, it is easy to give an abstract representation for its concrete syntax: all of the elements can be represented with simple datatypes enclosing all the information embedded in the syntactic rules, as shown in snippet 4.1.1, snippet 4.1.2, snippet 4.1.3 and snippet 4.1.4.

```
Ident : Set
Ident = String
```

see code snippet 4.1.1

```
data AExp : Set where
  const : (n : ℤ)  → AExp
  var   : (id : Ident) → AExp
  plus  : (a₁ a₂ : AExp) → AExp
```

see code snippet 4.1.2

```
data BExp : Set where
  const : (b : Bool) → BExp
  le    : (a₁ a₂ : AExp) → BExp
  not   : (b : BExp) → BExp
  and   : (b₁ b₂ : BExp) → BExp
```

see code snippet 4.1.3

```
data Command : Set where
  skip   : Command
  assign : (id : Ident) → (a : AExp) → Command
  seq    : (c₁ c₂ : Command) → Command
  ifelse : (b : BExp) → (c₁ c₂ : Command) → Command
  while  : (b : BExp) → (c : Command) → Command
```

see code snippet 4.1.4

4.1.2 Stores

Identifiers in Imp have an important role. Identifiers can be initialized or uninitialized (see Chapter 4.2 for a more detailed reasoning about their role) and their value, if any, can change in time. We need a way to keep track of identifiers and their value: this tool is the `Store`. Stores are defined as shown in snippet 4.1.5, that is, as *partial maps* with the use of the `Maybe` monad.

```
Store : Set
Store = Ident → Maybe ℤ
```

see code snippet 4.1.5

We now proceed to show basic definitions over partial maps.

1. **in-store predicate**: let id be an identifier and σ be a store. To say that id is in σ we write id $\in \sigma$; in other terms, it is the same as $\exists\, v \in \mathbb{Z}, \sigma$ id $\equiv$ just v.

2. **empty store**: we denote the empty store as $\emptyset$. For this special store, it is always $\forall\, \mathrm{id}, \mathrm{id} \in \emptyset \rightarrow \bot$ or $\forall\, \mathrm{id}, \emptyset\, \mathrm{id} \equiv \mathrm{nothing}$.

```
empty : Store
empty = λ _ → nothing
```

see code snippet 4.1.6

3. **adding an identifier**: let id be an identifier and $v : \mathbb{Z}$ be a value. We denote the insertion of the pair (id, v) in a store σ as $(\mathrm{id}, v) \mapsto \sigma$.

```
update : (id₁ : Ident) → (v : ℤ) → (s : Store) → Store
update id₁ v s id₂ with id₁ == id₂
... | true = (just v)
... | false = (s id₂)
```

see code snippet 4.1.7

4. **joining two stores**: let σ_1 and σ_2 be two stores. We define the store that contains an id if $\mathrm{id} \in \sigma_1$ or $\mathrm{id} \in \sigma_2$ as $\sigma_1 \cup \sigma_2$. Notice that the join operation is not commutative, as it may be that

$$\exists\, \mathrm{id}, \exists\, v_1, \exists\, v_2, v_1 \neq v_2 \wedge \sigma_1\, \mathrm{id} \equiv \mathrm{just}\, v_1 \wedge \sigma_2\, \mathrm{id} \equiv \mathrm{just}\, v_2$$

```
join : (s₁ s₂ : Store) → Store
join s₁ s₂ id with (s₁ id)
... | just v = just v
... | nothing = s₂ id
```

see code snippet 4.1.8

5. **merging two stores**: let σ_1 and σ_2 be two stores. We define the store that contains an id if and only if $\sigma_1\, \mathrm{id} \equiv \mathrm{just}\, v$ and $\sigma_2\, \mathrm{id} \equiv \mathrm{just}\, v$ as $\sigma_1 \cap \sigma_2$.

```
merge : (s₁ s₂ : Store) → Store
merge s₁ s₂ =
  λ id → (s₁ id) >>=
    λ v₁ → (s₂ id) >>=
      λ v₂ → if (⌊ v₁ ≟ v₂ ⌋) then just v₁ else nothing
---               ^ decidable boolean equality for integers
```

see code snippet 4.1.9

Definition 4.1.1 Let σ_1 and σ_2 be two stores. We define a new relation between them as

$$\forall \text{ id}, (\exists z,\ \sigma_1 \text{ id} \equiv \text{just } z) \to (\exists z,\ \sigma_2 \text{ id} \equiv \text{just } z) \tag{1}$$

and we denote it with $\sigma_1 \dotdiv \sigma_2$. In Agda:

```
_∻_ : Store → Store → Set
x ∻ x₁ = ∀ {id : Ident} → (∃ λ z → x id ≡ just z)
              → (∃ λ z → x₁ id ≡ just z)
```

see code snippet 4.1.10

And we prove the transitivity of this new relation:

Theorem 4.1.1 (Transitivity of ∻)

```
∻-trans : ∀ {s₁ s₂ s₃ : Store} (h₁ : s₁ ∻ s₂) (h₂ : s₂ ∻ s₃) → s₁ ∻ s₃
```

see code see proof a.2.1 snippet 4.1.11

4.2 Semantics

Having understood the syntax of Imp, we can move to the *meaning* of Imp programs. We will explore the operational semantics of the language using the formalism of inference rules, then we will show the implementation of the semantics (as an intepreter) for these rules.

Before describing the rules of the semantics, we will give a brief explaination of what we expect to be the result of the evaluation of an Imp program.

```
if true then skip else skip
```

snippet 4.2.1

An example of Imp program is shown in snippet 4.2.1. In general, we can expect the evaluation of an Imp program to terminate in some kind value or diverge. But what happens when, as mentioned in Subsection 4.1.1, an unitialized identifier is used, as shown for example in snippet 4.2.2? The execution of the program cannot possibly continue, and we define such a state as *failing* or *stuck* (see also Section 3.4).

Of course, there is a plethora of other kinds of failures we could model, both deriving from static analysis or from the dynamic execution of the program (for example, in a language with divisions, a division by 0), but we chose to model this kind of behaviour only.

```
while true do x <- y
```

snippet 4.2.2

We can now introduce the formal notation we will use to describe the semantics of Imp programs. We already introduced the concept of store, which keeps track of the mutation of identifiers and their value during the execution of the program. We write c, $\sigma \Downarrow \sigma_1$ to mean that the program c, when evaluated starting from the context σ, converges to the store σ_1; we write c, σ ↯ to say that the program c, when evaluated in context σ, does not converge to a result but, instead, execution gets stuck (that is, an unknown identifier is used).

The last possibility is for the execution to diverge, c, $\sigma \Uparrow$: this means that the evaluation of the program never stops and while no state of failure is reached no result is ever produced. An example of this behaviour is seen when evaluating snippet 4.2.3.

```
while true do skip
```

snippet 4.2.3

We are now able to give inference rules for each construct of the Imp language: we will start from simple ones, that is arithmetic and boolean expressions, and we will then move to commands. The inference rules we give follow the formalism of **big-step** operational semantics, that is, intermediate states of evaluation are not shown explicitly in the rules themselves.

4.2.1 Arithmetic expressions

Arithmetic expressions in Imp can be of three kinds: integer ($\mathbb{Z}$) constants, identifiers and sums. As anticipated, the evaluation of arithmetic expressions can fail, that is, the evaluation of arithmetic expressions is not a total function; again, the possibile erroneous states we can get into when evaluating an arithmetic expression mainly concerns the use of undeclared identifiers.

Without introducing them, we will use notations similar to that used earlier for commands, in particular $\cdot \Downarrow \cdot$.

$$\frac{}{\text{const n}, \sigma \Downarrow n} \qquad \frac{\text{id} \in \sigma}{\text{var id}, \sigma \Downarrow \sigma\,\text{id}} \qquad \frac{a_1, \sigma \Downarrow n_1 \quad a_2, \sigma \Downarrow n_2}{\text{plus}\ a_1 a_2, \sigma \Downarrow (n_1 + n_2)}$$

Table 9: Inference rules for the semantics of arithmetic expressions of Imp

The Agda code implementing the interpreter for arithmetic expressions is shown in snippet 4.2.4. As anticipated, the inference rules denote a partial function; however, since the predicate id ∈ σ is decidable, we can make the interpreter target the `Maybe` monad and make the intepreter a total function.

```
aeval : ∀ (a : AExp) (s : Store) → Maybe ℤ
aeval (const x) s = just x
aeval (var x) s = s x
aeval (plus a a₁) s = aeval a s >>= λ v₁ → aeval a₁ s >>= λ v₂ → just (v₁ + v₂)
```

see code snippet 4.2.4

4.2.2 Boolean expressions

Boolean expressions in Imp can be of four kinds: boolean constants, negation of a boolean expression, logical conjunction and, finally, comparison of arithmetic expressions.

$$\frac{}{\text{const c}, \sigma \Downarrow c} \qquad \frac{\text{b}, \sigma \Downarrow c}{\neg b, \sigma \Downarrow \neg c}$$

$$\frac{a_1, \sigma \Downarrow n_1 \quad a_2, \sigma \Downarrow n_2}{\text{le } a_1 a_2, \sigma \Downarrow (n_1 < n_2)} \qquad \frac{b_1, \sigma \Downarrow c_1 \quad b_2, \sigma \Downarrow c_2}{\text{and } b_1 b_2, \sigma \Downarrow (c_1 \wedge c_2)}$$

Table 10: Inference rules for the semantics of boolean expressions of Imp

The line of reasoning for the concrete implementation in Agda is the same as that for arithmetic expressions: the inference rules denote a partial function; since what makes this function partial – the definition of identifiers – is a decidable property, we can make the interpreter for boolean expressions a total function using the `Maybe` monad, as shown in snippet 4.2.5.

```
beval : ∀ (b : BExp) (s : Store) → Maybe Bool
beval (const c) s = just c
beval (le a₁ a₂) s = aeval a₁ s >>=
  λ v₁ → aeval a₂ s >>=
    λ v₂ → just (v₁ ≤ᵇ v₂)
beval (not b) s = beval b s >>= λ b → just (bnot b)
beval (and b₁ b₂) s = beval b₁ s >>=
  λ b₁ → beval b₂ s >>=
    λ b₂ → just (b₁ ∧ b₂)
```

see code snippet 4.2.5

4.2.3 Commands

$$\frac{}{\text{skip}, \sigma \Downarrow \sigma}\ \Downarrow\text{skip} \qquad \frac{a, \sigma \Downarrow v}{\text{assign id } a, \sigma \Downarrow \text{update id } v\ \sigma}\ \Downarrow\text{assign}$$

$$\frac{c_1, \sigma \Downarrow \sigma_1 \quad c_2, \sigma_1 \Downarrow \sigma_2}{\text{seq } c_1\ c_2, \sigma \Downarrow \sigma_2}\ \Downarrow\text{seq} \qquad \frac{c^t, \sigma \Downarrow \sigma^t \quad b, \sigma \Downarrow \text{true}}{\text{if } b \text{ then } c^t \text{ else } c^f, \sigma \Downarrow \sigma^t}\ \Downarrow\text{if-true}$$

$$\frac{c^f, \sigma \Downarrow \sigma^f \quad b, \sigma \Downarrow \text{false}}{\text{if } b \text{ then } c^t \text{ else } c^f, \sigma \Downarrow \sigma^f}\ \Downarrow\text{if-false} \qquad \frac{b, \sigma \Downarrow \text{false}}{\text{while } b \text{ do } c, \sigma \Downarrow \sigma}\ \Downarrow\text{while-false}$$

$$\frac{b, \sigma \Downarrow \text{true} \quad c, \sigma \Downarrow \sigma'}{\text{while } b \text{ do } c, \sigma \Downarrow \sigma'}\ \Downarrow\text{while-true}$$

Table 11: Inference rules for the semantics of commands

We need to be careful when examining the inference rules in Table 11. Although they are graphically rendered the same, the convergency propositions used in the inference rules are different from those in Chapter 4.2.2 or Chapter 4.2.1. In fact, while in the latter the only modeled effect is a decidable one, the convergency proposition here models two effects, partiality and failure. While failure, intended as we did before, is a decidable property, partiality is not, and we cannot design an interpreter for these rules targeting the `Maybe` monad only: we must thus combine the effects and target the `FailingDelay` monad, as shown in Section 3.4. The code for the intepreter is shown in snippet 4.2.6.

```
mutual
 ceval-while : ∀ {i} (c : Command) (b : BExp) (s : Store)
               → Thunk (Delay (Maybe Store)) i
 ceval-while c b s = λ where .force → (ceval (while b c) s)

 ceval : ∀ {i} → (c : Command) → (s : Store) → Delay (Maybe Store) i
 ceval skip s = now (just s)
 ceval (assign id a) s =
    now (aeval a s) ≫= λ v → now (just (update id v s))
 ceval (seq c c₁) s =
    ceval c s ≫= λ s' → ceval c₁ s'
 ceval (ifelse b c c₁) s =
    now (beval b s) ≫= (λ bᵥ → (if bᵥ then ceval c s else ceval c₁ s))
 ceval (while b c) s =
    now (beval b s) ≫=
      (λ bᵥ → if bᵥ
        then (ceval c s ≫=  λ s → later (ceval-while c b s))
        else now (just s))
```

see code snippet 4.2.6

The last rule (`while` for `beval b` converging to `just true`) is coinductive, and this is reflected in the code by having the computation happen inside a `Thunk` (see Section 2.3.2.1)

4.2.4 Properties of the interpreter

Regarding the intepreter, the most important property we want to show puts in relation the starting store a command is evaluated in and the (hypothetical) resulting store. Up until now, we kept the mathematical layer and the code layer separated; from now on we will collapse the two and allow ourselves to use mathematical notation to express formal statements about the code: in practice, this means that, for example, the mathematical names aeval, beval and ceval refer to names from the "code layer" `aeval`, `beval` and `ceval`, respectively.

Lemma 4.2.1 Let c be a command and σ_1 and σ_2 be two stores. Then

$$\text{ceval}\ c, \sigma_1 \Downarrow \sigma_2 \rightarrow \sigma_1 \ \dot{\div}\ \sigma_2$$

```
ceval⇓⇒÷ : ∀ (c : Command) (s s' : Store) (h⇓ : (ceval c s) ⇓ s') → s ÷ s'
```

see code see proof a.2.2 snippet 4.2.7

Lemma 4.2.1 will be fundamental for later proofs.

It is also important, now that all is set up, to underline that the meaning of c, $\sigma \Downarrow \sigma_1$, c, σ ↯ and c, $\sigma \Uparrow$ which we used giving an intuitive description but without a concrete definition, are exactly the types described in Section 3.4, with the parametric types adapted to the situation at hand: thus, saying c, $\sigma \Downarrow \sigma_1$ actually means that ceval $c\ \sigma \approx \text{now}\ (\text{just}\ \sigma_1)$, c, $\sigma \Uparrow$ means that ceval $c\ \sigma \approx$ never and c, σ ↯ means that ceval $c\ \sigma \approx$ now nothing.

4.3 Analyses and optimizations

We chose to demonstrate the use of coinduction in the definition of operational semantics implementing transformations on the code itself, then showing proofs regarding the result of the execution of the program. The main inspiration for these transformations is [44], and they are *source to source*, that is, they transform Imp programs to (pontentially untouched) Imp programs.

4.3.1 Definite initialization analysis

The first transformation we describe is **definite initialization analysis**. In general, the objective of this analysis is to ensure that no variable is ever used before being initialized, which is exactly the only kind of failure we chose to model.

Variables and indicator functions

This analysis deals with variables. Before delving into its details, we show first a function to compute the set of variables used in arithmetic and boolean expressions. The objective is to come up with a *set* of identifiers that appear in the expression: we chose to represent sets in Agda using characteristic functions, which we simply define as parametric functions from a parametric set to the set of booleans, that is `CharacteristicFunction = A → Bool`; later, we will instantiate this type for identifiers, giving the resulting type the name of `VarsSet`. First, we give a (parametric) notion of members equivalence (that is, a function `_==_ : A → A → Bool`); then, we the usual operations on sets (insertion, union, and intersection) and the usual definition of inclusion for characteristic functions.

```
module Data.CharacteristicFunction {a} (A : Set a) (_==_ : A → A → Bool) where
-- ...
CharacteristicFunction : Set a
CharacteristicFunction = A → Bool
-- ...

∅ : CharacteristicFunction
∅ = λ _ → false

_↦_ : (v : A) → (s : CharacteristicFunction) → CharacteristicFunction
(v ↦ s) x = (v == x) ∨ (s x)

_∪_ : (s₁ s₂ : CharacteristicFunction) → CharacteristicFunction
(s₁ ∪ s₂) x = (s₁ x) ∨ (s₂ x)

_∩_ : (s₁ s₂ : CharacteristicFunction) → CharacteristicFunction
(s₁ ∩ s₂) x = (s₁ x) ∧ (s₂ x)

_⊆_ : (s₁ s₂ : CharacteristicFunction) → Set a
s₁ ⊆ s₂ = ∀ x → (x-in-s₁ : s₁ x ≡ true) → s₂ x ≡ true
```

see code snippet 4.3.1

Theorem 4.3.1 (Equivalence of characteristic functions)
(using the **Axiom of extensionality**)

```
cf-ext : ∀ {s₁ s₂ : CharacteristicFunction}
  (a-ex : ∀ x → s₁ x ≡ s₂ x) → s₁ ≡ s₂
```

see code see proof a.2.3 snippet 4.3.2

Theorem 4.3.2 (Neutral element of union)

```
∪-∅ : ∀ {s : CharacteristicFunction} → (s ∪ ∅) ≡ s
```

see code snippet 4.3.3

Theorem 4.3.3 (Update inclusion)

```
↦⇒⊆ : ∀ {id} {s : CharacteristicFunction} → s ⊆ (id ↦ s)
```

see code snippet 4.3.4

Theorem 4.3.4 (Transitivity of inclusion)

```
⊆-trans : ∀ {s₁ s₂ s₃ : CharacteristicFunction} → (s₁⊆s₂ : s₁ ⊆ s₂)
          → (s₂⊆s₃ : s₂ ⊆ s₃) → s₁ ⊆ s₃
```

see code snippet 4.3.5

We will also need a way to get a `VarsSet` from a `Store`, which is shown in snippet 4.3.6.

```
dom : Store → VarsSet
dom s x with (s x)
... | just _ = true
... | nothing = false
```

see code snippet 4.3.6

Realization

Following [44], the first formal tool we need is a way to compute the set of variables mentioned in expressions, shown in snippet 4.3.7 and snippet 4.3.8. We also need a function to compute the set of variables that are definitely initialized in commands, which is shown in snippet 4.3.9.

```
avars : (a : AExp) → VarsSet
avars (const n) = ∅
avars (var id) = id ↦ ∅
avars (plus a₁ a₂) =
  (avars a₁) ∪ (avars a₂)
```

see code snippet 4.3.7

```
bvars : (b : BExp) → VarsSet
bvars (const b) = ∅
bvars (le a₁ a₂) =
      (avars a₁) ∪ (avars a₂)
bvars (not b) = bvars b
bvars (and b b₁) =
      (bvars b) ∪ (bvars b₁)
```

see code snippet 4.3.8

```
cvars : (c : Command) → VarsSet
cvars skip = ∅
cvars (assign id a) = id ↦ ∅
cvars (seq c c₁) = (cvars c) ∪ (cvars c₁)
cvars (ifelse b cᵗ cᶠ) = (cvars cᵗ) ∩ (cvars cᶠ)
cvars (while b c) =  ∅
```

see code snippet 4.3.9

It is worth to reflect upon the definition of snippet 4.3.9. This code computes the set of *initialized* variables in a command `c`; as done in [44], we construct this set of initialized variables in the most conservative way possible: of course, `skip` does not have any initialized variable and `assign id a` adds `id` to the set of initialized variables.

However, when considering composite commands, we must consider that, except for `seq c c₁`, not every branch of execution is taken; this means that we cannot know statically whether `ifelse b cᵗ cᶠ` will lead to the execution to the execution of `cᵗ` or `cᶠ`, we thus take the intersection of their initialized variables, that is we compute the set of variables that will be surely initialized wheter one or the other executes. The same reasoning applies to `while b c`: we cannot possibly know whether or not `c` will ever execute, thus we consider no new variables initialized.

At this point it should be clear that as `cvars c` computes the set of initialized variables in a conservative fashion, it is not necessarily true that the actual execution of the command will not add additional variables: however, knowing that if the evaluation of a command in a store σ converges to a value σ', that is $c, \sigma \Downarrow \sigma'$ then by Lemma 4.2.1 $\text{dom } \sigma \subseteq \text{dom } \sigma'$; this allows us to show the following lemma.

Lemma 4.3.1 Let c be a command and σ and σ' be two stores. Then

$$\text{ceval } c\, \sigma \Downarrow \sigma' \to (\text{dom } \sigma_1 \cup (\text{cvars } c)) \subseteq \left(\text{dom } \sigma'\right)$$

```
ceval⇓⇒sc⊆s' :  ∀ (c : Command) (s s' : Store) (h⇓ : (ceval c s) ⇓ s')
                   → (dom s ∪ (cvars c)) ⊆ (dom s')
```

see code see proof a.2.4 snippet 4.3.10

We now give inference rules that inductively build the relation that embodies the logic of the definite initialization analysis, shown in Table 12. In Agda, we define a datatype representing the relation of type `Dia : VarsSet → Command → VarsSet → Set`, which is shown in snippet 4.3.11. Lemma 4.3.1 will allow us to show that there is a relation between the VarsSet in the Dia relation and the actual stores that are used in the execution of a command.

$$\frac{}{\text{Dia v skip v}} \qquad \frac{\text{avars } a \subseteq v}{\text{Dia } v \text{ (assign id } a\text{) (id} \mapsto v)}$$

$$\frac{\text{Dia } v_1\ c_1\ v_2 \quad \text{Dia } v_2\ c_2\ v_3}{\text{Dia } v_1 \text{ (seq } c_1\ c_2)\ v_3} \qquad \frac{\text{bvars } b \subseteq v \quad \text{Dia } v\ c^t\ v^t \quad \text{Dia } v\ c^f\ v^f}{\text{Dia } v \text{ (if } b \text{ then } c^t \text{ else } c^f\text{) } (v^t \cap v^f)}$$

$$\frac{\text{bvars } b \subseteq v \quad \text{Dia } v\ c\ v_1}{\text{Dia } v \text{ (while } b\ c)\ v}$$

Table 12: Inference rules for the definite initialization analysis

```
data Dia : VarsSet → Command → VarsSet → Set where
  skip : ∀ (v : VarsSet) → Dia v (skip) v
  assign : ∀ a v id (a⊆v : (avars a) ⊆ v) → Dia v (assign id a) (id ↦ v)
  seq : ∀ v₁ v₂ v₃ c₁ c₂ → (relc₁ : Dia v₁ c₁ v₂) →
         (relc₂ : Dia v₂ c₂ v₃) → Dia v₁ (seq c₁ c₂) v₃
  if : ∀ b v vᵗ vᶠ cᵗ cᶠ (b⊆v : (bvars b) ⊆ v) → (relcᶠ : Dia v cᶠ vᶠ) →
         (relcᵗ : Dia v cᵗ vᵗ) → Dia v (ifelse b cᵗ cᶠ) (vᵗ ∩ vᶠ)
  while : ∀ b v v₁ c → (b⊆s : (bvars b) ⊆ v) →
         (relc : Dia v c v₁) → Dia v (while b c) v
```

see code snippet 4.3.11

What we want to show now is that if Dia holds, then the evaluation of a command c does not result in an error: while Theorem 4.3.5 and Theorem 4.3.6 show that if the variables in an arithmetic expression or a boolean expression are contained in a store the result of their evaluation cannot be a failure (i.e. they result in "just" something, as it cannot diverge), Theorem 4.3.7 shows that if Dia holds, then the evaluation of a program failing is absurd: therefore, by Postulate 3.4.1, the program either diverges or converges to some value.

Theorem 4.3.5 (Safety of arithmetic expressions)

```
adia-safe : ∀ (a : AExp) (s : Store) (dia : avars a ⊆ dom s)
                → (∃ λ v → aeval a s ≡ just v)
```

see code see proof a.2.5 snippet 4.3.12

Theorem 4.3.6 (Safety of boolean expressions)

```
bdia-safe : ∀ (b : BExp) (s : Store) (dia : bvars b ⊆ dom s)
                   → (∃ λ v → beval b s ≡ just v)
```

see code   see proof a.2.6   snippet 4.3.13

Theorem 4.3.7 (Safety of definite initialization for commands)

```
dia-safe : ∀ (c : Command) (s : Store) (v v' : VarsSet) (dia : Dia v c v')
  (v⊆s : v ⊆ dom s) → (h-err : (ceval c s) ↯) → ⊥
```

see code   see proof a.2.7   snippet 4.3.14

We now show an idea of the proof (the full proof, in Agda, is in Proof A.2.7), examining the two base cases `c ≡ skip` and `c ≡ assign id a` and the coinductive case `c ≡ while b c'`. The proof for the base cases is, in words, based on the idea that the evaluation cannot possibly go wrong: note that by the hypotheses, we have that `(ceval c s) ↯`, which we can express in math as ceval $c\ \sigma \approx$ now nothing.

Proof 4.3.1

1. Let c be the command `skip`. Then, for any store σ, by the definition of `ceval` in snippet 4.2.6 and by the inference rule ⇓skip in Table 11, the evaluation of c in the store σ must be

$$\text{ceval skip } \sigma \equiv \text{now } (\text{just } \sigma)$$

   Given the hypothesis that c, σ ↯, we now have that it must be now nothing $\approx$ now $(\text{just } \sigma)$, which is false for any σ, making the hypothesis c, σ ↯ impossible.

2. Let c be the command `assign id a`, for some identifier id and arithmetic expression a. By the hypothesis, we have that it must be Dia v $(\text{assign id } a)$ v' for some v and v', which entails that the variables that appear in a, which we named avars a, are all initialized in v, that is avars $a \subseteq v$; this and the hypothesis that $v \subseteq \text{dom } \sigma$ imply by Theorem 4.3.4 that avars $a \subseteq \text{dom } \sigma$.

   By Theorem 4.3.5, with the assumption that avars $a \subseteq \text{dom } \sigma$, it must be aeval $a\sigma \equiv$ just n for some $n : \mathbb{Z}$. Again, by the definition of `ceval` in snippet 4.2.6 and by the inference rule ⇓assign in Table 11, the evaluation of c in the store σ must be

$$\text{ceval } (\text{assign id } a)\ \sigma \equiv \text{now } (\text{just } (\text{update id } n\ \sigma))$$

and, as before, by the hypothesis that c fails it must thus be that now nothing ≈ now (just (update id n σ)), which is impossible for any σ, making the hypotesis c ↯ impossible.

3. Let c be the command `while b c'` for some boolean expression b and some command c'. By Theorem 4.3.6, with the assumption that bvars $b \subseteq$ dom σ, it must be beval b $\sigma \equiv$ just v for some $v : \mathbb{B}$.

   If $v \equiv$ false, then by the definition of `ceval` in snippet 4.2.6 and by the inference rule ⇓while-false in Table 11, the evaluation of c in the store σ must be

$$\text{ceval}\ (\text{while}\ b\ c')\ \sigma \equiv \text{now}\ (\text{just}\ \sigma)$$

   making the hypothesis that the evaluation of c fails impossible.

   If, instead, $v \equiv$ true, we must evaluate c' in σ. The case $c' \equiv$ now nothing is impossible by the inductive hypothesis.

   If $c' \equiv$ now (just σ') for some σ', then, by recursion, it must be

```
dia-sound (while b c) s' v v dia (⊆-trans v⊆s (ceval⇓⇒⊆ c s s' (≡⇒≈ eq-ceval-c)) w↯
```

   Finally, if $c' \equiv$ later x for some x, then we can prove inductively that

```
dia-sound-while-later : ∀ {x : Thunk (Delay (Maybe Store)) ∞} {b c} {v}
  (l↯⊥ : (later x)↯ → ⊥) (dia : Dia v (while b c) v)
  (l⇓s⇒⊆ : ∀ {s : Store} → ((later x) ⇓ s) → v ⊆ dom s)
  (w↯ : (bind (later x) (λ s → later (ceval-while c b s))) ↯) → ⊥
```

see code see proof a.2.8 snippet 4.3.15

The proof works by unwinding, inductively, the assumption that c ↯: if it fails, then ceval $c\,\sigma$ must eventually converge to now nothing. The proof thus works by showing base cases and, in the case of seq $c_1\,c_2$ and while $b\,c' \equiv$ if b then (seq c' (while $b\,c'$)) else skip, showing that by inductive hypotesis c_1 or c' cannot possibly fail; then, the assumption becomes that it is the second command (c_2 or while $b\,c'$) that fails, which we can inductively show absurd.

4.3.2 Pure constant folding optimization

Pure constant folding is the second and last transformation we consider. Again from [44], pure folding consists in statically examining the source code of the program in order to move, when possible, computations from runtime to (pre-)compilation.

The objective of pure constant folding is that of finding all the places in the source code where the result of expressions is computable statically: examples of this situation are `and true true`, `plus 1 1`, `le 0 1` and so on. This optimization is called *pure* because we avoid the phase of constant propagation, that is, we do not replace the value of identifiers even when their value is known at compile time.

Pure folding of arithmetic expressions

Pure folding optimization on arithmetic expressions is straighforward, and we define it as a function `apfold`. In words : let a be an arithmetic expression. Then, if a is a constant or an identifier the result of the optimization is a. If a is the sum of two other arithmetic expressions a_1 and a_2 ($a \equiv \text{plus}\ \ a_1\ a_2$), the optimization is performed on the two immediate terms a_1 and a_2, resulting in two potentially different expressions a_1' and a_2'. If both are constants v_1 and v_2 the result of the optimization is the constant $v_1 + v_2$; otherwise, the result of the optimization consists in the same arithmetic expression plus $a_1'\ a_2'$, that is, optimized immediate subterms. The Agda code for the function `apfold` is shown in snippet 4.3.16.

```
apfold : (a : AExp) → AExp
apfold (const x) = const x
apfold (var id) = var id
apfold (plus a₁ a₂) with (apfold a₁) | (apfold a₂)
... | const v₁ | const v₂ = const (v₁ + v₂)
... | a₁' | a₂' = plus a₁' a₂'
```

see code snippet 4.3.16

Of course, what we want to show is that this optimization does not change the result of the evaluation, as shown in Theorem 4.3.8.

Theorem 4.3.8 (Safety of pure folding for arithmetic expressions) Let a be an arithmetic expression and s be a store. Then

$$\text{aeval}\ a\ s \equiv \text{aeval}\ (\text{apfold}\ a)\ s$$

In Agda:

```
apfold-safe : ∀ a s → (aeval a s ≡ aeval (apfold a) s)
```

see code see proof a.2.9 snippet 4.3.17

Pure folding of boolean expressions

Pure folding of boolean expressions, which we define as a function `bpfold`, follows the same line of reasoning shown in Paragraph 4.3.2.1. Let b be a boolean expression. If b is an expression with no immediates (i.e. $b \equiv \text{const } n$) we leave it untouched. If, instead, b has immediate subterms, we compute the pure folding of them and build a result accordingly, as shown in snippet 4.3.18.

```
bpfold : (b : BExp) → BExp
bpfold (const b) = const b
bpfold (le a₁ a₂) with (apfold a₁) | (apfold a₂)
... | const n₁ | const n₂ = const (n₁ ≤ᵇ n₂ )
... | a₁ | a₂ = le a₁ a₂
bpfold (not b) with (bpfold b)
... | const n = const (lnot n)
... | b = not b
bpfold (and b₁ b₂) with (bpfold b₁) | (bpfold b₂)
... | const n₁ | const n₂ = const (n₁ ∧ n₂)
... | b₁' | b₂' = and b₁' b₂'
```

see code snippet 4.3.18

As before, our objective is to show that evaluating a boolean expression after the optimization yields the same result as the evaluation without optimization, as shown in Theorem 4.3.9.

Theorem 4.3.9 (Safety of pure folding for boolean expressions) Let b be a boolean expression and s be a store. Then

$$\text{beval } b\ s \equiv \text{beval } (\text{bpfold } b)\ s$$

```
bpfold-safe : ∀ b s → (beval b s ≡ beval (bpfold b) s)
```

see code see proof a.2.10 snippet 4.3.19

Pure folding of commands

Pure folding of commands builds on the definition of apfold and bpfold above combining the definitions as shown in snippet 4.3.20.

```
cpfold : Command → Command
cpfold skip = skip
cpfold (assign id a) with (apfold a)
... | const n = assign id (const n)
... | _ = assign id a
cpfold (seq c₁ c₂) = seq (cpfold c₁) (cpfold c₂)
cpfold (ifelse b c₁ c₂) with (bpfold b)
... | const false = cpfold c₂
... | const true = cpfold c₁
... | _ = ifelse b (cpfold c₁) (cpfold c₂)
cpfold (while b c) with (bpfold b)
... | const false = skip
... | b = while b c
```

see code snippet 4.3.20

And, again, what we want to show is that the pure folding optimization does not change the semantics of the program, that is, optimized and unoptimized programs converge to the same value or both diverge, as shown in Theorem 4.3.10.

Theorem 4.3.10 (Safety of pure folding for commands) Let c be a command and s be a store. Then

$$\text{ceval } c\ s \equiv \text{ceval } \left(\text{cpfold } b\right)\ s$$

```
cpfold-safe : ∀ (c : Command) (s : Store)
                → ∞ ⊢ (ceval c s) ≈ (ceval (cpfold c) s)
```

see code see proof a.2.11 snippet 4.3.21

Of course, what makes Theorem 4.3.10 different from the other safety proofs in this chapter is that we cannot use propositional equality and we must instead use weak bisimilarity. The execution of a program, in terms of chains of constructors `later` and `now`, changes for the same term if the pure folding optimization does indeed change the source. Take, for example, the case for `c ≡ while (plus 1 1) < 0 do skip`; this program will be optimized to `skip`, which results in a shorter evaluation.

4.4 Related works

The important aspect of this thesis is about the use of coinduction and sized types to express properties about the semantics of a language. Of course, this is not a new theoretical breakthrough, as it draws on a plethora of previous works, such as [12] and [4].

The general objective is that of coming up with a representation of the semantics of a language, be it functional or imperative, that allows a uniform representation of both the diverging and the fallible behaviour of the execution. Even if, surely, the idea comes up earlier in the literature, we choose to cite [4], where the author uses coinduction to model the diverging behaviour of the semantics of untyped λ-calculus but does so using a relational definition and not an equational one, making proofs concerning the semantics significantly more involved.

With the innovations proposed by Capretta's `Delay` monad, a new attempt to obtain such a representation was that of Danielsson in [12]; nonetheless, Agda's instrumentation for coinduction was not mature enough: it used the so-called *musical notation*, which suffered from the same limitations that regular induction has when using a syntax-based termination or productivity checker, and it is also worth noting that musical notation is potentially unsound when used together with sized types [24]. It would be unfair, however, not to mention that recent updates to the code related to [12] indeed uses sized types and goes beyond using concepts from cubical type theory.

In [44], the authors explore methods to apply transformations to programs of an imperative language and prove the equivalence of the semantics before and after such transformations; they do so using relational semantics without the use of coinduction, thus not considering the "effect" of non-termination. As noted, [44] is the work we followed to come up with transformations to explore.

In [45], the authors show four semantics: big-step and small-step relational semantics and big-step and small-step functional semantics. They achieve so using Coq, which had no concept such as sizes.

In [46], the authors show how to implement correct-by-construction compilers targeting the Delay monad.

CHAPTER 5

Conclusions

We defined an operational semantics for Imp targeting the `Delay` monad which, together with the `Maybe` monad, provided an adequate type to model the effects of divergence and failure: we did this with the aid of Agda, and we explored the implementation of such a semantics using sized types.

Our objective, other than the definition itself, was to use the semantics defined this way to show how it can be used when transforming a source program. The transformations we chose to implement, both suggested by Nipkow in [44], explore two source to source transformations.

The first, *definite initialization analysis* is, as the name suggests, a static check and in fact leaves the source code untouched; it provides, however, useful insights on the behaviour of the program when executed: if no `dia` relation can be built for the program at hand, it means that the program will surely crash and fail. On the other hand, if there exists a construction for `dia` for the program at hand, we are assured that the program will not fail – of course, it can still diverge.

The second, *pure folding optimization*, is a transformation that has the objective to lift information that is statically known to avoid run-time computations. We proved that this transformation, which indeed changes the syntactic structure of the program, does not change its semantics.

All throughout the work, *sizes* proved to be useful in the definitions and to keep track of termination and productivity: if compared with early versions of [12], one important difference is that, for example, we did not have to "trick" the termination checker, but every definition was fairly streamlined. We can also compare our realization with the work of Leroy in [4] and [47], which uses a relational definition of the semantics of Imp, which can make proofs more involved.

5.1 Future works

We chose to model only one kind of failure: an extension using `Result := Ok v | Error e` and a monad based on that type is fairly straightforward. The list of possible optimizations is long and well described in the literature: an interesting work can be the implementation of a general-purpose back-end and investigate various optimiziations used in the industry, starting from the translation of a low-level intermediate representation into static single-assignment form or continuation-passing style, proving easy properties of the transformations.

APPENDIX A

Proofs

A.1 The delay monad

Proof A.1.1 (for Theorem 3.3.1)

```
reflexive : Reflexive R → ∀ {i} → Reflexive (WeakBisim R i)
reflexive refl^R {i} {now x} = now refl^R
reflexive refl^R {i} {later x} = later λ where .force → reflexive (refl^R)

symmetric : Sym P Q → ∀ {i} → Sym (WeakBisim P i) (WeakBisim Q i)
symmetric sym^PQ (now x) = now (sym^PQ x)
symmetric sym^PQ (later x) = later λ where .force → symmetric (sym^PQ) (force x)
symmetric sym^PQ {i} (laterₗ x ) = laterᵣ (symmetric sym^PQ x)
symmetric sym^PQ (laterᵣ x) = laterₗ (symmetric sym^PQ x)

--- ...
transitive-now : ∀ {i} {x y z} (t : Trans P Q R) (p : WeakBisim P ∞ (now x) y)
 (q : WeakBisim Q ∞ y z) → WeakBisim R i (now x) z
transitive-now t (now p) (now q) = now (t p q)
transitive-now t (now p) (laterᵣ q) = laterᵣ (transitive-now t (now p) q)
transitive-now t (laterᵣ p) (later x) = laterᵣ (transitive-now t p (force x))
transitive-now t (laterᵣ p) (laterᵣ q) = laterᵣ (transitive-now t p (laterˡ⁻¹ q))
transitive-now t (laterᵣ p) (laterₗ q) = transitive-now t p q

mutual
 transitive-later : ∀ {i} {x y z} (t : Trans P Q R) (p : WeakBisim P ∞ (later x) y)
  (q : WeakBisim Q ∞ y z) → WeakBisim R i (later x) z
 transitive-later t p (later q)  = later λ { .force → transitive t (later⁻¹ p) (force q) }
 transitive-later t p (laterᵣ q) = later λ { .force → transitive t (laterˡ⁻¹ p) q }
 transitive-later t p (laterₗ q) = transitive-later t (laterʳ⁻¹ p) q
 transitive-later t (laterₗ p) (now q) = laterₗ (transitive t p (now q))

 transitive : ∀ {i} (t : Trans P Q R) → Trans (WeakBisim P ∞) (WeakBisim Q ∞) (WeakBisim R i)
 transitive t {now x} p q = transitive-now t p q
 transitive t {later x} p q = transitive-later t p q
```

see code snippet a.1.1

Proof A.1.2 (for Theorem 3.3.2)

```
left-identity : ∀ {i} (x : A) (f : A → Delay B i) → (now x) >>= f ≡ f x
left-identity {i} x f = _≡_.refl

right-identity : ∀ {i} (x : Delay A ∞) → i ⊢ x >>= now ≈ x
right-identity (now x) = now _≡_.refl
right-identity {i} (later x) = later (λ where .force → right-identity (force x))

associativity : ∀ {i} {x : Delay A ∞} {f : A → Delay B ∞} {g : B → Delay C ∞}
 → i ⊢ (x >>= f) >>= g ≈ x >>= λ y → (f y >>= g)
associativity {i} {now x} {f} {g} with (f x)
... | now x₁ = Codata.Sized.Delay.Bisimilarity.refl
... | later x₁ = Codata.Sized.Delay.Bisimilarity.refl
associativity {i} {later x} {f} {g} = later (λ where .force → associativity {x =
force x})
```

see code snippet a.1.2

A.2 The Imp programming language

Proof A.2.1 (for Theorem 4.1.1)

```
÷-trans : ∀ {s₁ s₂ s₃ : Store} (h₁ : s₁ ÷ s₂) (h₂ : s₂ ÷ s₃) → s₁ ÷ s₃
÷-trans h₁ h₂ id∈σ = h₂ (h₁ id∈σ)
```

see code snippet a.2.1

Proof A.2.2 (for Lemma 4.2.1)

```
ceval⇓⇒⊑ᵘ : ∀ (c : Command) (s s' : Store) (h⇓ : (ceval c s) ⇓ s')
              → s ⊑ᵘ s'
ceval⇓⇒⊑ᵘ skip s .s (nowj refl) x = x
ceval⇓⇒⊑ᵘ (assign id a) s s' h⇓ {id₁} x
 with (aeval a s)
... | just v
 with h⇓
... | nowj refl
 with (id == id₁) in eq-id
... | true rewrite eq-id = v , refl
... | false rewrite eq-id = x
ceval⇓⇒⊑ᵘ (ifelse b cᵗ cᶠ) s s' h⇓ x
 with (beval b s) in eq-b
... | just true rewrite eq-b = ceval⇓⇒⊑ᵘ cᵗ s s' h⇓ x
... | just false rewrite eq-b = ceval⇓⇒⊑ᵘ cᶠ s s' h⇓ x
ceval⇓⇒⊑ᵘ (seq c₁ c₂) s s' h⇓ {id}
 with (bindxf⇓⇒x⇓ {x = ceval c₁ s} {f = ceval c₂} h⇓)
... | sⁱ , c₁⇓sⁱ
 with (bindxf⇓-x⇓⇒f⇓ {x = ceval c₁ s} {f = ceval c₂} h⇓ c₁⇓sⁱ)
... | c₂⇓s' =
  ⊑ᵘ-trans (ceval⇓⇒⊑ᵘ c₁ s sⁱ c₁⇓sⁱ {id})
    (ceval⇓⇒⊑ᵘ c₂ sⁱ s' c₂⇓s' {id}) {id}
ceval⇓⇒⊑ᵘ (while b c) s s' h⇓ {id} x
 with (beval b s) in eq-b
... | just false with h⇓
... | nowj refl = x
ceval⇓⇒⊑ᵘ (while b c) s s' h⇓ {id} x
 | just true rewrite eq-b =
  while-⊑ᵘ c b s s' (λ s₁ s₂ h → ceval⇓⇒⊑ᵘ c s₁ s₂ h) h⇓ {id} x
```

see code snippet a.2.2

Proof A.2.3 (for Theorem 4.3.1)

```
cf-ext : ∀ {s₁ s₂ : CharacteristicFunction} → (a-ex : ∀ x → s₁ x ≡ s₂ x) → s₁ ≡ s₂
cf-ext a-ex = ext a Agda.Primitive.lzero a-ex
```

see code snippet a.2.3

Proof A.2.4 (for Lemma 4.3.1)

```
ceval⇓⇒sc⊆s' :  ∀ (c : Command) (s s' : Store) (h⇓ : (ceval c s) ⇓ s') → (dom s ∪
(cvars c)) ⊆ (dom s')
ceval⇓⇒sc⊆s' skip s .s (now refl) x x-in-s₁ rewrite (cvars-skip) rewrite (v-identityʳ
(dom s x)) = x-in-s₁
ceval⇓⇒sc⊆s' (assign id a) s s' h⇓ x x-in-s₁ with (aeval a s)
... | nothing with h⇓
... | now ()
ceval⇓⇒sc⊆s' (assign id a) s s' h⇓ x x-in-s₁ | just v with h⇓
... | now refl
 with (id == x) in eq-id
... | true = refl
... | false rewrite eq-id rewrite (v-identityʳ (dom s x)) with s x in eq-sx
... | just x₁ rewrite eq-sx = refl
ceval⇓⇒sc⊆s' (ifelse b cᵗ cᶠ) s s' h⇓ x x-in-s₁ with (beval b s) in eq-b
... | nothing with h⇓
... | now ()
ceval⇓⇒sc⊆s' (ifelse b cᵗ cᶠ) s s' h⇓ x x-in-s₁ | just false rewrite eq-b
 = ceval⇓⇒sc⊆s' cᶠ s s' h⇓ x (h {dom s x} {cvars cᵗ x} {cvars cᶠ x} x-in-s₁)
ceval⇓⇒sc⊆s' (ifelse b cᵗ cᶠ) s s' h⇓ x x-in-s₁ | just true rewrite eq-b
 = ceval⇓⇒sc⊆s' cᵗ s s' h⇓ x (h {dom s x} {cvars cᵗ x} {cvars cᶠ x}  x-in-s₁)
ceval⇓⇒sc⊆s' (seq c₁ c₂) s s' h⇓ x x-in-s₁
 with (bindxf⇓⇒x⇓ {x = ceval c₁ s} {f = ceval c₂} h⇓)
... | sⁱ , c₁⇓sⁱ with (bindxf⇓-x⇓⇒f⇓ {x = ceval c₁ s} {f = ceval c₂} h⇓ c₁⇓sⁱ)
... | c₂⇓s' with (ceval⇓⇒sc⊆s' c₁ s sⁱ c₁⇓sⁱ x)
... | n with (ceval⇓⇒sc⊆s' c₂ sⁱ s' c₂⇓s' x)
... | n' with (dom s x) | (cvars c₁ x) | (cvars c₂ x)
... | false | false | true rewrite (v-zeroʳ (dom sⁱ x)) = n' refl
... | false | true | false rewrite (v-zeroˡ (false)) rewrite (v-identityʳ (dom sⁱ x)) =
n' (n refl)
... | false | true | true rewrite (v-zeroˡ (false)) rewrite (v-zeroʳ (dom sⁱ x)) = n'
refl
... | true | n2 | n3 rewrite (v-zeroˡ (true)) rewrite (n refl) rewrite (v-identityʳ (dom
sⁱ x))
 = n' refl
ceval⇓⇒sc⊆s' (while b c) s s' h⇓ x x-in-s₁ rewrite (cvars-while {b} {c})
 rewrite (v-identityʳ (dom s x)) = ceval⇓⇒⊆ (while b c) s s' h⇓ x x-in-s₁
```

see code snippet a.2.4

Proof A.2.5 (for Theorem 4.3.5)

```
adia-safe : ∀ (a : AExp) (s : Store) → (dia : avars a ⊆ dom s) → (∃ λ v → aeval a s
≡ just v)
adia-safe (const n) s dia = n , refl
adia-safe (var id) s dia
 with (avars (var id) id) in eq-avars-id
... | false rewrite (==-refl {id}) with eq-avars-id
... | ()
adia-safe (var id) s dia | true = in-dom-has-value {s} {id} (dia id eq-avars-id)
adia-safe (plus a₁ a₂) s dia
 with (adia-safe a₁ s (⊆-trans (⊏ᵃ⇒⊆ a₁ (plus a₁ a₂) (plus-l a₁ a₂)) dia))
... | v₁ , eq-aev-a₁
 with (adia-safe a₂ s (⊆-trans (⊏ᵃ⇒⊆ a₂ (plus a₁ a₂) (plus-r a₁ a₂)) dia))
... | v₂ , eq-aev-a₂ rewrite eq-aev-a₁ rewrite eq-aev-a₂ = v₁ + v₂ , refl
```

see code snippet a.2.5

Proof A.2.6 (for Theorem 4.3.6)

```
bdia-safe : ∀ (b : BExp) (s : Store) → (dia : bvars b ⊆ dom s) → (∃ λ v → beval b s
≡ just v)
bdia-safe (const b) s dia = b , refl
bdia-safe (le a₁ a₂) s dia
 with (adia-safe a₁ s (⊆-trans (⊏ᵇᵃ⇒⊆ a₁ (le a₁ a₂) (le-l a₁ a₂)) dia))
  | (adia-safe a₂ s (⊆-trans (⊏ᵇᵃ⇒⊆ a₂ (le a₁ a₂) (le-r a₁ a₂)) dia))
... | v₁ , eq-a₁ | v₂ , eq-a₂ rewrite eq-a₁ rewrite eq-a₂ = (v₁ ≤ᵇ v₂) , refl
bdia-safe (BExp.not b) s dia
 with (bdia-safe b s (⊆-trans (⊏ᵇᵇ⇒⊆ b (BExp.not b) (_⊏ᵇ_.not b)) dia))
... | v , eq-b rewrite eq-b = (Data.Bool.not v) , refl
bdia-safe (and b₁ b₂) s dia
 with (bdia-safe b₁ s (⊆-trans (⊏ᵇᵇ⇒⊆ b₁ (and b₁ b₂) (and-l b₁ b₂)) dia))
  | (bdia-safe b₂ s (⊆-trans (⊏ᵇᵇ⇒⊆ b₂ (and b₁ b₂) (and-r b₁ b₂)) dia))
... | v₁ , eq-b₁ | v₂ , eq-b₂ rewrite eq-b₁ rewrite eq-b₂ = (v₁ ∧ v₂) , refl
```

see code snippet a.2.6

Proof A.2.7 (for Theorem 4.3.7)

```
dia-safe : ∀ (c : Command) (s : Store) (v v' : VarsSet) (dia : Dia v c v') (v⊆s : v ⊆
dom s) → (h-err : (ceval c s) ↯) → ⊥
dia-safe skip s v v' dia v⊆s (now ())
dia-safe (assign id a) s v .(id ↦ v) (assign .a .v .id a⊆v) v⊆s h-err with (adia-safe a
s (⊆-trans a⊆v v⊆s)) ... | a' , eq-aeval with h-err
... | now ()
dia-safe (ifelse b cᵗ cᶠ) s v .(vᵗ ∩ vᶠ) (if .b .v vᵗ vᶠ .cᵗ .cᶠ b⊆v diaᶠ diaᵗ) v⊆s h-
err with (bdia-safe b s λ x x-in-s₁ → v⊆s x (b⊆v x x-in-s₁))
... | false , eq-beval rewrite eq-beval rewrite eq-beval = dia-safe cᶠ s v vᶠ diaᶠ v⊆s
h-err
dia-safe (ifelse b cᵗ cᶠ) s v .(vᵗ ∩ vᶠ) (if .b .v vᵗ vᶠ .cᵗ .cᶠ b⊆v diaᶠ diaᵗ) v⊆s h-
err | true , eq-beval rewrite eq-beval rewrite eq-beval = dia-safe cᵗ s v vᵗ diaᵗ v⊆s
h-err
dia-safe (seq c₁ c₂) s v₁ v₃ dia v⊆s h-err with dia
... | seq .v₁ v₂ .v₃ .c₁ .c₂ dia-c₁ dia-c₂ with (ceval c₁ s) in eq-ceval-c₁
... | now nothing = dia-safe c₁ s v₁ v₂ dia-c₁ v⊆s (≡⇒≋ eq-ceval-c₁)
... | now (just s') rewrite eq-ceval-c₁ = dia-safe c₂ s' v₂ v₃ dia-c₂ (dia-ceval⇒⊆ dia-
c₁ v⊆s (≡⇒≋ eq-ceval-c₁)) h-err
dia-safe (seq c₁ c₂) s v₁ v₃ dia v⊆s h-err | seq .v₁ v₂ .v₃ .c₁ .c₂ dia-c₁ dia-c₂  |
later x with (dia-safe c₁ s v₁ v₂ dia-c₁ v⊆s)
... | c₁↯⊥ rewrite eq-ceval-c₁ = dia-safe-seq-later c₁↯⊥ dia-c₂ h h-err
dia-safe (while b c) s v v' dia v⊆s h-err with dia
... | while .b .v v₁ .c b⊆s dia-c with (bdia-safe b s (λ x x-in-s₁ → v⊆s x (b⊆s x x-in-
s₁)))
... | false , eq-beval rewrite eq-beval with h-err
... | now ()
dia-safe (while b c) s v v' dia v⊆s h-err | while .b .v v₁ .c b⊆s dia-c | true , eq-
beval with (ceval c s) in eq-ceval-c
... | now nothing = dia-safe c s v v₁ dia-c v⊆s (≡⇒≋ eq-ceval-c)
dia-safe (while b c) s v v' dia v⊆s h-err | while .b .v v₁ .c b⊆s dia-c | true , eq-
beval | now (just s') rewrite eq-beval rewrite eq-ceval-c with h-err
... | later₁ w↯ = dia-safe (while b c) s' v v dia (⊆-trans v⊆s (ceval⇓⇒⊆ c s s' (≡⇒≋
eq-ceval-c))) w↯
dia-safe (while b c) s v v' dia v⊆s h-err | while .b .v v₁ .c b⊆s dia-c | true , eq-
beval | later x with (dia-safe c s v v₁ dia-c v⊆s)
... | c↯⊥ rewrite eq-beval rewrite eq-ceval-c = dia-safe-while-later c↯⊥ dia h h-err
```

see code snippet a.2.7

Proof A.2.8

```
dia-sound-while-later : ∀ {x : Thunk (Delay (Maybe Store)) ∞} {b c} {v} (l↯⊥ : (later
x)↯ → ⊥)
   (dia : Dia v (while b c) v) (l⇓s⇒⊆ : ∀ {s : Store} → ((later x) ⇓ s) → v ⊆ dom s)
   (w↯ : (bind (later x) (λ s → later (ceval-while c b s))) ↯) → ⊥
  dia-sound-while-later {x} {b} {c} {v} l↯⊥ dia l⇓s⇒⊆ w↯ with (force x) in eq-force-x
  ... | now nothing = l↯⊥ (later₁ (≡⇒≋ eq-force-x))
  dia-sound-while-later {x} {b} {c} {v} l↯⊥ dia l⇓s⇒⊆ w↯ | now (just s') with w↯
  ... | later₁ w↯' rewrite eq-force-x with w↯'
  ... | later₁ w↯'' = dia-sound (while b c) s' v v dia (l⇓s⇒⊆ (later₁ (≡⇒≋ eq-force-
x))) w↯''
  dia-sound-while-later {x} {b} {c} {v} l↯⊥ dia l⇓s⇒⊆ w↯ | later x₁ with w↯
  ... | later₁ w↯' rewrite eq-force-x with w↯'
  ... | later₁ w↯'' = dia-sound-while-force {x₁} fx↯⇒⊥ dia fx⇓⇒⊆ w↯''
   where
    lx↯⇒⊥ : (hl : (later x₁) ↯) → ⊥
    lx↯⇒⊥ hl rewrite (sym eq-force-x) = l↯⊥ (later₁ hl)
    fx↯⇒⊥ : (h : (force x₁) ↯) → ⊥
    fx↯⇒⊥ h = lx↯⇒⊥ (later₁ {xs = x₁} h)
    lx⇓⇒⊆ : ∀ {s : Store} → (lx₁⇓s : later x₁ ⇓ s) → v ⊆ dom s
    lx⇓⇒⊆ lx₁⇓s rewrite (sym eq-force-x) =  l⇓s⇒⊆ (later₁ lx₁⇓s)
    fx⇓⇒⊆ : ∀ {s : Store} → (fx₁⇓s : force x₁ ⇓ s) → v ⊆ dom s
    fx⇓⇒⊆ fx₁⇓s = lx⇓⇒⊆ (later₁ {xs = x₁} fx₁⇓s)
```

see code snippet a.2.8

Proof A.2.9 (for Theorem 4.3.8)

```
-- Pure constant folding preserves semantics.
apfold-safe : ∀ a s → (aeval a s ≡ aeval (apfold a) s)
apfold-safe (const n) _ = refl
apfold-safe (var id) _ = refl
apfold-safe (plus a₁ a₂) s
 rewrite (apfold-safe a₁ s)
 rewrite (apfold-safe a₂ s)
 with (apfold a₁) in eq-a₁ | (apfold a₂) in eq-a₂
... | const n | const n₁  = refl
... | const n | var id     = refl
... | const n | plus v₂ v₃ = refl
... | var id     | v₂ = refl
... | plus v₁ v₃ | v₂ = refl
```

see code snippet a.2.9

Proof A.2.10 (for Theorem 4.3.9)

```
bpfold-safe : ∀ b s  → (beval b s ≡ beval (bpfold b) s)
bpfold-safe (const b) s  = refl
bpfold-safe (le a₁ a₂) s rewrite (apfold-safe a₁ s) rewrite (apfold-safe a₂ s)
 with (apfold a₁) | (apfold a₂)
... | const n | const n₁ = refl
... | const n | var id = refl
... | const n | plus v₂ v₃ = refl
... | var id | v₂ = refl
... | plus v₁ v₃ | v₂ = refl
bpfold-safe (not b) s rewrite (bpfold-safe b s) with (bpfold b)
... | const b₁ = refl
... | le a₁ a₂ = refl
... | not v = refl
... | and v v₁ = refl
bpfold-safe (and b₁ b₂) s rewrite (bpfold-safe b₁ s) rewrite (bpfold-safe b₂ s)
 with (bpfold b₁) | (bpfold b₂)
... | const b | const b₃ = refl
... | const b | le a₁ a₂ = refl
... | const b | not v₂ = refl
... | const b | and v₂ v₃ = refl
... | le a₁ a₂ | v₂ = refl
... | not v₁ | v₂ = refl
... | and v₁ v₃ | v₂ = refl
```

see code snippet a.2.10

Proof A.2.11 (for Theorem 4.3.10)

```
cpfold-safe : ∀ (c : Command) (s : Store) → ∞ ⊢ (ceval c s) ≋ (ceval (cpfold c) s)
cpfold-safe skip s rewrite (cpfold-skip) = now refl
cpfold-safe (assign id a) s = ≡⇒≋ (cpfold-assign a id s)
cpfold-safe (ifelse b cᵗ cᶠ) s = cpfold-if b cᵗ cᶠ s
cpfold-safe (seq c₁ c₂) s = cpfold-seq c₁ c₂ s
cpfold-safe (while b c) s = cpfold-while b c s
cpfold-assign : ∀ (a : AExp) (id : Ident) (s : Store)
 → (ceval (assign id a) s) ≡ (ceval (cpfold (assign id a)) s)
cpfold-assign a id s
 with (apfold-sound a s)
... | asound
 with (aeval a s) in eq-av
... | nothing
 rewrite eq-av
 rewrite (eqsym asound)
 with (apfold a) in eq-ap
... | var id₁ rewrite eq-ap rewrite eq-av rewrite eq-av = eqrefl
... | plus n n₁ rewrite eq-ap rewrite eq-av rewrite eq-av = eqrefl
cpfold-assign a id s | asound | just x
 rewrite eq-av
 rewrite (eqsym asound)
 with (apfold a) in eq-ap
... | var id₁ rewrite eq-ap rewrite eq-av rewrite eq-av = eqrefl
cpfold-assign a id s | asound | just x | plus n n₁
 rewrite eq-ap rewrite eq-av rewrite eq-av = eqrefl
cpfold-assign a id s | asound | just x | const n
 rewrite eq-ap rewrite eq-av rewrite eq-av
 with asound
... | eqrefl = eqrefl
cpfold-if : ∀ (b : BExp) (cᵗ cᶠ : Command) (s : Store)
 → ∞ ⊢ (ceval (ifelse b cᵗ cᶠ) s) ≋ (ceval (cpfold (ifelse b cᵗ cᶠ)) s)
cpfold-if b cᵗ cᶠ s
 with (bpfold-sound b s)
... | bsound
 with (beval b s) in eq-b
... | nothing
 rewrite eq-b
 rewrite (eqsym bsound)
 with (bpfold b) in eq-bp
... | le a₁ a₂ rewrite eq-bp rewrite eq-b rewrite eq-b = now eqrefl
... | not n rewrite eq-bp rewrite eq-b rewrite eq-b = now eqrefl
... | and n n₁ rewrite eq-bp rewrite eq-b rewrite eq-b = now eqrefl
-- cont
```

see code snippet a.2.11

Ringraziamenti

Ringrazio innanzitutto il mio relatore Alberto: in questi mesi i suoi consigli mi sono stati molto utili. Oltre alla scrittura della tesi, senza le sue lezioni non mi sarei appassionato a questo ambito dell'informatica tanto quanto ora mi accorgo di esserlo.

Grazie ai miei genitori, Lidia e Franco. Senza i loro sforzi non sarei mai potuto arrivare a questo importante traguardo. Grazie per avermi sostenuto nei momenti in cui non credevo in me stesso.

Grazie ai miei amici Luca, gli A. M., Luca (chi è quale?), Elisa, Stefano e, infine, Emma. Il tempo passato con voi e le parole scambiate assieme sono incredibilmente importanti per me, vi voglio bene.

Bibliography

[1] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, Cambridge, MA, USA: MIT Press, 1993.

[2] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Methods Program.*, pp. 17–139, 2004.

[3] G. Kahn, "Natural semantics," in *STACS 87, 4th Annu. Symp. Theor. Aspects Comput. Science, Passau, Germany, February 19-21, 1987, Proc.* in Lecture Notes in Computer Science, vol. 247, 1987, pp. 22–39, doi: 10.1007/BFb0039592. [Online]. Available: https://doi.org/10.1007/BFb0039592

[4] X. Leroy, and H. Grall, "Coinductive big-step operational semantics," *Inf. Comput.*, vol. 207, no. 2, pp. 284–304, 2009, doi: 10.1016/j.ic.2007.12.004. [Online]. Available: https://doi.org/10.1016/j.ic.2007.12.004

[5] F. Allen, "A catalogue of optimizing transformation," in *Des. Optim. Compilers*, 1972.

[6] LLVM Project, *The LLVM Compiler Infrastructure*, (2021). [Online]. Available: https://llvm.org/

[7] J. Lee, C.-K. Hur, and N. P. Lopes, "AliveInLean: a verified LLVM peephole optimization verifier," in *Proc. 31st Int. Conf. Computer-Aided Verification (CAV)*, Jul. 2019, doi: 10.1007/978-3-030-25543-5_25.

[8] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," *SIGPLAN Not.*, vol. 47, no. 1, p. 427, Jan. 2012, doi: 10.1145/2103621.2103709. [Online]. Available: https://doi.org/10.1145/2103621.2103709

[9] Y. Zakowski, C. Beck, et al., "Modular, compositional, and executable formal semantics for LLVM IR," *Proc. ACM Program. Languages*, vol. 5, no. ICFP, pp. 1–30, Aug. 2021, doi: 10.1145/3473572. [Online]. Available: https://doi.org/10.1145/3473572

[10] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, 2009, doi: 10.1007/s10817-009-9155-4. [Online]. Available: https://doi.org/10.1007/s10817-009-9155-4

[11] T. C. D. Team, “The coq proof assistant,” Zenodo, 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8161141

[12] N. A. Danielsson, “Operational semantics using the partiality monad,” in *ACM SIGPLAN Int. Conf. Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, 2012, pp. 127–138, doi: 10.1145/2364527.2364546. [Online]. Available: https://doi.org/10.1145/2364527.2364546

[13] C. Coquand, D. Synek, and M. Takeyama, “An emacs-interface for type-directed support for constructing proofs and programs.”

[14] U. Norell, “Towards a practical programming language based on dependent type theory,” Thesis, Dept. Comput. Sci. Engineering, Chalmers Univ. Technol., SE-412 96 Göteborg, Sweden, 2007.

[15] H. Geuvers, “Proof assistants: history, ideas and future,” *Sadhana*, vol. 34, no. 1, pp. 3–25, Feb. 2009, doi: 10.1007/s12046-009-0001-5. [Online]. Available: https://doi.org/10.1007/s12046-009-0001-5

[16] H. B. Curry, “Functionality in combinatory logic,” *Proc. Nat. Acad. Sciences*, vol. 20, no. 11, pp. 584–590, Nov. 1934, doi: 10.1073/pnas.20.11.584. [Online]. Available: https://doi.org/10.1073/pnas.20.11.584

[17] W. A. Howard, “The formulae-as-types notion of construction,” in *H. B. Curry: Essays Combinatory Logic, Lambda Calculus, Formalism*, H. Curry, H. B., S. J. Roger, and P. Jonathan, Eds., Academic Press.

[18] A. Heyting, *Intuitionism: An Introduction*, North-Holland Publishing Company, 1966. [Online]. Available: https://books.google.it/books?id=4rhLAAAAMAAJ

[19] A. Kolmogoroff, “Zur deutung der intuitionistischen logik,” *Mathematische Zeitschrift*, vol. 35, no. 1, pp. 58–65, Dec. 1932, doi: 10.1007/bf01186549. [Online]. Available: https://doi.org/10.1007/bf01186549

[20] A. S. Troelstra, *Principles of Intuitionism*, Springer Berlin Heidelberg, 1969. [Online]. Available: https://doi.org/10.1007/bfb0080643

[21] N. G. de Bruijn, “Automath, a language for mathematics,” J. H. Siekmann, and G. Wrightson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 159–200. [Online]. Available: https://doi.org/10.1007/978-3-642-81955-1_11

[22] T. Coquand, and G. Huet, “The calculus of constructions,” *Inf. Computation*, vol. 76, no. 2, pp. 95–120, 1988, doi: https://doi.org/10.1016/0890-5401(88)90005-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540188900053

[23] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013. [Online]. Available: https://homotopytypetheory.org/book

[24] U. Norell, A. Abel, and others, "Agda documentation," 2023. [Online]. Available: https://agda.readthedocs.io/en/v2.6.3.20230914/index.html

[25] F. Acerbi, "Plato: parmenides 149a7-c3. a proof by complete induction?," *Arch. Hist. Exact Sciences*, vol. 55, no. 1, pp. 57–76, Aug. 2000, doi: 10.1007/s004070000020. [Online]. Available: https://doi.org/10.1007/s004070000020

[26] D. Sangiorgi, "Bisimulation: from the origins to today," in *19th IEEE Symp. Log. Comput. Sci. (LICS 2004), 14-17 July 2004, Turku, Finland, Proc.*, 2004, pp. 298–302, doi: 10.1109/LICS.2004.1319624. [Online]. Available: https://doi.org/10.1109/LICS.2004.1319624

[27] D. Sangiorgi, and J. J. M. M. Rutten, Eds., *Advanced Topics in Bisimulation and Coinduction*, vol. 52, Cambridge University Press, 2012. [Online]. Available: http://www.cambridge.org/gb/knowledge/isbn/item6542021

[28] D. Pous, "Coinduction all the way up," in *Proc. 31st Annu. ACM/IEEE Symp. Log. Comput. Science, LICS '16, New York, NY, USA, July 5-8, 2016*, 2016, pp. 307–316, doi: 10.1145/2933575.2934564. [Online]. Available: https://doi.org/10.1145/2933575.2934564

[29] B. C. Pierce, *Types and Programming Languages*, MIT Press, 2002.

[30] J. Hughes, L. Pareto, and A. Sabry, "Proving the correctness of reactive systems using sized types," in *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Princ. Program. Languages* in Popl '96, St. Petersburg Beach, Florida, USA, 1996, p. 410, doi: 10.1145/237721.240882. [Online]. Available: https://doi.org/10.1145/237721.240882

[31] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu, "Type-based termination of recursive definitions," *Math. Structures Comput. Sci.*, vol. 14, no. 1, p. 97, 2004.

[32] F. Blanqui, "A type-based termination criterion for dependently-typed higher-order rewrite systems," 2004, pp. 24–39, doi: 10.1007/978-3-540-25979-4_2.

[33] A. Abel, "Miniagda: integrating sized and dependent types," in *Partiality Recursion Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010* in Epic Series, vol. 5, 2010, pp. 18–33, doi: 10.29007/322q. [Online]. Available: https://doi.org/10.29007/322q

[34] A. Abel, "Type theory - coinduction in type theory," ESSLLI 2016. [Online]. Available: https://www.cse.chalmers.se/~abela/esslli2016/talkESSLLI-coinduction.pdf

[35] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer, "Copatterns: programming infinite structures by observations," *SIGPLAN Not.*, vol. 48, no. 1, p. 27, Jan. 2013, doi: 10.1145/2480359.2429075. [Online]. Available: https://doi.org/10.1145/2480359.2429075

[36] Agda, "Github - agda/agda: agda is a dependently typed programming language and interactive theorem prover. [Online]. Available: https://github.com/agda/agda

[37] A. Vezzosi, A. Mörtberg, and A. Abel, "Cubical agda: A dependently typed programming language with univalence and higher inductive types," *J. Funct. Program.*, vol. 31, 2021, doi: 10.1017/S0956796821000034. [Online]. Available: https://doi.org/10.1017/S0956796821000034

[38] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical type theory: a constructive interpretation of the univalence axiom," *Corr*, 2016. [Online]. Available: http://arxiv.org/abs/1611.02108

[39] E. Moggi, "Computational lambda-calculus and monads," in *Proc. Fourth Annu. Symp. Log. Comput. Sci. (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, 1989, pp. 14–23, doi: 10.1109/LICS.1989.39155. [Online]. Available: https://doi.org/10.1109/LICS.1989.39155

[40] C. Kohl, and C. Schwaiger, "Monads in computer science," 2021. [Online]. Available: https://ncatlab.org/nlab/files/KohlSchwaiger-Monads.pdf

[41] V. Capretta, "General recursion via coinductive types," *Log. Methods Comput. Sci.*, vol. 1, no. 2, 2005, doi: 10.2168/LMCS-1(2:1)2005. [Online]. Available: https://doi.org/10.2168/LMCS-1(2:1)2005

[42] A. Abel, and J. Chapman, "Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types," in *Proc. 5th Workshop Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014* in EPTCS, vol. 153, 2014, pp. 51–67, doi: 10.4204/EPTCS.153.4. [Online]. Available: https://doi.org/10.4204/EPTCS.153.4

[43] B. C. Pierce, A. Azevedo de Amorim, et al., *Software Foundations*, Electronic textbook, 2017. [Online]. Available: https://softwarefoundations.cis.upenn.edu/

[44] T. Nipkow, and G. Klein, *Concrete Semantics - With Isabelle/hol*, Springer, 2014. [Online]. Available: https://doi.org/10.1007/978-3-319-10542-0

[45] K. Nakata, and T. Uustalu, "Trace-based coinductive operational semantics for while," in *Theorem Proving Higher Order Logics, 22nd Int. Conference, Tphols 2009, Munich, Germany, August 17-20, 2009. Proc.* in Lecture Notes in Computer Science, vol.

5674, 2009, pp. 375–390, doi: 10.1007/978-3-642-03359-9_26. [Online]. Available: https://doi.org/10.1007/978-3-642-03359-9/_26

[46] P. Bahr, and G. Hutton, "Monadic compiler calculation (functional pearl)," *Proc. ACM Program. Lang.*, vol. 6, no. ICFP, Aug. 2022, doi: 10.1145/3547624. [Online]. Available: https://doi.org/10.1145/3547624

[47] X. Leroy, "Mechanized semantics." https://github.com/xavierleroy/cdf-mech-sem