Program Transformations in the Delay Monad

A Case Study for Coinduction via Copatterns and Sized Types

Edoardo Marangoni
University of Milan

A thesis submitted for the degree of
Master of Science

`datetime(year: 2023, month: 9, day: 25)`

"...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

TODOs

1: Tell more about where this definition comes from

2: To cite: Danielsson's operational semantics - Leroy's coinductive big step operational semantics - Abel (various) - Hutton's Calculating dependently-typed compilers (functional pearl)

CHAPTER I

Introduction

CHAPTER 2

Induction and coinduction

All throughout this work we make use of the mathematical technique called *coinduction*: it is far from easy to choose 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 means 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 will start by examining this last possibility, as our use of coinduction is "limited" to coinductive datatypes and proofs by corecursion: we will thus introduce these concepts with their role as dual to inductive datatypes and recursion.

2.1 Recursive datatypes

The easiest and most intuitive inductive datatype is that of natural numbers. In type theory, one may represent them as follows:

$$\frac{}{\mathbb{N} : \text{Type}}\ \text{type formation}$$

$$\frac{}{0 : \mathbb{N}}\ \text{zero} \qquad \frac{n : \mathbb{N}}{S\,n : \mathbb{N}}\ \text{succ}$$

Or, in Agda:

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

We can imagine concrete instances of this datatype as trees reflecting the structure of the constructors as shown in Figure 2. We might want to show some properties of these inductive datatypes, and the tool to do so is the *principle of induction*.

2.2 Infinite proofs

2.3 Agda

In this section we will introduce the Agda programming language, a dependently typed programming language and proof assistant. We briefly introduce what proof assistants

$$\begin{tikzcd} \dots \\ \text{succ succ zero} \arrow[u] \\ \text{succ zero} \arrow[u] \\ \text{zero} \arrow[u] \end{tikzcd}$$

Figure 2: Structure of a natural number as tree of constructors

are, what makes proof assistants such as Agda useful and how Agda in particular deals with coinduction.

2.3.1 Type systems

Agda is both a proof assistant and a programming language. It achieves so by exploiting interactive programming and a powerful type system, namely a *dependent* one. In brief, such a type system allows types to depend on terms; it is however worth to analyse a bit more the concepts of type systems whole.

2.3.2 Termination and productivity

2.3.3 Sized types

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 (Moggi 1989) 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 trivial 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
```

from Agda's stdlib

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
```

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 means to model a behaviour of the execution, or

effects: in fact, they're also called "computation builders" in the context of programming. Let us give an example:

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

from Agda's stdlib

The underlying idea of monads in the context of computer science, as explained by Moggi in (Moggi 1989), is to describe "notions of computations" that may have consequences comparable to *side effects* of imperative programming languages 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 inspired by (Kohl and Schwaiger 2021).

Definition 3.1.1.1 (Monad): Let A, B and C be types. A monad M is defined as the triple (`M`, `unit`, `_>>=_`) where `M` is a monadic constructor denoting some side-effect or impure behaviour; `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 (Capretta 2005). As described in (Abel and Chapman 2014), 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 2.3.3):

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

from Agda's stdlib

We equip with the following `bind` function:

```
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
```

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. Of course, 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
```

snippet 3.3.1 see code

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
```

snippet 3.3.2 see code

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. Of course, 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 ∞
```

snippet 3.3.3 see code

We thus define an equivalence relation on `Delay` which we call **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[1]. The definition we give in Definition 3.3.1 follows those given by

! TODO !

Tell more about where this definition comes from

[1] **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.

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 sized to deal with coinductive definitions (see Subsection 2.3.3).

```
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)
```

snippet 3.3.4 see code

This definition also allows us to abstract over the relation between the values of the parameter type A and have a single definition for multiple kinds of relations. Propositional equality is still the most frequently used relation, so we define a special notation for this specialization:

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

snippet 3.3.5 see code

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.

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 i) (WeakBisim Q i) (WeakBisim R i)
```

see code see proof

Theorem 3.3.2 affirms 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 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

3.4 Convergence, divergence and failure

Now that we have a means to relate computations, we also want to define propositions to characterize them. The `Delay` monads allows us to model the effect of non-termination, but we also want to model the behaviour of program 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)
```

snippet 3.4.1 see code

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

Definition 3.4.1 (Converging program):

```
_⇓_ : ∀ (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

We then define a program 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 program):

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

see code

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

Definition 3.4.3 (Failing program):

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

see code

We can say that a program, in our semantics, cannot show any other kind of behaviour, therefore theorem Theorem 3.4.1 seems clearly true; in a constructive environment like Agda we can, however, only postulate it.

Theorem 3.4.1:

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

see code

CHAPTER 4

The Imp programming language

In this chapter we will go over the implementation of a simple imperative language called **Imp**, as described in (Pierce et al. 2023). After defining its syntax, we will give rules for its semantics and show its implementation in Agda. After this introductory work, we will discuss analysis and optimization of Imp programs.

4.1 Introduction

The Imp language was devised to work as a simple example of an imperative language; albeit having a handful of syntactic constructs, it clearly is a Turing complete language.

4.1.1 Syntax

The syntax of the Imp language can be described in a handful of EBNF rules, as shown in Table 3.

$$\textbf{aexp} \coloneqq n \mid \text{id} \mid a_1 + a_2$$
$$\textbf{bexp} \coloneqq b \mid a_1 < a_2 \mid \neg b \mid b_1 \land b_2$$
$$\textbf{command} \coloneqq \text{skip} \mid \text{id} \leftarrow \textbf{aexp} \mid c_1 ; c_2 \mid \text{if } \textbf{bexp} \text{ then } c_1 \text{ else } c_2 \mid \text{while } \textbf{bexp} \text{ do c}$$

Table 3: Syntax rules for the Imp language

The syntactic elements of this language are three: *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 them can be represented with simple datatypes enclosing all the information of the syntactic rules, as shown in snippet 4.1.1.1, snippet 4.1.1.2, snippet 4.1.1.3 and snippet 4.1.1.4.

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

snippet 4.1.1.1 see code

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

snippet 4.1.1.2 see code

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

snippet 4.1.1.3 see code

```
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
```

snippet 4.1.1.4 see code

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 means to keep track of identifiers and their value: this means is the `Store`, which we define in this section, while also giving some useful definition. Stores are defined as shown in snippet 4.1.2.1, that is, partial maps made total with the use of the `Maybe` monad.

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

snippet 4.1.2.1 see code

We now proceed to show some basic definition of *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's the same as $\exists\, v \in \mathbb{Z}, \sigma$ id $\equiv$ just v.
2. **empty store** we define the empty store as $\emptyset$. For this special store, it is always $\forall$ id, id $\in \emptyset \rightarrow \bot$ or $\forall$ id, $\emptyset$ id $\equiv$ nothing.

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

snippet 4.1.2.2 see code

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

snippet 4.1.2.3 see code

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

$$\exists\, \text{id}, \exists\, v_1, \exists\, v_2, v_1 \neq v_2 \land \sigma_1\, \text{id} \equiv \text{just}\, v_1 \land \sigma_2\, \text{id} \equiv \text{just}\, v_2$$

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

snippet 4.1.2.4 see code

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 σ_1 id $\equiv$ just v and σ_2 id $\equiv$ 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
```

snippet 4.1.2.5 see code

Definition 4.1.2.1: Let σ_1 and σ_2 be two stores. We define the unvalued inclusion between them as

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

and we denote it with $\sigma_1 \ \dot{\div}\ \sigma_2$. In Agda:

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

see code

Theorem 4.1.2.1 (Transitivity of ∻):

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

see code see proof

4.2 Semantics

Having understood the syntax of Imp, we can move to the *meaning* of Imp programs. We'll explore the operational semantics of the language using the formalism of inference rules, then we'll 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. As shown in Table 3, Imp programs are composed of three entities: arithmetic expression, boolean expression and commands.

```
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 (such as failures of type-checking) or from the dynamic execution of the program, 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're now able to give inference rules for each construct of the Imp language: we'll start from simple ones, that is arithmetic and boolean expressions, and we'll then move to commands.

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 expression 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 ($\cdot \Downarrow \cdot$ and $\cdot$ ↯)

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

Table 4: Inference rules for the semantics of arithmetic expressions of Imp

The Agda code implementing the interpreter for arithmetic expressions is shown in snippet 4.2.1.1. 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)
```

snippet 4.2.1.1 see code

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.

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

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

Table 5: 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 4.2.2.1.

```
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)
```

snippet 4.2.2.1 see code

4.2.3 Commands

The inference rules we give for commands follow the formalism of **big-step** operational semantics, that is, intermediate states of evaluation are not shown explicitly in the rules themselves.

$$\frac{}{\text{skip}, \sigma \Downarrow \sigma} \qquad \frac{a \Downarrow v}{\text{assign id}\ a, \sigma \Downarrow \text{update id}\ v\ \sigma} \qquad \frac{c_1, \sigma \Downarrow \sigma_1 \quad c_2, \sigma_1 \Downarrow \sigma_2}{\text{seq}\ c_1\ c_2, \sigma \Downarrow \sigma_2}$$

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

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

Table 6: Inference rules for the semantics of commands

We need to be careful when examining the inference rules in Table 6. 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 4.2.3.1.

```
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))
```

snippet 4.2.3.1 see code

The last rule (`while` for `beval b` convering to `just true`) is coinductive, and this is reflected in the code by having the computation happen inside a `Thunk`, that is, the actual tree of execution is expanded only when the computation is forced.

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 4.2.4.1: Let c be a command and σ_1 and σ_2 be two stores. Then

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

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

see code see proof

Lemma 4.2.4.1 will be fundamental for later proofs.

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 (that is, they are static), then showing proofs regarding the result of the execution of the program. The main inspiration for these operations is (Nipkow and Klein 2014).

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 the kind of failure, among many, 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`. Foremost, we give a (parametric) notion of members equivalence (that is, a function `_==_ : A → A → Bool`); then, we equip characteristic functions of the usual operations on sets: insertion, union, and intersection and the usual definition of inclusion.

```
ϕ : CharacteristicFunction
ϕ = 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
```

snippet 4.3.1.1.1 see code

Theorem 4.3.1.1.1 (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

Theorem 4.3.1.1.2 (Neutral element of union):

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

see code

Theorem 4.3.1.1.3 (Update inclusion):

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

see code

Theorem 4.3.1.1.4 (Transitivity of inclusion):

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

see code

We will also need a way to get a VarsSet from a Store, which is shown in snippet 4.3.1.1.6.

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

snippet 4.3.1.1.6 see code

Realization

Following (Nipkow and Klein 2014), the first formal tool we need is a means to compute the set of variables mentioned in expressions, shown in snippet 4.3.1.2.1 and snippet 4.3.1.2.2. We also need a function to compute the set of variables that are definitely initialized in commands, which is shown in snippet 4.3.1.2.3.

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

snippet 4.3.1.2.1 see code

```
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)
```

snippet 4.3.1.2.2 see code

```
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) =  ∅
```

snippet 4.3.1.2.3 see code

It is worth to reflect upon the definition of snippet 4.3.1.2.3. What this code does it compute the set of *initialized* variables in a command `c`; as done in (Nipkow and Klein 2014), we construct this set of initialized variables in the most careful 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 `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 a the evaluation of a command in a store σ converges to a value σ', that is $c, \sigma \Downarrow \sigma'$ then by Lemma 4.2.4.1 $\operatorname{dom} \sigma \subseteq \operatorname{dom} \sigma'$; this allows us to show the following lemma.

Lemma 4.3.1.2.1: Let c be a command and σ_1 and σ_2 be two stores. Then

$$\text{ceval}\ c\ \sigma_1 \Downarrow \sigma_2 \to (\operatorname{dom} \sigma_1 \cup (\text{cvars}\ c)) \subseteq (\operatorname{dom} \sigma_2)$$

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

see code   see proof

We now give inference rules that inductively build the relation that embodies the logic of the definite initialization analysis, shown in Table 7. In Agda, we define a datatype representing the relation of type `Dia : VarsSet → Command → VarsSet → Set`, which is shown in snippet 4.3.1.2.5. Lemma 4.3.1.2.4 will allow us to show that there's a relation between the `VarsSet` in the `Dia` relation and the actual stores that are used in the execution of a command.

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

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

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

Table 7: 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
```

snippet 4.3.1.2.5 see code

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 4.3.1.2.1 and Theorem 4.3.1.2.2 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 4.3.1.2.3 shows that if `Dia` holds, then the evaluation of a program failing is absurd: therefore, by Theorem 3.4.1, the program either diverges or converges to some value.

Theorem 4.3.1.2.1 (Soundness of definite initialization for arithmetic expressions):

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

see code see proof

Theorem 4.3.1.2.2 (Soundness of definite initialization for boolean expressions):

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

see code see proof

Theorem 4.3.1.2.3 (Soundness of definite initialization for commands):

```
dia-sound : ∀ (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

We now show an idea of the proof in a discursive manner (the full proof, in Agda, is in snippet 4.3.2.3.1.9). Let us start with the two simple (non-recursive) cases. The first is `c ≡ skip`, where we have the following situation:

```
 dia-sound' skip s v v' dia v⊆s h-err = {! !}
-- ────────────────────────────────────────────────────────────
-- Goal: ⊥
-- ────────────────────────────────────────────────────────────
-- dia : Dia v skip v'
-- h-err : WeakBisim _≡_ ∞ (ceval skip s) (now nothing)
-- s : Ident → Maybe ℤ
-- v v' : String → Bool
-- v⊆s : (x : String) → v x ≡ true → dom s x ≡ true
-- ────────────────────────────────────────────────────────────
```

Forcing Agda to inspect `h-err`, we get the assumption that `just s ≡ nothing`; inspecting this assumption we easily get snippet 4.3.1.2.10.

```
dia-sound skip s v v' dia v⊆s (now ())
```

snippet 4.3.1.2.10 see code

Let us move to the second "easy" case, assignment:

```
dia-sound' (assign id a) s v v' dia v⊆s h-err = {! !}
-- ______________________________________________________________
-- Goal: ⊥
-- ______________________________________________________________
-- a : AExp
-- dia : Dia v (assign id a) v'
-- h-err : WeakBisim _≡_ ∞ (ceval (assign id a) s) (now nothing)
-- id : String
-- s : Ident → Maybe ℤ
-- v v' : String → Bool
-- v⊆s : (x : String) → v x ≡ true → dom s x ≡ true
-- ______________________________________________________________
```

We can make Agda aware of what `v'` is, that is, `v' ≡ (id ↦ v)` by splitting on `dia`:

```
dia-sound' (assign id a) s v .(id ↦ v) (assign .a .v .id a⊆v) v⊆s h-err = {! !}
```

And using the value of `aeval a s` using `adia-sound` (Theorem 4.3.1.2.1) we conclude this piece of proof as shown in snippet 4.3.1.2.13.

```
dia-sound (assign id a) s v .(id ↦ v) (assign .a .v .id a⊆v) v⊆s h-err
 with (adia-sound a s (⊆-trans a⊆v v⊆s))
... | a' , eq-aeval
 rewrite eq-aeval
 rewrite eq-aeval
 with h-err
... | now ()
```

snippet 4.3.1.2.13 see code

Let us examine a bit more difficult case, `while`, as shown in snippet 4.3.1.2.14. In this case, we proceed with the value of `beval b s` using `bdia-sound` (Theorem 4.3.1.2.2) we get to two outcomes: if `beval b s ≡ just false`, then `ceval (while b c) s` converges to `s` itself, and we get to the same absurd hypothesis `just s ≡ nothing`, which concludes this branch of the proof. If, instead, `beval b s ≡ just true`, we consider the value of `ceval c s` (snippet 4.3.1.2.15), which can have three outcomes: it can fail, that is `ceval c s ≡ now nothing` and we conclude this branch of the proof with a recursive call on `dia-sound c`; it can converge to `ceval c s ≡ now (just s)'` and we conclude this branch of the proof with a recursive call on `dia-sound (while b c) s'`.

```
 dia-sound' (while b c) s v v' dia v⊆s h-err = {! !}
-- ______________________________________________________________
-- Goal: ⊥
-- ______________________________________________________________
-- b : BExp
-- c : Command
-- dia : Dia v (while b c) v'
-- h-err : WeakBisim _≡_ ∞ (ceval (while b c) s) (now nothing)
-- s : Ident → Maybe ℤ
-- v v' : String → Bool
-- v⊆s : (x : String) → v x ≡ true → dom s x ≡ true
-- ______________________________________________________________
```

snippet 4.3.1.2.14

```
dia-sound (while b c) s v v' dia v⊆s h-err | true
 with (ceval c s) in eq-ceval-c --- ⇐ here
... | now nothing = dia-sound c s v v₁ dia-c v⊆s (≡⇒≋ eq-ceval-c)
dia-sound (while b c) s v v' dia v⊆s h-err | true | now (just s')
 with h-err
... | later₁ w↯ =
 dia-sound (while b c) s' v v dia
   (⊆-trans v⊆s (ceval⇓⇒⊆ c s s' (≡⇒≋ eq-ceval-c))) w↯
dia-sound (while b c) s v v' dia v⊆s h-err | true | later x
 with (dia-sound c s v v₁ dia-c v⊆s)
... | c↯⊥  = dia-sound-while-later c↯⊥ dia h h-err
```

snippet 4.3.1.2.15 see code

If, finally, `ceval c s ≡ later x`, we use an helper function (a lemma), shown in snippet 4.3.1.2.16.

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

snippet 4.3.1.2.16 see code

What this last piece of code does is coinductively "unwind" the execution of `while b c` to check whether or not `ceval c` in a generic store `s` converges to a store `s'`; if so, then check that `ceval (while b c) s'` does not fail by checking that `ceval c s'` does not fail and so on; while if `ceval c s` fails, use the assumption that it can't fail (which is just a preventive call to `dia-sound`).

4.3.2 Pure constant folding optimization

Pure constant folding is the second and last transformation we considered. Again from (Nipkow and Klein 2014), 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 passage 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, what this optimization does is the following: 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 ($a \equiv$ 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 left untouched. The Agda code for the function `apfold` is shown in snippet 4.3.2.1.1.

```
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'
```

snippet 4.3.2.1.1 see code

Of course, what we want to show is that this optimization does not change the result of the evaluation (Theorem 4.3.2.1.1).

Theorem 4.3.2.1.1 (Soundness 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-sound : ∀ a s → (aeval a s ≡ aeval (apfold a) s)
```

see code see proof

Pure folding of boolean expressions

Pure folding of boolean expressions, which we define as a function `bpfold`, follows the same line of reasoning exposed in Chapter 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 4.3.2.2.1.

```
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
```

snippet 4.3.2.2.1 see code

As before, our objective is to show that evaluating a boolean expressions after the optimization yields the same result as the evaluation without optimization.

Theorem 4.3.2.2.1 (Soundness 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-sound : ∀ b s → (beval b s ≡ beval (bpfold b) s)
```

see code see proof

Pure folding of commands

Pure folding of commands builds on the definition of apfold and bpfold above combining the definitions as shown in snippet 4.3.2.3.1.

```
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) = while (bpfold b) (cpfold c)
```

snippet 4.3.2.3.1 see code

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 values converge to the same value or both diverge (Theorem 4.3.2.3.1).

Theorem 4.3.2.3.1 (Soundness 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-sound : ∀ (c : Command) (s : Store)
                 → ∞ ⊢ (ceval c s) ≈ (ceval (cpfold c) s)
```

see code see proof

Of course, what makes Theorem 4.3.2.3.1 different from the other soundess proofs in this chapter is that we cannot use propositional equality and we must instead use weak bisimilarity; we use the weak version as in terms of chains of `later` and `now`, if the optimization does indeed change the syntactic tree of the command, if the evaluation converges to a value it may do so in a different number of steps; for example, the program `while 1 < 0 do skip` will be optimized to `while false do skip`, resulting in a shorter evaluation, as `1 < 0` will not be evaluated at runtime.

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 (Danielsson 2012) and (Leroy and Grall 2008).

Our work implements an *imperative* language using *sized* types to target the *coinductive* `FailingDelay` monad. Many of the works in literature – for example (Danielsson 2012), (Leroy and Grall 2008), and (Abel 2010) – chose to implement lambda calculus and its typed variants. Furthermore, some of the cited articles target the `Delay` or `FailingDelay` monad, but not each of them uses sized types; for example, (Danielsson 2012) does not use sizes, albeit a newer version of the code in the paper does.

! TODO !

To cite: Danielsson's operational semantics - Leroy's coinductive big step operational semantics - Abel (various) - Hutton's Calculating dependently-typed compilers (functional pearl)

APPENDIX A

Proofs

In this appendix we will show the Agda code for all the theorems mentioned in the thesis.

A.1 The delay monad

Theorem 3.3.1

Proof:

```
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)
```

□

Theorem 3.3.2

Proof:

```
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})
```

□

A.2 The Imp programming language

Theorem 4.1.2.1

Proof:

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

see code

□

Theorem 4.2.4.1

Proof:

```
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

□

Theorem 4.3.1.1.1

Proof:

```
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

□

Lemma 4.3.1.2.4

Proof:

```
ceval⇓⇒scçs' :  ∀ (c : Command) (s s' : Store) (h⇓ : (ceval c s) ⇓ s') →
(dom s ∪ (cvars c)) ⊆ (dom s')
ceval⇓⇒scçs' skip s .s (now refl) x x-in-s₁ rewrite (cvars-skip) rewrite (v-
identityʳ (dom s x)) = x-in-s₁
ceval⇓⇒scçs' (assign id a) s s' h⇓ x x-in-s₁ with (aeval a s)
... | nothing with h⇓
... | now ()
ceval⇓⇒scçs' (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⇓⇒scçs' (ifelse b cᵗ cᶠ) s s' h⇓ x x-in-s₁ with (beval b s) in eq-b
... | nothing with h⇓
... | now ()
ceval⇓⇒scçs' (ifelse b cᵗ cᶠ) s s' h⇓ x x-in-s₁ | just false rewrite eq-b
 = ceval⇓⇒scçs' cᶠ s s' h⇓ x (h {dom s x} {cvars cᵗ x} {cvars cᶠ x} x-in-s₁)
ceval⇓⇒scçs' (ifelse b cᵗ cᶠ) s s' h⇓ x x-in-s₁ | just true rewrite eq-b
 = ceval⇓⇒scçs' cᵗ s s' h⇓ x (h {dom s x} {cvars cᵗ x} {cvars cᶠ x}  x-in-
s₁)
ceval⇓⇒scçs' (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⇓⇒scçs' c₁ s sⁱ c₁⇓sⁱ x)
... | n with (ceval⇓⇒scçs' 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⇓⇒scçs' (while b c) s s' h⇓ x x-in-s₁ rewrite (cvars-while {b} {c})
 rewrite (v-identityʳ (dom s x)) = ceval⇓⇒ç (while b c) s s' h⇓ x x-in-s₁
```

see code

□

Theorem 4.3.1.2.1

Proof:

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

□

Theorem 4.3.1.2.2

Proof:

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

□

Theorem 4.3.1.2.3

Proof:

```
dia-sound : ∀ (c : Command) (s : Store) (v v' : VarsSet) (dia : Dia v c v')
(v⊆s : v ⊆ dom s) → (h-err : (ceval c s) ↯) → ⊥
dia-sound skip s v v' dia v⊆s (now ())
dia-sound (assign id a) s v .(id ↦ v) (assign .a .v .id a⊆v) v⊆s h-err with
(adia-sound a s (⊆-trans a⊆v v⊆s)) ... | a' , eq-aeval with h-err
... | now ()
dia-sound (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-sound 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-sound cᶠ s v
vᶠ diaᶠ v⊆s h-err
dia-sound (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-
sound cᵗ s v vᵗ diaᵗ v⊆s h-err
dia-sound (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-sound c₁ s v₁ v₂ dia-c₁ v⊆s (≡⇒≋ eq-ceval-c₁)
... | now (just s') rewrite eq-ceval-c₁ = dia-sound c₂ s' v₂ v₃ dia-c₂ (dia-
ceval⇒⊆ dia-c₁ v⊆s (≡⇒≋ eq-ceval-c₁)) h-err
dia-sound (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-sound c₁ s v₁ v₂ dia-c₁ v⊆s)
... | c₁↯⊥ rewrite eq-ceval-c₁ = dia-sound-seq-later c₁↯⊥ dia-c₂ h h-err
dia-sound (while b c) s v v' dia v⊆s h-err with dia
... | while .b .v v₁ .c b⊆s dia-c with (bdia-sound 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-sound (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-sound c s v v₁ dia-c v⊆s (≡⇒≋ eq-ceval-c)
dia-sound (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-sound (while b c) s' v v dia (⊆-trans v⊆s (ceval⇓⇒⊆ c
s s' (≡⇒≋ eq-ceval-c))) w↯
dia-sound (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-sound c s v v₁ dia-c v⊆s)
... | c↯⊥ rewrite eq-beval rewrite eq-ceval-c = dia-sound-while-later c↯⊥ dia
h h-err
```

see code

□

Theorem 4.3.2.1.1

Proof:

```
-- Pure constant folding preserves semantics.
apfold-sound : ∀ a s → (aeval a s ≡ aeval (apfold a) s)
apfold-sound (const n) _ = refl
apfold-sound (var id) _ = refl
apfold-sound (plus a₁ a₂) s
 rewrite (apfold-sound a₁ s)
 rewrite (apfold-sound 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

□

Theorem 4.3.2.2.1

Proof:

```
bpfold-sound : ∀ b s  → (beval b s ≡ beval (bpfold b) s)
bpfold-sound (const b) s  = refl
bpfold-sound (le a₁ a₂) s rewrite (apfold-sound a₁ s) rewrite (apfold-sound
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-sound (not b) s rewrite (bpfold-sound b s) with (bpfold b)
... | const b₁ = refl
... | le a₁ a₂ = refl
... | not v = refl
... | and v v₁ = refl
bpfold-sound (and b₁ b₂) s rewrite (bpfold-sound b₁ s) rewrite (bpfold-sound
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

□

Theorem 4.3.2.3.1

Proof:

```
cpfold-sound : ∀ (c : Command) (s : Store) → ∞ ⊢ (ceval c s) ≈ (ceval (cpfold
c) s)
cpfold-sound skip s rewrite (cpfold-skip) = now refl
cpfold-sound (assign id a) s = ≡⇒≈ (cpfold-assign a id s)
cpfold-sound (ifelse b cᵗ cᶠ) s = cpfold-if b cᵗ cᶠ s
cpfold-sound (seq c₁ c₂) s = cpfold-seq c₁ c₂ s
cpfold-sound (while b c) s = cpfold-while b c s
-- way to long..
```

see code

□

Bibliography

Abel, Andreas. “Miniagda: Integrating Sized and Dependent Types.” doi:10.48550/ARXIV.1012.4896.

Abel, Andreas, and Chapman, James. “Normalization by Evaluation in the Delay Monad: A Case Study for

Coinduction Via Copatterns and Sized Types.” *Electronic Proceedings in Theoretical Computer Science*, vol. 153, doi:10.4204/eptcs.153.4.

Capretta, Venanzio. “General Recursion Via Coinductive Types.” *Logical Methods in Computer Science*, doi:10.2168/lmcs-1(2:1)2005.

Danielsson, Nils Anders. “Operational Semantics Using the Partiality Monad.” *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ACM, doi:10.1145/2364527.2364546.

Kohl, Christina, and Schwaiger, Christina. “Monads in Computer Science.” https://ncatlab.org/nlab/files/KohlSchwaiger-Monads.pdf.

Leroy, Xavier, and Grall, Hervé. “Coinductive Big-Step Operational Semantics.” doi:10.48550/ARXIV.0808.0586.

Moggi, E. “Computational Lambda-Calculus and Monads.” *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer*

Science, vol. 0, doi:10.1109/LICS.1989.39155.

Nipkow, Tobias, and Klein, Gerwin. *Concrete Semantics: With Isabelle/hol.* Springer Publishing Company, 2014.

Pierce, Benjamin C., et al. *Logical Foundations.* 2023.