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: 18)`

"...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

CHAPTER 2

Induction and coinduction

2.1 Infinite datatypes

2.2 Infinite proofs

2.3 Relation with fixed points

CHAPTER 3

Agda

In this chapter we will introduce the Agda programming language.

1. the shortest history of proof assistants ever
2. what makes agda useful, i.e., dependent types

3.1 Dependent types

3.2 Termination and productivity

3.3 Sized types

CHAPTER 4

The partiality monad

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

4.1 Monads

In 1989, computer scientist 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
```

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
```

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's give an example:

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

Without `bind`, `h` would be a match on the value of `x`: if `x` is `just v` then do something, otherwise, if `x` is `nothing`, return `nothing`. 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.

4.1.1 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 4.1.1.1 (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)`

4.2 The Delay monad

In 2005, computer scientist 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 (partiality) an effect, taking the perspective of

In Agda, the `Delay` type is defined as follows (using *sizes*, see Chapter 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
```

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
```

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 Listing 1: of course, `bind' never f` would never terminate.

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

Listing 1: Non-terminating term in the `Delay` monad

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.

4.3 Bisimilarity

An important notion relating terms of `Delay` type is that of *bisimilarity*. **Strong** bisimilarity relates diverging computations and computations that converge to the same value using the same number of steps (Danielsson 2012); the formal definition we give is from (Chapman, Uustalu, and Veltri 2015), and is shown in Definition 4.3.1 properties of this relation are given in Theorem 4.3.1.

A less strict relation is **weak** bisimilarity: it equally relates diverging terms (coinductively), but it also allows the relation between computations that converge to the same value but in different number of steps.

Definition 4.3.1 (Strong bisimilarity): Let A be a type for which there exists an equivalence relation R_A, and let a_1 and a_2 be two terms of type A. Furthermore, let x_1 and x_1 be two delayed terms. Then, we define the strong bisimilarity relation $\sim_A$ as

$$\frac{a_1 \mathrel{R} a_2}{\approx\textit{-now}\colon \text{now}\ a_1 \sim_A \text{now}\ a_2} \qquad \frac{x_1 \sim_A x_2}{\approx\textit{-later}\colon \text{later}\ x_1 \sim_A \text{later}\ x_2}$$

In Agda:

```
data Bisim {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 → Bisim R i (now x) (now y)
  later : ∀ {xs ys} → Thunk^R (Bisim R) i xs ys → Bisim R i (later xs) (later ys)
```

Theorem 4.3.1 (Strong bisimilarity is an equivalence relation): For every equivalence relation A, the strong bisimilarity relation $\sim_A$ is an equivalence relation. Furthermore, strong bisimilarity is a transitive relation. In Agda:

```
reflexive : Reflexive R → ∀ {i} → Reflexive (Bisim R i)
symmetric : Sym P Q → ∀ {i} → Sym (Bisim P i) (Bisim Q i)
transitive : Trans P Q R → ∀ {i} → Trans (Bisim P i) (Bisim Q i) (Bisim R i)
```

Definition 4.3.2 (Weak bisimilarity): Let A be a type for which there exists an equivalence relation R_A, and let a_1 and a_2 be two terms of type A. Furthermore, let x_1 and x_1 be two delayed terms. Then, we define the weak bisimilarity relation $\sim_A$ as

$$\frac{a_1 \mathrel{R} a_2}{\approx\textit{-now}\colon \text{now}\ a_1 \sim_A \text{now}\ a_2} \qquad \frac{\text{force}\ x_1 \sim_A \text{force}\ x_2}{\approx\textit{-later}\colon \text{later}\ x_1 \sim_A \text{later}\ x_2}$$

$$\frac{\text{force}\ x_1 \sim_a x_2}{\approx\textit{-later-l}\colon \text{later}\ x_1 \sim_a x_2} \qquad \frac{x_1 \sim_a \text{force}\ x_2}{\approx\textit{-later-r}\colon x_1 \sim_a \text{later}\ x_2}$$

In Agda:

```
data Bisim {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 → Bisim R i x y
  later : ∀ {xs ys} → Thunk^R (Bisim R) i xs ys → Bisim R i (later xs) (later
ys)
  laterₗ : ∀ {xs ys} → Bisim R i (force xs) ys → Bisim R i (later xs) ys
  laterᵣ : ∀ {xs ys} → Bisim R i xs (force ys) → Bisim R i xs (later ys)
```

Theorem 4.3.2 (Weak bisimilarity is an equivalence relation): For every equivalence relation A, the weak bisimilarity relation $\sim_A$ is an equivalence relation. In Agda:

```
reflexive : Reflexive R → ∀ {i} → Reflexive (WeakBisim R i)
symmetric : Sym P Q → ∀ {i} → Sym (Bisim P i) (WeakBisim Q i)
```

It's also trivial to show that strong bisimilarity implies weak bisimilarity. We can also now prove monad laws up to strong bisimilarity, as shown in Theorem 4.3.3.

Theorem 4.3.3 (`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) → bind (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)
```

CHAPTER 5

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.

5.1 Introduction

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

5.1.1 Syntax

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

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

Table 1: Syntax rules for the Imp language

The syntactic elements of this language are three: *commands*, *arithmetic expressions* and *boolean expressions*. Given its simple nature, it's easy to give an abstract representation for its concrete syntax: all three can be represented with simple datatypes enclosing all the information of the syntactic rule.

Another important atomic element of Imp are *identifiers*. Identifiers can mutate in time and, when misused, cause errors during the execution of programs: in fact, there is no way to enforce the programmer to use only initialized identifiers merely by syntax rules – it would take a context-sensitive grammar to achieve so, at least. A concept related to identifiers is that of *stores*, which are conceptually instantaneous descriptions of the state of identifiers in the ideal machine executing the program.

We now show how we implemented the syntactic elements of Imp in Agda and show a handful of trivial properties: in Listing 2 we show the datatypes for identifiers and stores, while in Listing 3 we show the datatypes for the other syntactic constructs.

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

Listing 2: Datatypes for identifiers and stores

Notice that the implementation of `Stores` reflect the behaviour described earlier in that they are intended as functions from `Ident` to `Maybe ℤ`.

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

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

```
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
```

Listing 3: Datatype for expressions of Imp

5.1.2 Properties of stores

The first properties we show regard stores. We equip stores with the trivial operations of adding an identifier, merging two stores and joining two stores, as shown in Listing 2.

```
empty : Store
empty = λ _ → nothing
update : (id₁ : Ident) → (v : ℤ) → (s : Store) → Store
update id₁ v s id₂
 with id₁ == id₂
... | true = (just v)
... | false = (s id₂)
join : (s₁ s₂ : Store) → Store
join s₁ s₂ id
 with (s₁ id)
... | just v = just v
... | nothing = s₂ id
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
```

Listing 2: Operations on stores

A trivial property of stores is that of unvalued inclusion, that is, a property stating that if an identifier has a value in a store σ_1, then it also has a value (not necessarily the same) in another store σ_2:

Property 5.1.2.1 (Unvalued store inclusion): Let σ_1 and σ_2 be two stores. We define the unvalued inclusion between them as

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

and we denote it with $\sigma_1 \overset{u}{\sqsubseteq} \sigma_2$. In Agda:

We equip Property 5.1.2.1 with a notion of transitivity.

Theorem 5.1.2.1 (Transitivity of unvalued store inclusion): Let σ_1, σ_2 and σ_3 be three stores. Then

$$\sigma_1 \overset{u}{\sqsubseteq} \sigma_2 \wedge \sigma_2 \overset{u}{\sqsubseteq} \sigma_3 \to \sigma_1 \overset{u}{\sqsubseteq} \sigma_3 \tag{2}$$

In Agda:

The operations we define on stores are multiple: adding an identifier paired with a value to a store, removing an identifier from a store, joining stores and merging stores. We now define notations:

1. **in-store predicate** let id : Ident and σ : Store. To say that id is in σ we write id $\in \sigma$; in other terms, it's the same as $\exists\, v \in \mathbb{Z}, \sigma \text{ id} \equiv \text{just } v$.
2. **empty store** we define the empty store as $\emptyset$. For this special store, it is always $\forall \text{ id}, \text{id} \in \emptyset \to \perp$ or $\forall \text{ id}, \emptyset \text{ id} \equiv \text{nothing}$.
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 $(\text{id}, v) \mapsto \sigma$.
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
$$\exists \text{ id}, \exists\, v_1, \exists\, v_2, v_1 \neq v_2 \wedge \sigma_1 \text{ id} \equiv \text{just } v_1 \wedge \sigma_2 \text{ id} \equiv \text{just } v_2 \tag{3}$$
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 $\sigma_1 \cap \sigma_2$.

5.1.3 Properties of expressions

The properties of expressions we show here regard the syntactic relation between elements. The property we define is that of *subterm relation*. In Agda, as will be shown in the definitions, these properties are implemented as datatypes. Properties 5.1.3.3, 5.1.3.2 and 5.1.3.1 will be used later to relate semantic aspects of subterms with that of the containing term itself or vice versa.

Property 5.1.3.1 (Arithmetic subterms): Let a_1 and a_2 be arithmetic expressions.

Then

$$a_1 \overset{a}{\sqsubset} \text{plus } a_1\, a_2 \qquad a_2 \overset{a}{\sqsubset} \text{plus } a_1\, a_2$$

In Agda:

Property 5.1.3.2 (Boolean subterms): Let a_1 and a_2 be arithmetic expressions and b_1 and b_2 be boolean expressions.

Then

$$a_1 \overset{b}{\sqsubset} \text{le } a_1\, a_2 \qquad a_2 \overset{b}{\sqsubset} \text{le } a_1\, a_2$$
$$b_1 \overset{b}{\sqsubset} \text{and } b_1\, b_2 \qquad b_2 \overset{b}{\sqsubset} \text{and } b_1\, b_2$$
$$b_1 \overset{b}{\sqsubset} \text{not } b_1$$

In Agda:

```
data
  _⊑ᵇ_ : {A : Set} → A → BExp → Set where
    not : (b : BExp) → b ⊑ᵇ (not b)
    and-l : (b₁ b₂ : BExp) → b₁ ⊑ᵇ (and b₁ b₂)
    and-r : (b₁ b₂ : BExp) → b₂ ⊑ᵇ (and b₁ b₂)
    le-l : (a₁ a₂ : AExp) → a₁ ⊑ᵇ (le a₁ a₂)
    le-r : (a₁ a₂ : AExp) → a₂ ⊑ᵇ (le a₁ a₂)
```

In Agda:

Property 5.1.3.3 (Command subterms): Let id be an identifier, a be an arithmetic expressions, b be a boolean expression and c_1 and c_2 be commands.
Then

$$a \overset{c}{\sqsubset} \text{assign id } a \qquad c_1 \overset{c}{\sqsubset} \text{seq } c_1\, c_2 \qquad c_2 \overset{c}{\sqsubset} \text{seq } c_1\, c_2 \qquad b \overset{c}{\sqsubset} \text{if } b\, c_1\, c_2$$

$$c_1 \overset{c}{\sqsubset} \text{if } b\, c_1\, c_2 \qquad c_2 \overset{c}{\sqsubset} \text{if } b\, c_1\, c_2 \qquad b \overset{c}{\sqsubset} \text{while } b\, c_1 \qquad c_1 \overset{c}{\sqsubset} \text{while } b\, c_1$$

In Agda:

5.2 Semantics

Having understood the concrete and abstract 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 1, Imp programs are composed of three entities: arithmetic expression, boolean expression and commands.

```
true then skip else 1
```

Listing 3: A simple Imp program

An example of Imp program is shown in Listing 3: note that is technically not well-typed, but we don't care about this now. In general, we can expect the evaluation of an Imp program to terminate in some kind value or diverge, but it might also **fail**: this is the case when an uninitialized variable is used, as we mentioned in Chapter 5.1.1.

We could model other kinds of failures, 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: an example of this can be seen in Listing 4.

```
while true do x := y
```

Listing 4: A failing (not diverging!) Imp program

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 Listing 5.

```
while true do skip
```

Listing 5: A diverging Imp program

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.

5.2.1 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, conceptually, 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 and, as we did for stores, we can target the Maybe monad.

Without introducing them, we will use notations similar to that used earlier for commands ($\cdot \Downarrow \cdot$ and $\cdot$ ↯)

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

$$\frac{\text{id} \notin \sigma}{\text{var id}\ ↯} \qquad \frac{a_1\ ↯}{\text{plus}\ a_1 a_2\ ↯} \qquad \frac{a_1 \Downarrow \text{just}\ n_1 \quad a_2\ ↯}{\text{plus}\ a_1 a_2\ ↯}$$

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

In Agda, these rules are implemented as shown in Listing 6.

```
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)
```

Listing 6: Agda interpreter for arithmetic expressions

5.2.2 Boolean expressions

Boolean expressions in Imp can be of four kinds: boolean constants, negation of a boolean expression, logical $\wedge$ and, finally, comparison of arithmetic expressions. The line of reasoning for the definition of semantic rules follows what we underlined earlier, that is, we again target the Maybe monad.

$\dfrac{}{\text{const c} \Downarrow \text{just c}}$	$\dfrac{b \Downarrow c}{\neg b \Downarrow \neg c}$	$\dfrac{a_1 \Downarrow \text{just } n_1 \quad a_2 \Downarrow \text{just } n_2}{\text{le } a_1 a_2 \Downarrow \text{just } (n_1 < n_2)}$
$\dfrac{b_1 \Downarrow \text{just } c_1 \quad b_2 \Downarrow \text{just } c_2}{\text{and } b_1 b_2 \Downarrow \text{just } (c_1 \wedge c_2)}$	$\dfrac{b \lightning}{\neg b \lightning}$	$\dfrac{a_1 \lightning}{\text{le } a_1 a_2 \lightning}$
$\dfrac{a_1 \Downarrow \text{just } n_1 \quad a_2 \lightning}{\text{le } a_1 a_2 \lightning}$	$\dfrac{b_1 \lightning}{\text{and } b_1 b_2 \lightning}$	$\dfrac{b_1 \Downarrow \text{just } c_1 \quad b_2 \lightning}{\text{and } b_1 b_2 \lightning}$

Table 5: Inference rules for the semantics of boolean expressions of Imp

In Agda, these rules are implemented as shown in Listing 7.

```
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)
```

Listing 7: Agda interpreter for boolean expressions

5.2.3 Commands

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

In Agda, these rules are implemented as shown in Listing 8.

5.2.4 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.

```
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 ≫=m λ v → just (update id v s))
 ceval (seq c c₁) s = ceval c s ≫=p λ s' → ceval c₁ s'
 ceval (ifelse b c c₁) s = now (beval b s) ≫=p
  (λ bᵥ → (if bᵥ then ceval c s else ceval c₁ s))
 ceval (while b c) s = now (beval b s) ≫=p
  (λ bᵥ →
   if bᵥ then (ceval c s ≫=p  λ s → later (ceval-while c b s))
   else now (just s))
```

Listing 8: Agda interpreter for commands

Theorem 5.2.4.1 (`ceval` does not remove identifiers): Let c be a command and σ_1 and σ_2 be two stores. Then

$$\text{ceval}\ c\ \sigma_1 \Downarrow \sigma_2 \to \sigma_1 \overset{u}{\sqsubseteq} \sigma_2 \tag{4}$$

In Agda:

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

Theorem 5.2.4.1 will be fundamental for later proofs.

5.3 Analyses and optimizations

We chose to demonstrate the use of coinduction in the definition of operational semantics implementing operations on the code itself (that is, they're static analyses), then showing proofs regarding the result of the execution of the program. The main inspiration for these operations is (Nipkow and Klein 2014).

5.3.1 Definite initialization analysis

The first operationn 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 indicator functions, which we trivially define as parametric functions from a parametric set to the set of booleans, that is `IndicatorFunction = 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 indicator functions of the usual operations on sets: insertion, union, and intersection and define the usual property of inclusion.

```
∅ : IndicatorFunction
∅ = λ _ → false

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

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

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

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

Listing 9: Implementation of indicator functions in Agda

Important properties of `IndicatorFunctions` (and thus of `VarsSets`) follows.

Theorem 5.3.1.1.1 (Equivalence of indicator functions):
(using the **Axiom of extensionality**)

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

Theorem 5.3.1.1.2 (Neutral element of union):

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

Theorem 5.3.1.1.3 (Update inclusion):

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

Theorem 5.3.1.1.4 (Transitivity of inclusion):

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

We will also need a way to get a `VarsSet` from a `Store`, which is shown in Listing 10.

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

Listing 10: Code to compute the domain of a `Store` in Agda

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 Listing 6; we also need a function to compute the set of variables that are definitely initialized in commands, which is shown in Listing 11.

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

```
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)
```

Listing 6: Agda code to compute variables in arithmetic and boolean expressions

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

Listing 11: Agda code to compute initialized variables in commands

Theorem 5.2.4.1 allows us to show the following theorem.

Theorem 5.3.1.2.1 (`ceval` adds at least the variables in commands):
Let c be a command and σ_1 and σ_2 be two stores. Then

$$\text{ceval}\ c\ \sigma_1 \Downarrow \sigma_2 \longrightarrow (\text{dom}\ \sigma_1 \cup (\text{cvars}\ c)) \subseteq (\text{dom}\ \sigma_2) \tag{5}$$

In Agda:

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

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 Listing 12.

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

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

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

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
```

Listing 12: Dia relation in Agda

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 5.3.1.2.2 and Theorem 5.3.1.2.3 show that if the variables in an arithmetic expression or a boolean expression are contained in a store the result of their evaluation can't be a failure (i.e. they result in "just" something), Theorem 5.3.1.2.4 shows that if `Dia` holds, then the evaluation of a program failing is absurd.

Theorem 5.3.1.2.2 (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)
```

Theorem 5.3.1.2.3 (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)
```

Theorem 5.3.1.2.4 (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) ↯) → ⊥
```

Here, we show the proof of Theorem 5.3.1.2.4:

Proof:

```
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)
  ... | ()
  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
   where
    h : ∀ {s'} (h : (later x) ⇓ s') → v₂ ⊆ dom s'
    h h₁ rewrite (sym eq-ceval-c₁) = dia-ceval⇒⊆ dia-c₁ v⊆s h₁
  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 = case h-err of λ ()
  ... | 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
   where
    h : ∀ {s'} (h : (later x) ⇓ s') → v ⊆ dom s'
    h {s'} h₁ rewrite (sym eq-ceval-c) = (⊆-trans v⊆s (ceval⇓⇒⊆ c s s' h₁))
```

□

5.3.2 Pure constant folding optimization

Pure constant folding is the second and last operation we considered. Again from (Nipkow and Klein 2014), the operation of 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 don't 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 ($a \equiv \text{plus}\ \ a_1\ a_2$), 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 $a_1\ a_2$ left untouched. The Agda code for the function `apfold` is shown in Listing 13.

```
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'
```

Listing 13: Agda code for pure folding of arithmetic expressions

Of course, what we want to show is that this optimization does not change the result of the evaluation (Theorem 5.3.2.1.1).

Theorem 5.3.2.1.1 (Soundness of pure folding for arithmetic expressions): Let a be an arithmetic expression and s be a store. Then

$$\text{aeval}\ a\ s \equiv \text{aeval}\ (\text{apfold}\ a)\ s \tag{6}$$

In Agda: `apfold-sound : ∀ a s → (aeval a s ≡ aeval (apfold a) s)`

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 5.3.2.1: 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 Listing 14.

```
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
```

Listing 14: Agda code for pure folding of arithmetic expressions

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 5.3.2.2.1 (Soundness of pure folding for boolean expressions): Let b be a boolean expression and s be a store. Then

$$\text{beval}\ b\ s \equiv \text{beval}\ (\text{bpfold}\ b)\ s \tag{7}$$

In Agda:

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

Pure folding of commands

Pure folding of commands builds on the definition of apfold and bpfold above combining the definitions as shown in Listing 15.

```
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)
```

Listing 15: Agda code for pure folding of commands

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 5.3.2.3.1).

Theorem 5.3.2.3.1 (Soundness of pure folding for commands): Let c be a command and s be a store. Then

$$\text{ceval}\; c\; s \equiv \text{ceval}\left(\text{cpfold}\; b\right) s \tag{8}$$

In Agda:

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

Of course, what makes Theorem 5.3.2.3.1 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.

APPENDIX A

Proofs

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

A.1 The partiality monad

A.1.1 Bisimilarity

Theorem 4.3.1

Proof:

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

symmetric : Sym P Q → ∀ {i} → Sym (Bisim P i) (Bisim Q i)
symmetric sym^PQ (now p)     = now (sym^PQ p)
symmetric sym^PQ (later ps) = later λ where .force → symmetric sym^PQ (ps .force)

transitive : Trans P Q R → ∀ {i} → Trans (Bisim P i) (Bisim Q i) (Bisim R i)
transitive trans^PQR (now p)    (now q)     = now (trans^PQR p q)
transitive trans^PQR (later ps) (later qs) =
  later λ where .force → transitive trans^PQR (ps .force) (qs .force)
```

□

Theorem 4.3.2

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 4.3.3

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

A.2.1 Properties of stores

Theorem 5.1.2.1

Proof:

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

□

A.2.2 Semantics

Theorem 5.2.4.1

Proof:

```
mutual
 private
  while-⊑ᵘ-later :  ∀ {x : Thunk (Delay (Maybe Store)) ∞} (c : Command) (b : BExp)
(s s' : Store)
   (f : ∀ (sⁱ : Store) → later x ⇓ sⁱ → s ⊑ᵘ sⁱ) (h⇓ : ((later x) >>=ᵖ (λ s →
later (ceval-while c b s))) ⇓ s')
    → s ⊑ᵘ s'
  while-⊑ᵘ-later {x} c b s s' f (later₁ h⇓) {id} (z , id∈s)
   with (force x) in eq-fx
```

```
  ... | now (just sⁱ) rewrite eq-fx
   with h⇓
  ... | later₁ w⇓
   with (beval b sⁱ) in eq-b
  ... | just false with w⇓
  ... | nowj refl = f sⁱ (later₁ (≡⇒≈ eq-fx)) (z , id∈s)
  while-⊑ᵘ-later {x} c b s s' f (later₁ h⇓) {id} (z , id∈s) | now (just sⁱ) |
later₁ w⇓
   | just true rewrite eq-b
   with (bindxf⇓⇒x⇓ {x = ceval c sⁱ} {f = λ s → later (ceval-while c b s)} w⇓)
  ... | sⁱ' , cⁱ⇓sⁱ'
   with (f sⁱ (later₁ (≡⇒≈ eq-fx)) {id})
  ... | s⊑sⁱ
   with (while-⊑ᵘ c b sⁱ s' (λ { s₁ sⁱ₁ c⇓sⁱ {id} (z' , id∈s₁) → ceval⇓⇒⊑ᵘ c s₁
sⁱ₁ c⇓sⁱ {id} (z' , id∈s₁)}) w⇓ {id})
  ... | sⁱ⊑s' = ⊑ᵘ-trans s⊑sⁱ sⁱ⊑s' {id} (z , id∈s)
  while-⊑ᵘ-later {x} c b s s' f (later₁ h⇓) {id} (z , id∈s)
   | later x₁ = while-⊑ᵘ-later {x₁} c b s s' (λ { sⁱ x₂ x₃ → f sⁱ (later₁ (≡⇒⇓
eq-fx x₂)) x₃}) h⇓ {id} (z , id∈s)


  while-⊑ᵘ : ∀ (c : Command) (b : BExp) (s s' : Store) (f : ∀ (s sⁱ : Store) →
(ceval c s) ⇓ sⁱ → s ⊑ᵘ sⁱ)
   (h⇓ : ((ceval c s) >>=ᵖ (λ s → later (ceval-while c b s))) ⇓ s') → s ⊑ᵘ s'
  while-⊑ᵘ c b s s' f h⇓ {id}
   with (ceval c s) in eq-c
  ... | now (just sⁱ)
   with (f s sⁱ (≡⇒≈ eq-c))
  ... | s⊑sⁱ rewrite eq-c
   with h⇓
  ... | later₁ w⇓
   with (beval b sⁱ) in eq-b
  ... | just false rewrite eq-b with w⇓
  ... | nowj refl = s⊑sⁱ {id}
  while-⊑ᵘ c b s s' f h⇓ {id} | now (just sⁱ) | s⊑sⁱ | later₁ w⇓ | just true
   rewrite eq-b
   = ⊑ᵘ-trans {s} {sⁱ} {s'} s⊑sⁱ (while-⊑ᵘ c b sⁱ s' f w⇓) {id}
  while-⊑ᵘ c b s s' f h⇓
   | later x
   with h⇓
  ... | later₁ w⇓ = while-⊑ᵘ-later {x} c b s s' (λ { sⁱ x₁ x₂ → f s sⁱ (≡⇒⇓ eq-
c x₁) x₂}) h⇓

 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
```

□

Bibliography

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.

Chapman, James, et al. "Quotienting the Delay Monad by Weak Bisimilarity." *Theoretical Aspects of Computing - ICTAC 2015*, Springer International Publishing, doi: 10.1007/978-3-319-25150-9_8.

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.

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.