Operational Semantics Using the Partiality Monad

Nils Anders Danielsson
Chalmers University of Technology and University of Gothenburg
nad@chalmers.se

Abstract

The operational semantics of a partial, functional language is often given as a relation rather than as a function. The latter approach is arguably more natural: if the language is functional, why not take advantage of this when defining the semantics? One can immediately see that a functional semantics is deterministic and, in a constructive setting, computable.

This paper shows how one can use the coinductive partiality monad to define big-step or small-step operational semantics for lambda-calculi and virtual machines as total, computable functions (total definitional interpreters). To demonstrate that the resulting semantics are useful type soundness and compiler correctness results are also proved. The results have been implemented and checked using Agda, a dependently typed programming language and proof assistant.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; E.1 [*Data Structures*]; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords Dependent types; mixed induction and coinduction; partiality monad

1. Introduction

Consider the untyped λ-calculus with a countably infinite set of constants c:

$$t ::= c \mid x \mid \lambda x.t \mid t_1\ t_2$$

Closed terms written in this language can compute to a value (a constant c or a closure $\lambda x.t\rho$), but they can also go wrong (crash) or fail to terminate.

How would you write down an operational semantics for this language? A common choice is to define the semantics as an inductively defined relation, either using small steps or big steps. For an example of the latter, see Figure 1: $\rho \vdash t \Downarrow v$ means that the term t can terminate with the value v when evaluated in the environment ρ. However, as noted by Leroy and Grall (2009), this definition provides no way to distinguish terms which go wrong from terms which fail to terminate. If we want to do this, then we can define two more relations, see Figure 2: $\rho \vdash t \Uparrow$, defined *coinductively*, means that the term t can fail to terminate when evaluated in the environment ρ; and $\rho \vdash t \lightning$ means that t goes wrong.

$$\rho \vdash c \Downarrow c \qquad \frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \qquad \rho \vdash \lambda x.t \Downarrow \lambda x.t\rho$$

$$\frac{\rho \vdash t_1 \Downarrow \lambda x.t'\rho' \qquad \rho \vdash t_2 \Downarrow v' \qquad \rho', x\ =\ v' \vdash t' \Downarrow v}{\rho \vdash t_1\ t_2 \Downarrow v}$$

Figure 1. A call-by-value operational semantics for the untyped λ-calculus with constants, specifying which terms can terminate with what values (very close to a semantics given by Leroy and Grall (2009)).

$$\frac{\rho \vdash t_1 \Uparrow}{\rho \vdash t_1\ t_2 \Uparrow} \qquad \frac{\rho \vdash t_1 \Downarrow v \qquad \rho \vdash t_2 \Uparrow}{\rho \vdash t_1\ t_2 \Uparrow}$$

$$\frac{\rho \vdash t_1 \Downarrow \lambda x.t'\rho' \qquad \rho \vdash t_2 \Downarrow v' \qquad \rho', x\ =\ v' \vdash t' \Uparrow}{\rho \vdash t_1\ t_2 \Uparrow}$$

$$\rho \vdash t \lightning \quad \stackrel{\text{def}}{=} \quad \neg\ (\exists v.\ \rho \vdash t \Downarrow v)\ \wedge\ \neg\ (\rho \vdash t \Uparrow)$$

Figure 2. Two more operational semantics for the untyped λ-calculus with constants, specifying which terms can fail to terminate or go wrong. The definition written using double lines is coinductive, and is taken almost verbatim from Leroy and Grall (2009).

Now we have a complete definition. However, this definition is somewhat problematic:

1. There are four separate rules which refer to application. For a small language this may be acceptable, but for large languages it seems to be easy to forget some rule, and "rule duplication" can be error-prone.
2. It is not immediately obvious whether the semantics is deterministic and/or computable: these properties need to be proved.
3. If we want to define an interpreter which is correct by construction, then the setup with three relations is awkward. Consider the following type-signature, where $\_\uplus\_$ is the sum type constructor:

   $$\mathit{eval}\ :\ \forall\ \rho\ t \to (\exists v.\ \rho \vdash t \Downarrow v)\ \uplus\ \rho \vdash t \Uparrow\ \uplus\ \rho \vdash t \lightning$$

   This signature states that, for any environment ρ and term t, the interpreter either returns a value v and a proof that t can terminate with this value when evaluated in the given environment; or a proof that t can fail to terminate; or a proof that t goes wrong. It should be clear that it is impossible to implement *eval*

in a total, constructive language, as this amounts to solving the halting problem.

The situation may have been a bit less problematic if we had defined a small-step semantics instead, but small-step semantics are not necessarily better: Leroy and Grall (2009) claim that "big-step semantics is more convenient than small-step semantics for some applications", including proving that a compiler is correct.

I suggest another approach: define the semantics as a *function* in a *total* meta-language, using the *partiality monad* (Capretta 2005) to represent non-termination, where the partiality monad is defined coinductively as $A_\bot = \nu X.\, A \uplus X$. If this approach is followed then we avoid all the problems above:

1. We have one clause for applications, and the meta-language is total, so we cannot forget a clause.
2. The semantics is a total function, and hence deterministic and computable.
3. The semantics is an interpreter, and its type signature does not imply that we solve the halting problem:

$$[\![\_]\!] : \mathit{Term} \to \mathit{Environment} \to (\mathit{Maybe}\ \mathit{Value})_\bot$$

An additional advantage of using a definitional interpreter is that this can make it easy to test the semantics (if the interpreter is not too inefficient). Such tests can be useful in the design of non-trivial languages (Aydemir et al. 2005).

The main technical contribution of this paper is that I show that one can prove typical meta-theoretical properties directly for a semantics defined using the partiality monad:

- A big-step, functional semantics is defined and proved to be classically equivalent to the relational semantics above (Sections 3 and 5; for simplicity well-scoped de Bruijn indices are used instead of names).
- Type soundness is proved for a simple type system with recursive types (Section 4).
- The meaning of a virtual machine is defined as a small-step, functional semantics (Section 6).
- A compiler correctness result is proved (Section 7).
- The language and the type soundness and compiler correctness results are extended to a non-deterministic setting in order to illustrate that the approach can handle languages where some details—like evaluation order—are left up to the compiler writer (Section 8).
- Finally Section 9 contains a brief discussion of term equivalences (applicative bisimilarity and contextual equivalence).

As far as I know these are the first proofs of type soundness or compiler correctness for operational semantics defined using the partiality monad. The big-step semantics avoids the rule duplication mentioned above, and this is reflected in the proofs: there is only one case for application, as opposed to four cases in some corresponding proofs for relational semantics due to Leroy and Grall (2009). Related work is discussed further in Section 1.3.

1.1 Operational?

At this point some readers may complain that $[\![\_]\!]$ does not define an operational semantics, but rather a denotational one. Perhaps a better term would be "hybrid operational/denotational", but the semantics is *not* denotational:

- It is not defined in a compositional way: $[\![\ t\ ]\!]$ is not defined by recursion on the structure of t, but rather a combination of corecursion and structural recursion (see Section 3).
- Furthermore the "semantic domain" is rather syntactic: it includes closures, and is not defined as the solution to a domain equation.

I do not see this kind of semantics as an alternative to denotational semantics, but rather as an alternative to usual operational ones. (See also the discussion of term equivalences in Section 9.)

1.2 Mechanisation

The development presented below has been formalised in the dependently typed, functional language Agda (Norell 2007; Agda Team 2012), and the code has been made available to download.

In order to give a clear picture of how the results can be mechanised Agda-like code is also used in the paper. Unfortunately Agda's support for total corecursion is somewhat limited,[1] so to avoid distracting details the code is written in an imaginary variant of Agda with a very clever productivity checker (and some other smaller changes). The accompanying code is written in actual Agda, sometimes using workarounds (Danielsson 2010) to convince Agda that the code is productive. There are also other, minor differences between the accompanying code and the code in the paper.

1.3 Related Work

Reynolds (1972) discusses definitional interpreters, and there is a large body of work on using monads to structure semantics and interpreters, going back at least to Moggi (1991) and Wadler (1992).

The toy language above is taken from Leroy and Grall (2009), who bring up some of the disadvantages of (inductive) big-step semantics mentioned above. The type system in Section 4 is also taken from Leroy and Grall, who discuss various formulations of type soundness (but not the main formulations given below). Finally the virtual machine and compiler defined in Sections 6–7 are also taken from Leroy and Grall, who give a compiler correctness proof.

Leroy and Grall also define a semantics based on approximations: First the semantics is defined (functionally) at "recursion depth" n; if $n = 0$, then the result $\bot$ is returned. This function is similar to the functional semantics $[\![\_]\!]$ defined in Section 3, but defined using recursion on n instead of corecursion and the partiality monad. The semantics of a term t is then defined (relationally) to be s if there is a recursion depth n_0 such that the semantics at recursion depth n is s for all $n \geqslant n_0$. Leroy and Grall prove that this semantics is equivalent to a relational, big-step semantics. This proof is close to the proof in Section 5 which shows that $[\![\_]\!]$ is equivalent to a relational, big-step semantics.

Further comparisons to the work of Leroy and Grall is included below.

The type soundness proof in Section 4 is close to proofs given by Tofte (1990) and Milner and Tofte (1991). They use ordinary, inductive big-step definitions to give semantics of languages with cyclic closures, define typing relations for values coinductively (as greatest fixpoints of monotone operators F), and use coinduction ($x \in \nu F$ if $x \in X$ for some $X \subseteq F(X)$) to prove that certain values have certain types. In this paper the value typing relation is defined inductively rather than coinductively. However, another typing relation, that for possibly non-terminating computations, is defined coinductively, and the proof still uses coinduction (which takes the form of corecursion, see Section 2).

Capretta (2005) discusses the partiality monad, and gives a semantics for partial recursive functions (primitive recursive functions plus minimisation) as a function of type $\forall\, n.\ (\mathbb{N}^n \rightharpoonup \mathbb{N}) \to ({\mathbb{N}_\bot}^n \to \mathbb{N}_\bot)$.

[1] The same applies to Coq (Coq Development Team 2011).

Nakata and Uustalu (2009) define coinductive big-step and small-step semantics, in both relational and functional style, for a while language. Their definitions do not use the partiality monad, but are trace-based, and have the property that the trace can be computed (productively) for any source term, converging or diverging. My opinion is that the *relational* big-step definition is rather technical and brittle; the authors discuss several modifications to the design which lead to absurd results, like while true do skip having an arbitrary trace. The *functional* big-step semantics avoids these issues, because the semantics is required to be a productive function from a term and an initial state to a trace. Nakata and Uustalu have extended their work to a while language with interactive input/output (2010), but in this work they use relational definitions.

Paulin-Mohring (2009) defines partial streams using (essentially) the partiality monad, shows that partial streams form a pointed CPO, and uses this CPO to define a functional semantics for (a minor variation of) Kahn networks.

Benton et al. (2009) use the partiality monad to construct a lifting operator for CPOs, and use this operator to give denotational semantics for one typed and one untyped λ-calculus; the former semantics is crash-free by construction, the latter uses $\bot$ to represent crashes. Benton and Hur (2009) define a compiler from one of these languages to a variant of the SECD machine (with a relational, small-step semantics), and prove compiler correctness.

Ghani and Uustalu (2004) introduce the partiality monad *transformer*, $\lambda M\, A.\ \nu X.\ M\ (A \uplus X)$. (In the setting of Agda M should be restricted to be strictly positive.)

Goncharov and Schröder (2011) use the partiality monad transformer (they use the term *resumption monad transformer*) to give a class of functional semantics for a concurrent language.

Rutten (1999) defines an operational semantics for a while language corecursively as a function, using a "non-constructive" variant of the partiality monad, $A_\bot = (A \times \mathbb{N}) \uplus \{\infty\}$ (where ∞ represents non-termination and the natural number stands for the number of computation steps needed to compute the value of type A). With this variant of the monad the semantics is not a *computable* function, because the semantics returns ∞ iff a program fails to terminate. Rutten also discusses weak bisimilarity and explains how to construct a compositional semantics from the operational one.

Cousot and Cousot (1992, 2009) describe *bi-inductive* definitions, which generalise inductive and coinductive definitions, and give a number of examples of their use. One of their examples is a big-step semantics for a call-by-value λ-calculus. This semantics captures both terminating and non-terminating behaviours in a single definition, with less "duplication" of rules than in Figures 1–2, but more than in Section 3. An operator F on $\wp(\mathit{Term} \times (\mathit{Term} \cup \{\bot\}))$, where *Term* stands for the set of terms and $\bot$ stands for non-termination, is first defined by the following inference rules (where v ranges over values):

$$v \Rightarrow v \qquad \frac{t_1 \Rightarrow \bot}{t_1\ t_2 \Rightarrow \bot} \qquad \frac{t_1 \Rightarrow v \qquad t_2 \Rightarrow \bot}{t_1\ t_2 \Rightarrow \bot}$$

$$\frac{t_1 \Rightarrow \lambda x.t \qquad t_2 \Rightarrow v \qquad t[x := v] \Rightarrow r}{t_1\ t_2 \Rightarrow r}$$

These rules should neither be read inductively nor coinductively. The semantics is instead obtained as the least fixpoint of F with respect to the order $\_\sqsubseteq\_$ defined by

$$X \sqsubseteq Y \quad = \quad X^+ \subseteq Y^+ \quad \wedge \quad X^- \supseteq Y^-,$$

where $Z^+ = \{\, (t, s) \in Z \mid s \neq \bot \,\}$ and $Z^- = Z \setminus Z^+$. F is not monotone with respect to $\_\sqsubseteq\_$ (which forms a complete lattice), so Cousot and Cousot give an explicit proof of the existence of a least fixpoint (for a closely related semantics).

2. The Partiality Monad

Agda is a total language (assuming that the implementation is bug-free, etc.). Ordinary data types are *inductive*. For instance, we can define the type *Fin n* of natural numbers less than n, and the type *Vec A n* of A-lists of length n, as follows:

```
data Fin : ℕ → Set where
  zero : {n : ℕ}            → Fin (1 + n)
  suc  : {n : ℕ} → Fin n → Fin (1 + n)

data Vec (A : Set) : ℕ → Set where
  []  :                                  Vec A 0
  _::_ : {n : ℕ} → A → Vec A n → Vec A (1 + n)
```

(Cons is an infix operator, _::_; the underscores mark the argument positions.) Inductive types can be destructed using structural recursion. As an example we can define a safe lookup/indexing function:

```
lookup : {A : Set} {n : ℕ} → Fin n → Vec A n → A
lookup zero    (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

The arguments within braces, {. . .}, are *implicit*, and can be omitted if Agda can infer them. To avoid clutter most implicit argument declarations are omitted, together with a few explicit instantiations of implicit arguments.

Agda also supports "infinite" data through the use of coinduction (Coquand 1994). Coinductive types can be introduced using suspensions: $\infty\, A$ is the type of suspensions, that if forced give us something of type A. Suspensions can be forced using $^\flat$, and created using $^\sharp\_$:

```
♭  : ∞ A →     A
♯_ :     A → ∞ A
```

(Here $^\sharp\_$ is a tightly binding prefix operator. In this paper nothing binds tighter except for ordinary function application.)

The partiality monad is defined coinductively as follows:

```
data _⊥ (A : Set) : Set where
  now   : A          → A⊥
  later : ∞ (A⊥) → A⊥
```

You can read this as the greatest fixpoint $\nu X.A \uplus X$.[2] The constructor now returns a value immediately, and later postpones a computation. Computations can be postponed forever:

```
never : A⊥
never = later (♯ never)
```

Here *never* is defined using *corecursion*, in a *productive* way: even though *never* can unfold forever, the next constructor can always be computed in a finite number of steps. Note that structural recursion is not supported for coinductive types, as this would allow the definition of non-productive functions.

The partiality monad is a monad, with now as its return operation, and bind defined corecursively as follows:

```
_>>=_ : A⊥ → (A → B⊥) → B⊥
now x   >>= f = f x
later x >>= f = later (♯ (♭ x >>= f))
```

If x fails to terminate, then $x \gg\!= f$ also fails to terminate, and if x terminates with a value, then f is applied to that value.

It is easy to prove the monad laws up to (strong) *bisimilarity*, which is a coinductively defined relation:

[2] This is not entirely correct in the current version of Agda (Altenkirch and Danielsson 2010), but for the purposes of this paper the differences are irrelevant.

```
data _≅_ : A⊥ → A⊥ → Set where
  now   :                        now x ≅ now x
  later : ∞ (♭ x ≅ ♭ y) → later x ≅ later y
```

(Note that the constructors have been overloaded.) This equivalence relation relates diverging computations, and it also relates computations which converge to the same value *using the same number of steps*.

Note that $\_\cong\_$ is a type of potentially infinite proof terms. Proving $x \cong y$ amounts to constructing a term with this type. This proof technique is quite different from the usual coinductive proof technique (where $x \in \nu F$ for a monotone F if $x \in X$ for some $X \subseteq F(X)$), so let me show in detail how one can prove that bind is associative:

```
associative :
  (x : A⊥) (f : A → B⊥) (g : B → C⊥) →
  (x >>= f >>= g) ≅ (x >>= λ y → f y >>= g)
```

We can do this using corecursion and case analysis on x:

```
associative (now x) f g = ?
associative (later x) f g = ?
```

We can ask Agda what types the two goals (?) have. The first one has type $f\ x \gg\!= g \cong f\ x \gg\!= g$, and can be completed by appeal to reflexivity (*refl*-$\cong$: $(x : A_\perp) \to x \cong x$ can be proved separately):

```
associative (now x) f g = refl-≅ (f x >>= g)
```

The second goal has type later $s_1 \cong$ later s_2 for some suspensions s_1 and s_2, so we can refine the goal using a later constructor and a suspension:

```
associative (later x) f g = later (♯ ?)
```

The new goal has type

$$(\flat\ x \gg\!= f \gg\!= g) \cong (\flat\ x \gg\!= \lambda\ y \to f\ y \gg\!= g),$$

so we can conclude by appeal to the coinductive hypothesis:

```
associative (later x) f g = later (♯ associative (♭ x) f g)
```

Note that the proof is productive. Agda can see this, because the corecursive call is *guarded* by a constructor and a suspension.

Strong bisimilarity is very strict. In many cases *weak* bisimilarity, which ignores finite differences in the number of steps, is more appropriate:[3]

```
data _≈_ : A⊥ → A⊥ → Set where
  now    :                        now x ≈ now x
  later  : ∞ (♭ x ≈ ♭ y) → later x ≈ later y
  later¹ :      ♭ x ≈   y → later x ≈       y
  later² :          x ≈ ♭ y →       x ≈ later y
```

This relation is defined using mixed induction and coinduction (induction nested inside coinduction, $\nu X.\mu Y.\ F\ X\ Y$). Note that later is coinductive, while later¹ and later² are inductive. An infinite sequence of later constructors is allowed, for instance to prove *never* $\approx$ *never*:

```
allowed : never ≈ never
allowed = later (♯ allowed)
```

However, only a finite number of consecutive later¹ and later² constructors is allowed, because otherwise we could prove *never* $\approx$ now x:

```
disallowed : never ≈ now x
disallowed = later¹ disallowed
```

On the other hand, because the induction is nested *inside* the coinduction it is fine to use an infinite number of later¹ or later² constructors if they are non-consecutive, with intervening later constructors:

```
also-allowed : never ≈ never
also-allowed = later² (later (♯ also-allowed))
```

If we omit the later² constructor from the definition of weak bisimilarity, then we get a preorder $\_\gtrsim\_$ with the property that $x \gtrsim y$ holds if y terminates in fewer steps than x (with the same value), but not if x terminates in strictly fewer steps than y, or if one of the two computations terminates and the other does not:

```
data _≳_ : A⊥ → A⊥ → Set where
  now    :                        now x ≳ now x
  later  : ∞ (♭ x ≳ ♭ y) → later x ≳ later y
  later¹ :      ♭ x ≳   y → later x ≳       y
```

It is easy to prove that $x \cong y$ implies $x \gtrsim y$, which in turn implies $x \approx y$.

The three relations above are transitive, but one needs to be careful when using transitivity in corecursive proofs, because otherwise one can "prove" absurd things. For instance, given *refl*-$\approx$: $(x : A_\perp) \to x \approx x$ and *trans*-$\approx$: $x \approx y \to y \approx z \to x \approx z$ we can "prove" that weak bisimilarity is trivial:

```
trivial : (x y : A⊥) → x ≈ y
trivial x y =
  trans-≈ (later² (refl-≈ x))
          (trans-≈ (later (♯ trivial x y))
                   (later¹ (refl-≈ y)))
```

This "proof" uses the following equational reasoning steps: $x \approx$ later $(\sharp\ x) \approx$ later $(\sharp\ y) \approx y$. The problem is that *trivial* is not productive: *trans*-$\approx$ is "too strict". This issue is closely related to the problem of weak bisimulation up to weak bisimilarity (Sangiorgi and Milner 1992).

Fortunately some uses of transitivity are safe. For instance, if we are proving a weak bisimilarity, then it is safe to make use of *already proved* greater-than results, in the following way (where $y \lesssim z$ is a synonym for $z \gtrsim y$):

$$x \gtrsim y \to y \approx z \to x \approx z$$
$$x \approx y \to y \lesssim z \to x \approx z$$

(Compare Sangiorgi and Milner's "expansion up to $\lesssim$".) Agda does not provide a simple way to show that these lemmas are safe, but this could be done using sized types as implemented in MiniAgda (Abel 2010).[4] With sized types one can define $x \approx^i y$ to stand for potentially incomplete proofs of $x \approx y$ of size (at least) i, and prove the following lemma:

$$\forall\ i.\ x \gtrsim y \to y \approx^i z \to x \approx^i z$$

This lemma is not "too strict": the type tells us that the (bound on the) size of the incomplete definition is preserved. Unfortunately MiniAgda, which is a research prototype, is very awkward to use in larger developments.

For more details about coinduction and corecursion in Agda, and further discussion of transitivity in a coinductive setting, see Danielsson and Altenkirch (2010).

[3] Capretta (2005) defines weak bisimilarity in a different but equivalent way.

[4] The experimental implementation of sized types in Agda does not support coinduction.

3. A Functional, Operational Semantics

This section defines an operational semantics for the untyped λ-calculus with constants. Let us start by defining the syntax of the language. Just as Leroy and Grall (2009) I use de Bruijn indices to represent variables, but I use a "well-scoped" approach, using the type system to keep track of the free variables. Terms of type *Tm n* have at most *n* free variables:

```
data Tm (n : ℕ) : Set where
  con : ℕ              → Tm n   -- Constant.
  var : Fin n          → Tm n   -- Variable.
  lam : Tm (1 + n)     → Tm n   -- Abstraction.
  _·_ : Tm n → Tm n    → Tm n   -- Application.
```

Environments and values are defined mutually:

```
mutual
  Env : ℕ → Set
  Env n = Vec Value n

  data Value : Set where
    con : ℕ                    → Value   -- Constant.
    lam : Tm (1 + n) → Env n   → Value   -- Closure.
```

Note that the body of a closure has at most one free variable which is not bound in the environment.

The language supports two kinds of "effects", partiality and crashes. The partiality monad is used to represent partiality, and the maybe monad is used to represent crashes:

$$[\![\_]\!] : Tm\ n \to Env\ n \to (Maybe\ Value)_\bot$$

(*Maybe A* has two constructors, nothing : *Maybe A* and just : *A* → *Maybe A*.) The combined monad is the maybe monad transformer ($\lambda M\ A.\ M\ (Maybe\ A)$) applied to the partiality monad. We can define a failing computation, as well as return and bind, as follows:

```
fail : (Maybe A)⊥
fail = now nothing

return : A → (Maybe A)⊥
return x = now (just x)

_>>=_ : (Maybe A)⊥ → (A → (Maybe B)⊥) → (Maybe B)⊥
now nothing >>= f = fail
now (just x) >>= f = f x
later x      >>= f = later (♯ (♭ x >>= f))
```

It should also be possible to use the reader monad transformer to handle the environment, but I believe that this would make the code harder to follow.

With the monad in place it is easy to define the semantics using two mutually (co)recursive functions:

```
mutual
  [[_]] : Tm n → Env n → (Maybe Value)⊥
  [[ con i ]] ρ = return (con i)
  [[ var x ]] ρ = return (lookup x ρ)
  [[ lam t ]] ρ = return (lam t ρ)
  [[ t₁ · t₂ ]] ρ = [[ t₁ ]] ρ >>= λ v₁ →
                    [[ t₂ ]] ρ >>= λ v₂ →
                    v₁ • v₂

  _•_ : Value → Value → (Maybe Value)⊥
  con i₁ • v₂ = fail
  lam t₁ ρ₁ • v₂ = later (♯ ([[ t₁ ]] (v₂ :: ρ₁)))
```

Constants are returned immediately, variables are looked up in the environment, and abstractions are paired up with the environment to form a closure. The interesting case is application: $t_1 \cdot t_2$ is evaluated by first evaluating t_1 to a value v_1, then (if the evaluation of t_1 terminates without a crash) t_2 to v_2, and finally evaluating the application $v_1 \bullet v_2$. If v_1 is a constant, then we crash. If v_1 is a closure, then a later constructor is emitted and the closure's body is evaluated in its environment extended by v_2. The result contains one later constructor for every β-redex that has been reduced (infinitely many in case of non-termination).

Note that this is a call-by-value semantics, with functions evaluated before arguments. Note also that the semantics is not compositional, i.e. not defined by recursion on the structure of the term, so it is not a denotational semantics. (It would be if $\_\bullet\_$ were defined prior to $[\![\_]\!]$; it is easy to construct a compositional semantics on top of this one.)

Agda does not accept the code above; it is not obvious to the productivity checker that $[\![\_]\!]$ and $\_\bullet\_$ are total (productive) functions. If bind had been a constructor, then Agda would have found that the code uses a lexicographic combination of guarded corecursion and structural recursion: every call path from $[\![\_]\!]$ to $[\![\_]\!]$ is either

1. guarded by one or more constructors and at least one suspension (and nothing else), or
2. guardedness is "preserved" (zero or more constructors/suspensions), and the term argument becomes strictly smaller.

Now, bind is not a constructor, but it does preserve guardedness: it takes apart its first argument, but introduces a new suspension before forcing an old one—in MiniAgda one can show that bind preserves the sizes of its arguments. For a formal explanation of totality, see the accompanying code.[5]

The semantics could also have been defined using continuation-passing style, and then we could have avoided the use of bind:

```
mutual
  [[_]]CPS : Tm n → Env n → (Value → (Maybe A)⊥) →
             (Maybe A)⊥
  [[ con i ]]CPS ρ k = k (con i)
  [[ var x ]]CPS ρ k = k (lookup x ρ)
  [[ lam t ]]CPS ρ k = k (lam t ρ)
  [[ t₁ · t₂ ]]CPS ρ k = [[ t₁ ]]CPS ρ (λ v₁ →
                         [[ t₂ ]]CPS ρ (λ v₂ →
                         (v₁ •CPS v₂) k))

  _•CPS_ : Value → Value → (Value → (Maybe A)⊥) →
           (Maybe A)⊥
  (con i₁ •CPS v₂) k = fail
  (lam t₁ ρ₁ •CPS v₂) k = later (♯ ([[ t₁ ]]CPS (v₂ :: ρ₁) k))
```

This definition would not have made the productivity checker any happier (it is productive, though, see the accompanying code). However, it avoids the inefficient implementation of bind; note that bind traverses the full prefix of later constructors before encountering the now constructor, if any.

Before we leave this section, let us work out a small example. The term $(\lambda x.xx)\ (\lambda x.xx)$ can be defined as follows (writing 0 instead of zero):

```
Ω : Tm 0
Ω = lam (var 0 · var 0) · lam (var 0 · var 0)
```

It is easy to show that this term does not terminate:

[5] In the accompanying code $[\![\_]\!]$ is defined using a data type containing the constructors return, _>>=_, fail and later, thus ensuring guardedness. These constructors are interpreted in the usual way in a second pass over the result. This technique is explained in detail by Danielsson (2010).

```
Ω-loops : ⟦ Ω ⟧ [] ≈ never
Ω-loops = later (♯ Ω-loops)
```

4. Type Soundness

To illustrate how the semantics can be used, let us define a type system and prove type soundness.

I follow Leroy and Grall (2009) and define recursive, simple types coinductively as follows:

```
data Ty : Set where
  nat  : Ty
  _→▹_ : ∞ Ty → ∞ Ty → Ty
```

Contexts can be defined as vectors of types:

```
Ctxt : ℕ → Set
Ctxt n = Vec Ty n
```

The type system can then be defined inductively. $\Gamma \vdash t \in \sigma$ means that t has type σ in context Γ:

```
data _⊢_∈_ (Γ : Ctxt n) : Tm n → Ty → Set where
  con : Γ ⊢ con i ∈ nat
  var : Γ ⊢ var x ∈ lookup x Γ
  lam : ♭ σ :: Γ ⊢ t ∈ ♭ τ  →  Γ ⊢ lam t ∈ σ →▹ τ
  _·_ : Γ ⊢ t₁ ∈ σ →▹ τ  →  Γ ⊢ t₂ ∈ ♭ σ  →
        Γ ⊢ t₁ · t₂ ∈ ♭ τ
```

The use of negative recursive types implies that there are well-typed terms which do not terminate. For instance, Ω is typeable with *any* type:

```
Ω-well-typed : (τ : Ty)  →  [] ⊢ Ω ∈ τ
Ω-well-typed τ = _·_ {σ = ♯ σ} {τ = ♯ τ}
                     (lam (var · var)) (lam (var · var))
  where σ = ♯ σ →▹ ♯ τ
```

(Some implicit arguments which Agda could not infer have been given explicitly using the $\{x = \ldots\}$ notation.)

Let us now prove that well-typed programs (closed terms) do not go wrong. It is easy to state what should be proved:

```
type-soundness : [] ⊢ t ∈ σ  →  ¬ (⟦ t ⟧ [] ≈ fail)
```

Here $\neg\_$ is negation ($\neg A = A \to \mathit{Empty}$, where *Empty* is the empty type). As noted by Leroy and Grall it is harder to state type soundness for usual big-step semantics, because such semantics do not distinguish between terms which go wrong and terms which fail to terminate.

We can start by defining a reusable predicate transformer which lifts predicates on A to predicates on $(\mathit{Maybe}\ A)_\perp$. If $\mathit{Lift}\ P\ x$ holds, then we know both that the computation x does not crash, and that if x terminates with a value, then the value satisfies P. *Lift* is defined coinductively as follows:

```
data Lift (P : A → Set) : (Maybe A)⊥ → Set where
  now-just : P x                → Lift P (return x)
  later    : ∞ (Lift P (♭ x)) → Lift P (later x)
```

The proof below uses the fact that bind "preserves" *Lift*:

```
_>>=-cong_ : Lift P x → ({x : A} → P x → Lift Q (f x)) →
             Lift Q (x >>= f)
```

Let us now define some typing predicates for values and computations, introduced mainly as part of the proof of type soundness. $WF_{\mathrm{V}}\ \sigma\ v$ means that the value v is well-formed with respect to the type σ. This relation is defined inductively, mutually with a corresponding relation for environments:

```
mutual
  data WF_V : Ty → Value → Set where
    con : WF_V nat (con i)
    lam : ♭ σ :: Γ ⊢ t ∈ ♭ τ → WF_E Γ ρ →
          WF_V (σ →▹ τ) (lam t ρ)
  data WF_E : Ctxt n → Env n → Set where
    []  :                          WF_E []       []
    _::_ : WF_V σ v → WF_E Γ ρ → WF_E (σ :: Γ) (v :: ρ)
```

The most interesting case above is that for closures. A closure $\mathsf{lam}\ t\ \rho$ is well-formed with respect to $\sigma \to\!\!\triangleright\ \tau$ if there is a context Γ such that $\Gamma \vdash \mathsf{lam}\ t \in \sigma \to\!\!\triangleright\ \tau$ and ρ is well-formed with respect to Γ. The predicates are related by the following unsurprising lemma:

```
lookup_wf : (x : Fin n)  →  WF_E Γ ρ  →
            WF_V (lookup x Γ) (lookup x ρ)
```

We can use the predicate transformer introduced above to lift WF_{V} to computations:

```
WF⊥ : Ty → (Maybe Value)⊥ → Set
WF⊥ σ x = Lift (WF_V σ) x
```

Non-terminating computations are well-formed, and terminating computations are well-formed if they are successful (not nothing) and the value is well-formed. The following lemma implies that type soundness can be established by showing that $[\![\ t\ ]\!]\ []$ is well-formed:

```
does-not-go-wrong : WF⊥ σ x → ¬ (x ≈ fail)
does-not-go-wrong (now-just _) ()
does-not-go-wrong (later wf)     (later¹ eq) =
  does-not-go-wrong (♭ wf) eq
```

Recall that negation is a function into the empty type. The lemma is proved by structural recursion: induction on the structure of the proof of $x \approx \mathit{fail}$. The first clause contains an "absurd pattern", (), to indicate that there is no constructor application of type $\mathit{return}\ v \approx \mathit{fail}$.

We can now prove the main lemma, which states that the computations resulting from evaluating well-typed terms in well-formed environments are well-formed. This lemma uses the same form of nested corecursion/structural recursion as the definition of the semantics:

```
mutual
  [[]]_wf : Γ ⊢ t ∈ σ  →  WF_E Γ ρ  →  WF⊥ σ ([[ t ]] ρ)
  [[]]_wf con       ρ_wf = now-just con
  [[]]_wf (var {x = x}) ρ_wf = now-just (lookup_wf x ρ_wf)
  [[]]_wf (lam t∈)  ρ_wf = now-just (lam t∈ ρ_wf)
  [[]]_wf (t₁∈ · t₂∈) ρ_wf =
    [[]]_wf t₁∈ ρ_wf >>=-cong λ f_wf →
    [[]]_wf t₂∈ ρ_wf >>=-cong λ v_wf →
    •_wf f_wf v_wf

  •_wf : WF_V (σ →▹ τ) f  →  WF_V (♭ σ) v  →
         WF⊥ (♭ τ) (f • v)
  •_wf (lam t₁∈ ρ₁wf) v₂wf =
    later (♯ [[]]_wf t₁∈ (v₂wf :: ρ₁wf))
```

The implicit variable pattern $\{x = x\}$ is used to bind the variable x, which is used on the right-hand side.

Finally we can conclude:

```
type-soundness : [] ⊢ t ∈ σ  →  ¬ (⟦ t ⟧ [] ≈ fail)
type-soundness t∈ = does-not-go-wrong ([[]]_wf t∈ [])
```

Note that there is only one case for application in the proof above (plus one sub-case in $\bullet_{\mathrm{wf}}$).

The proof of type soundness is formulated for a functional semantics defined using environments and closures, whereas Leroy and Grall (2009) prove type soundness for relational semantics defined using substitutions. I have chosen to use environments and closures in this paper to avoid distracting details related to substitutions. However, given an implementation of the operation which substitutes a term for variable zero it is easy to define a substitution-based functional semantics using the partiality monad, and given a proof showing that this operation preserves types it is easy to adapt the proof above to this semantics. See the accompanying code for details.

The proof above can be compared to a typical type soundness proof formulated for a relational, substitution-based small-step semantics. Such a proof often amounts to proving progress and preservation:

$$\begin{aligned}&\mathit{progress} &&: [\,] \vdash t \in \sigma \;\to\; \mathit{Value}\ t \uplus \exists\, \lambda\, t' \to t \rightsquigarrow t'\\ &\mathit{preservation} &&: [\,] \vdash t \in \sigma \;\to\; t \rightsquigarrow t' \;\to\; [\,] \vdash t' \in \sigma\end{aligned}$$

Here *Value t* means that t is a value, $\_\rightsquigarrow\_$ is the small-step relation, and $\exists\, \lambda\, t' \to \ldots$ can be read as "there exists a t' such that…". Given these two lemmas one can prove type soundness using classical reasoning (Leroy and Grall 2009):

$$\begin{aligned}&\mathit{type\text{-}soundness} : [\,] \vdash t \in \sigma \;\to\\ &\qquad t \rightsquigarrow^{\infty} \uplus \exists\, \lambda\, t' \to t \rightsquigarrow^{\star} t' \times \mathit{Value}\ t'\end{aligned}$$

Here $\_\rightsquigarrow^{\star}\_$ is the reflexive transitive closure of $\_\rightsquigarrow\_$, $t \rightsquigarrow^{\infty}$ means that t can reduce forever, and $\_\times\_$ can be read as "and". (Note that this statement of type soundness is inappropriate for non-deterministic languages, as it does not rule out the possibility of crashes.) The lemma $[\![]\!]_{\mathrm{wf}}$ above can be seen as encompassing both progress and preservation, plus the combination of these two lemmas into type soundness. This combination does not need to involve classical reasoning, because $WF_{\perp}$ is defined coinductively.

5. The Semantics are Classically Equivalent

Let us now prove that the semantics given in Section 3 is classically equivalent to a relational semantics.

The semantics given in Figures 1–2 can be adapted to a setting with well-scoped terms and de Bruijn indices in the following way:

$$\begin{aligned}&\textbf{data}\ \_\vdash\_\Downarrow\_\ (\rho : \mathit{Env}\ n) : \mathit{Tm}\ n \to \mathit{Value} \to \mathit{Set}\ \textbf{where}\\ &\quad \mathsf{con} : \rho \vdash \mathsf{con}\ i \Downarrow \mathsf{con}\ i\\ &\quad \mathsf{var} : \rho \vdash \mathsf{var}\ x \Downarrow \mathit{lookup}\ x\ \rho\\ &\quad \mathsf{lam} : \rho \vdash \mathsf{lam}\ t \Downarrow \mathsf{lam}\ t\ \rho\\ &\quad \mathsf{app} : \rho \vdash t_1 \Downarrow \mathsf{lam}\ t'\ \rho' \;\to\; \rho \vdash t_2 \Downarrow v' \;\to\\ &\qquad\quad v' :: \rho' \vdash t' \Downarrow v \;\to\; \rho \vdash t_1 \cdot t_2 \Downarrow v\\ &\textbf{data}\ \_\vdash\_\Uparrow\ (\rho : \mathit{Env}\ n) : \mathit{Tm}\ n \to \mathit{Set}\ \textbf{where}\\ &\quad \mathsf{app}^{\mathsf{l}} : \infty\,(\rho \vdash t_1 \Uparrow) \;\to\; \rho \vdash t_1 \cdot t_2 \Uparrow\\ &\quad \mathsf{app}^{\mathsf{r}} : \rho \vdash t_1 \Downarrow v \;\to\; \infty\,(\rho \vdash t_2 \Uparrow) \;\to\; \rho \vdash t_1 \cdot t_2 \Uparrow\\ &\quad \mathsf{app} : \rho \vdash t_1 \Downarrow \mathsf{lam}\ t'\ \rho' \;\to\; \rho \vdash t_2 \Downarrow v' \;\to\\ &\qquad\quad \infty\,(v' :: \rho' \vdash t' \Uparrow) \;\to\; \rho \vdash t_1 \cdot t_2 \Uparrow\\ &\_\vdash\_\lightning : \mathit{Env}\ n \to \mathit{Tm}\ n \to \mathit{Set}\\ &\rho \vdash t \lightning = \neg\,(\exists\, \lambda\, v \to \rho \vdash t \Downarrow v) \times \neg\,(\rho \vdash t \Uparrow)\end{aligned}$$

Note that $\_\vdash\_\Downarrow\_$ is defined inductively and $\_\vdash\_\Uparrow$ coinductively.

How should we state the equivalence of $\_\vdash\_\Downarrow\_/\_\vdash\_\Uparrow/\_\vdash\_\lightning$ and $[\![\_]\!]$? The following may seem like a suitable statement:

$$\begin{aligned}\rho \vdash t \Downarrow v \quad &\Leftrightarrow \quad [\![\, t \,]\!]\, \rho \approx \mathit{return}\ v\\ \rho \vdash t \Uparrow \quad &\Leftrightarrow \quad [\![\, t \,]\!]\, \rho \approx \mathit{never}\\ \rho \vdash t \lightning \quad &\Leftrightarrow \quad [\![\, t \,]\!]\, \rho \approx \mathit{fail}\end{aligned}$$

However, in a constructive setting one cannot prove that $[\![\, t \,]\!]\, \rho \approx \mathit{never}$ implies $\rho \vdash t \Uparrow$. To see why, let us try. Assume that we have a proof p of type $[\![\, t_1 \cdot t_2 \,]\!]\, \rho \approx \mathit{never}$. Now we need to construct a proof starting with either $\mathsf{app}^{\mathsf{l}}$, $\mathsf{app}^{\mathsf{r}}$ or app. In order to do this we need to know whether t_1 terminates or not, but this is not decidable given only the proof p. It also seems unlikely that we can prove that $\rho \vdash t \lightning$ implies $[\![\, t \,]\!]\, \rho \approx \mathit{fail}$: one might imagine that this can be proved by just executing $[\![\, t \,]\!]\, \rho$ until it terminates and then performing a case analysis, but the fact that t does not fail to terminate is not (obviously) enough to convince Agda that it does terminate.

We can avoid these issues by assuming the following form of excluded middle, which states that everything (in *Set*) is decidable:

$$\begin{aligned}&\mathit{EM} : \mathit{Set}_1\\ &\mathit{EM} = (A : \mathit{Set}) \;\to\; A \uplus \neg\, A\end{aligned}$$

We end up with the following six proof obligations:

$$\begin{aligned}&\rho \vdash t \Downarrow v &&\to\ [\![\, t \,]\!]\, \rho \approx \mathit{return}\ v && (1)\\ &\rho \vdash t \Uparrow &&\to\ [\![\, t \,]\!]\, \rho \approx \mathit{never} && (2)\\ &[\![\, t \,]\!]\, \rho \approx \mathit{return}\ v &&\to\ \rho \vdash t \Downarrow v && (3)\\ \mathit{EM} \to\ &[\![\, t \,]\!]\, \rho \approx \mathit{never} &&\to\ \rho \vdash t \Uparrow && (4)\\ \mathit{EM} \to\ &\rho \vdash t \lightning &&\to\ [\![\, t \,]\!]\, \rho \approx \mathit{fail} && (5)\\ &[\![\, t \,]\!]\, \rho \approx \mathit{fail} &&\to\ \rho \vdash t \lightning && (6)\end{aligned}$$

The last two follow easily from the previous ones, so let us focus on the first four:

1. Given $p : \rho \vdash t \Downarrow v$ it is easy to prove $[\![\, t \,]\!]\, \rho \approx \mathit{return}\ v$ by recursion on the structure of p.

   The only interesting case is application. Let us introduce the following abbreviation:

   $$x_1\ [\![\cdot]\!]\ x_2 = x_1 \gg\!= \lambda\, v_1 \to x_2 \gg\!= \lambda\, v_2 \to v_1 \bullet v_2$$

   We can then proceed as follows (using the same names as in the app constructor's type signature):

   $$\begin{aligned}&[\![\, t_1 \cdot t_2 \,]\!]\, \rho && \cong\\ &[\![\, t_1 \,]\!]\, \rho\ [\![\cdot]\!]\ [\![\, t_2 \,]\!]\, \rho && \approx\\ &\mathit{return}\ (\mathsf{lam}\ t'\ \rho')\ [\![\cdot]\!]\ \mathit{return}\ v' && \gtrsim\\ &[\![\, t' \,]\!]\, (v' :: \rho') && \approx\\ &\mathit{return}\ v\end{aligned}$$

   The inductive hypothesis is used twice in the second step and once in the last one.

2. One can prove that $\rho \vdash t \Uparrow$ implies $[\![\, t \,]\!]\, \rho \approx \mathit{never}$ using corecursion plus an inner recursion on the structure of t.

   In the case of the app constructor we can proceed as follows:

   $$\begin{aligned}&[\![\, t_1 \cdot t_2 \,]\!]\, \rho && \cong\\ &[\![\, t_1 \,]\!]\, \rho\ [\![\cdot]\!]\ [\![\, t_2 \,]\!]\, \rho && \gtrsim\\ &\mathit{return}\ (\mathsf{lam}\ t'\ \rho')\ [\![\cdot]\!]\ \mathit{return}\ v' && \cong\\ &\mathsf{later}\ (^{\sharp}\, [\![\, t' \,]\!]\, (v' :: \rho')) && \approx\\ &\mathit{never}\end{aligned}$$

   The second step uses (1) twice, once for $p_1 : \rho \vdash t_1 \Downarrow \mathsf{lam}\ t'\ \rho'$ and once for $p_2 : \rho \vdash t_2 \Downarrow v'$, plus the fact that $x \approx \mathsf{now}\ v$ implies that $x \gtrsim \mathsf{now}\ v$. The last step uses the coinductive hypothesis (under a guard) for $p_3 : v' :: \rho' \vdash t' \Uparrow$.

   The $\mathsf{app}^{\mathsf{l}}$ case is different:

   $$\begin{aligned}&[\![\, t_1 \cdot t_2 \,]\!]\, \rho && \cong\\ &[\![\, t_1 \,]\!]\, \rho\ [\![\cdot]\!]\ [\![\, t_2 \,]\!]\, \rho && \approx\\ &\mathit{never}\quad [\![\cdot]\!]\ [\![\, t_2 \,]\!]\, \rho && \cong\\ &\mathit{never}\end{aligned}$$

   The last step uses the fact that *never* is a left zero of bind. The second step uses the *inductive* hypothesis for $p : \rho \vdash t_1 \Uparrow$; note that t_1 is structurally smaller than $t_1 \cdot t_2$, and that this call is not guarded.

The $\mathsf{app}^{\mathsf{r}}$ case is similar to the $\mathsf{app}^{\mathsf{l}}$ one, and omitted.

Note that the use of transitivity in this proof is safe, as discussed in Section 2.

3. Given $p : [\![t]\!]\ \rho \approx \mathit{return}\ v$ one can observe that p cannot contain the constructors later or $\mathsf{later}^{\mathsf{r}}$: it must have the form $\mathsf{later}^{\mathsf{l}}\ (\ldots\ (\mathsf{later}^{\mathsf{l}}\ \mathsf{now})\ \ldots)$, with a finite number of $\mathsf{later}^{\mathsf{l}}$ constructors—one for every β-reduction in the computation of $[\![t]\!]\ \rho$. Let the *size* of p be this number. One can prove that $[\![t]\!]\ \rho \approx \mathit{return}\ v$ implies $\rho \vdash t \Downarrow v$ by complete induction on this size.

   Only the application case is interesting. We can prove the following inversion lemma:

$$(x \gg\!= f) \approx \mathit{return}\ v \rightarrow \exists\, \lambda\, v' \rightarrow (x \approx \mathit{return}\ v') \times (f\ v' \approx \mathit{return}\ v)$$

   Here the size of the left-hand proof is equal to the sum of the sizes of the two right-hand proofs. If we have $[\![t_1 \cdot t_2]\!]\ \rho \approx \mathit{return}\ v$, then we can use inversion twice plus case analysis to deduce that $[\![t_1]\!]\ \rho \approx \mathit{return}\ (\mathsf{lam}\ t'\ \rho')$ and $[\![t_2]\!]\ \rho \approx \mathit{return}\ v'$ for some t', ρ', v' such that $[\![t']\!]\ (v' :: \rho') \approx \mathit{return}\ v$. We can finish by applying app to three instances of the inductive hypothesis, after making sure that the proofs are small enough.

   This proof is a bit awkward when written out in detail, due to the use of sizes.

4. Finally we should prove that excluded middle and $[\![t]\!]\ \rho \approx \mathit{never}$ imply $\rho \vdash t \Uparrow$. This can be proved using corecursion.

   As before the only interesting case is application. We can prove the following inversion lemma by using excluded middle:

$$(x \gg\!= f) \approx \mathit{never} \rightarrow x \approx \mathit{never} \uplus \exists\, \lambda\, v \rightarrow (x \approx \mathit{return}\ v) \times (f\ v \approx \mathit{never})$$

   If $x \gg\!= f$ does not terminate, then either x fails to terminate, or x terminates with a value v and $f\ v$ does not terminate. Given a proof of $[\![t_1 \cdot t_2]\!]\ \rho \approx \mathit{never}$ we can use inversion twice to determine which of $\mathsf{app}^{\mathsf{l}}$, $\mathsf{app}^{\mathsf{r}}$ and app to emit, in each case continuing corecursively (and in the latter two cases also using (3)).

6. Virtual Machine

This section defines a virtual machine (VM), following Leroy and Grall (2009) but defining the semantics functionally instead of relationally, and using a well-scoped approach. (The accompanying code contains a relational semantics and a proof showing that it is equivalent to the functional one.)

The VM is stack-based, and uses the following instructions:

```
mutual
  data Instr (n : ℕ) : Set where
    var : Fin n           → Instr n   -- Push variable.
    con : ℕ               → Instr n   -- Push constant.
    clo : Code (1 + n)    → Instr n   -- Push closure.
    app :                   Instr n   -- Apply function.
    ret :                   Instr n   -- Return.

  Code : ℕ → Set
  Code n = List (Instr n)
```

Instructions of type *Instr n* have at most n free variables. The type family *Code* consists of sequences of instructions.

Values and environments (*VM-Value* and *VM-Env*) are defined as in Section 3, but using *Code* instead of *Tm* in the definition of closures. Stacks contain values and return frames:

```
data Stack-element : Set where
  val : VM-Value          → Stack-element
  ret : Code n → VM-Env n → Stack-element

Stack : Set
Stack = List Stack-element
```

The VM operates on states containing three components, the code, a stack, and an environment:

```
data State : Set where
  ⟨_,_,_⟩ : Code n → Stack → VM-Env n → State
```

The result of running the VM one step, starting in a given state, is either a new state, normal termination with a value, or abnormal termination (a crash):

```
data Result : Set where
  continue : State    → Result
  done     : VM-Value → Result
  crash    :            Result
```

The function *step* (see Figure 3) shows how the result is computed. Given *step* it is easy to define the VM's semantics corecursively:

```
exec : State → (Maybe VM-Value)⊥
exec s with step s
... | continue s′ = later (♯ exec s′)
... | done v      = return v
... | crash       = fail
```

In a state s, run *step s*. If the result is continue s', continue running from s'; if it is done v, return v; and if it is crash, fail.

The function *exec* is an example of a functional, *small-step* operational semantics. As before it is clear that the semantics is deterministic and computable, and just as with a relational small-step semantics we avoid duplication of rules. However, the use of a wild-card in the last clause of *step* means that it is possible to forget a rule. If we tried to omit one of the clauses from the definition of $[\![\_]\!]$ (Section 3), then the definition would be rejected, but this is not the case for the first six clauses of *step*.

7. Compiler Correctness

Let us now define a compiler from *Tm* to *Code* and prove that it preserves the semantics of the input program. The definition follows Leroy and Grall (2009), but uses a code continuation to avoid the use of list append and some proof overhead (Hutton 2007, Section 13.7):

```
comp : Tm n → Code n → Code n
comp (con i)   c = con i :: c
comp (var x)   c = var x :: c
comp (lam t)   c = clo (comp t [ret]) :: c
comp (t₁ · t₂) c = comp t₁ (comp t₂ (app :: c))
```

We can also "compile" values:

```
compᵥ : Value → VM-Value
compᵥ (con i)     = con i
compᵥ (lam t ρ)   = lam (comp t [ret]) (map compᵥ ρ)
```

I state compiler correctness as follows:

```
correct : (t : Tm 0) →
          exec ⟨ comp t [], [], [] ⟩ ≈
            ([[ t ]] [] >>= λ v → return (compᵥ v))
```

Given a closed term t, the result of running the corresponding compiled code (*comp t* [ ]) on the VM (starting with an empty stack and environment), should be the same as evaluating the term (in

$$
\begin{array}{lllll}
\mathit{step} : \mathit{State} \to \mathit{Result} \\
\mathit{step}\ \langle\ [\,] & ,\ \mathsf{val}\ v :: [\,] & ,[\,]\ \rangle & = & \mathsf{done}\ v \\
\mathit{step}\ \langle\ \mathsf{var}\ x :: c, & & s, \rho\ \rangle & = & \mathsf{continue}\ \langle\ c\ ,\mathsf{val}\ (\mathit{lookup}\ x\ \rho) :: s, \quad \rho\ \rangle \\
\mathit{step}\ \langle\ \mathsf{con}\ i :: c, & & s, \rho\ \rangle & = & \mathsf{continue}\ \langle\ c\ ,\mathsf{val}\ (\mathsf{con}\ i) :: s, \quad \rho\ \rangle \\
\mathit{step}\ \langle\ \mathsf{clo}\ c' :: c, & & s, \rho\ \rangle & = & \mathsf{continue}\ \langle\ c\ ,\mathsf{val}\ (\mathsf{lam}\ c'\ \rho) :: s, \quad \rho\ \rangle \\
\mathit{step}\ \langle\ \mathsf{app} & :: c, \mathsf{val}\ v :: \mathsf{val}\ (\mathsf{lam}\ c'\ \rho') & :: s, \rho\ \rangle & = & \mathsf{continue}\ \langle\ c', \mathsf{ret}\ c\ \rho :: s, v :: \rho'\ \rangle \\
\mathit{step}\ \langle\ \mathsf{ret} & :: c, \mathsf{val}\ v :: \mathsf{ret}\ c'\ \rho' & :: s, \rho\ \rangle & = & \mathsf{continue}\ \langle\ c', \mathsf{val}\ v :: s, \quad \rho'\ \rangle \\
\mathit{step}\ \_ & & & = & \mathsf{crash}
\end{array}
$$

Figure 3. A function which computes the result of running the virtual machine one step from a given state.

an empty environment) and, if evaluation terminates with a value, return the "compiled" variant of this value.

We can compare this statement to a corresponding statement phrased for relational semantics:

$$
\begin{array}{lll}
([\,] \vdash t \Downarrow v & \Leftrightarrow & \langle\ \mathit{comp}\ t\ [\,], [\,], [\,]\ \rangle \leadsto^{\star} \\
 & & \quad \langle\ [\,], \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v) :: [\,], [\,]\ \rangle) \times \\
([\,] \vdash t \Uparrow & \Leftrightarrow & \langle\ \mathit{comp}\ t\ [\,], [\,], [\,]\ \rangle \leadsto^{\infty}) \qquad \times \\
([\,] \vdash t\ \lightning & \Leftrightarrow & \langle\ \mathit{comp}\ t\ [\,], [\,], [\,]\ \rangle \leadsto^{\lightning})
\end{array}
$$

Here $\_\leadsto\_ : \mathit{State} \to \mathit{State} \to \mathit{Set}$ is the VM's small-step relation, $\_\leadsto^{\star}\_$ its reflexive transitive closure, $s \leadsto^{\infty}$ means that there is an infinite transition sequence starting in s, and $s \leadsto^{\lightning}$ means that there is a "stuck" transition sequence starting in s (i.e., a sequence which cannot be extended further, and which does not end with a state of the form $\langle\ [\,], \mathsf{val}\ \_ :: [\,], [\,]\ \rangle$). I prefer the statement of *correct* above: I find it easier to understand and get correct.

Let us now prove *correct*. In order to do this the statement can be generalised as follows:

$$
\begin{array}{l}
\mathit{correct}' : \\
\quad (t : \mathit{Tm}\ n)\ \{k : \mathit{Value} \to (\mathit{Maybe}\ \mathit{VM\text{-}Value})_{\bot}\} \\
\quad (\mathit{hyp} : (v : \mathit{Value}) \to \\
\qquad \mathit{exec}\ \langle\ c, \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v) :: s, \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho\ \rangle \approx k\ v) \to \\
\quad \mathit{exec}\ \langle\ \mathit{comp}\ t\ c, s, \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho\ \rangle \approx ([\![\ t\ ]\!]\ \rho \ggg k)
\end{array}
$$

This statement is written in continuation-passing style to avoid some uses of transitivity (which can be problematic, as discussed in Section 2). The statement is proved mutually with the following one:

$$
\begin{array}{l}
\bullet\text{-}\mathit{correct} : \\
\quad (v_1\ v_2 : \mathit{Value})\ \{k : \mathit{Value} \to (\mathit{Maybe}\ \mathit{VM\text{-}Value})_{\bot}\} \\
\quad (\mathit{hyp} : (v : \mathit{Value}) \to \\
\qquad \mathit{exec}\ \langle\ c, \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v) :: s, \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho\ \rangle \approx k\ v) \to \\
\quad \mathit{exec}\ \langle\ \mathsf{app} :: c, \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v_2) :: \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v_1) :: s, \\
\qquad\quad \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho\ \rangle \\
\qquad \approx (v_1 \bullet v_2 \ggg k)
\end{array}
$$

The statements can be proved using the same recursion structure as $[\![\_]\!]_{\mathrm{CPS}}/\_\bullet_{\mathrm{CPS}}\_$: mixed corecursion/structural recursion.

The interesting case of *correct'* is application, where we can proceed as follows (with safe uses of transitivity):

$$
\begin{array}{l}
\mathit{exec}\ \langle\ \mathit{comp}\ t_1\ (\mathit{comp}\ t_2\ (\mathsf{app} :: c)), s, \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho\ \rangle \qquad \approx \\
[\![\ t_1\ ]\!]\ \rho \ggg \lambda\ v_1 \to\ [\![\ t_2\ ]\!]\ \rho \ggg \lambda\ v_2 \to v_1 \bullet v_2 \ \ggg k \cong \\
[\![\ t_1\ ]\!]\ \rho \ggg \lambda\ v_1 \to ([\![\ t_2\ ]\!]\ \rho \ggg \lambda\ v_2 \to v_1 \bullet v_2) \ggg k \cong \\
([\![\ t_1\ ]\!]\ \rho \ggg \lambda\ v_1 \to\ [\![\ t_2\ ]\!]\ \rho \ggg \lambda\ v_2 \to v_1 \bullet v_2) \ggg k \cong \\
[\![\ t_1 \cdot t_2\ ]\!]\ \rho \ggg k
\end{array}
$$

The last three steps use associativity of bind twice. (These uses of associativity could have been avoided by using continuation-passing style instead of bind when defining the semantics. See the accompanying code.) The first step is more complicated. Here is its proof term:

$$
\mathit{correct}'\ t_1\ (\lambda\ v_1 \to \mathit{correct}'\ t_2\ (\lambda\ v_2 \to \bullet\text{-}\mathit{correct}\ v_1\ v_2\ \mathit{hyp}))
$$

First an appeal to the inductive hypothesis (t_1 is structurally smaller than $t_1 \cdot t_2$), then, in the continuation, another appeal to the inductive hypothesis, and finally, in the nested continuation, a use of $\bullet$-*correct*.

The interesting case of $\bullet$-*correct* is when v_1 is a closure, $\mathsf{lam}\ t_1\ \rho_1$, in which case we need to prove that

$$
\begin{array}{l}
\mathit{exec}\ \langle\ \mathsf{app} :: c, \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v_2) :: \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ (\mathsf{lam}\ t_1\ \rho_1)) :: s, \\
\qquad \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho\ \rangle
\end{array}
$$

is weakly bisimilar to

$$
\mathsf{lam}\ t_1\ \rho_1 \bullet v_2 \ggg k.
$$

We can start by emitting a later constructor and suspension:

$$
\mathsf{later}\ (\sharp\ ?)
$$

The question mark should be replaced by a proof showing that

$$
\begin{array}{l}
\mathit{exec}\ \langle\ \mathit{comp}\ t_1\ [\mathsf{ret}], \mathsf{ret}\ c\ (\mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho) :: s, \\
\qquad \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ (v_2 :: \rho_1)\ \rangle
\end{array}
$$

is weakly bisimilar to

$$
[\![\ t_1\ ]\!]\ (v_2 :: \rho_1) \ggg k.
$$

This can be proved by appeal to the coinductive hypothesis:

$$
\mathit{correct}'\ t_1\ (\lambda\ v \to \mathsf{later}^{\mathrm{l}}\ (\mathit{hyp}\ v))
$$

Here the use of $\mathsf{later}^{\mathrm{l}}$ corresponds to the reduction of

$$
\begin{array}{l}
\mathit{exec}\ \langle\ [\mathsf{ret}], \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v) :: \mathsf{ret}\ c\ (\mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho) :: s, \\
\qquad \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ (v_2 :: \rho_1)\ \rangle
\end{array}
$$

to

$$
\mathit{exec}\ \langle\ c, \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v) :: s, \mathit{map}\ \mathit{comp}_{\mathrm{v}}\ \rho\ \rangle,
$$

which has the right form for the use of *hyp*.

The proof sketch above—and especially the compact proof terms—may look a bit bewildering. Fortunately one does not have to understand every detail of a machine-checked proof. It is more important to understand the statement of the theorem.[6] Furthermore, the writer of the proof has the support of a proof assistant, that in my case provided much help with the construction of the proof terms.

The proof above can be compared to that of Leroy and Grall (2009), who prove the following two implications (in their slightly different setting):

$$
\begin{array}{lll}
[\,] \vdash t \Downarrow v & \to & \langle\ \mathit{comp}\ t\ [\,], [\,], [\,]\ \rangle \leadsto^{\star} \\
 & & \quad \langle\ [\,], \mathsf{val}\ (\mathit{comp}_{\mathrm{v}}\ v) :: [\,], [\,]\ \rangle \\
[\,] \vdash t \Uparrow & \to & \langle\ \mathit{comp}\ t\ [\,], [\,], [\,]\ \rangle \leadsto^{\infty}
\end{array}
$$

[6] With the caveat that one should not put too much trust into Agda, which is a very experimental system.

Consider application. In the proof above there is *one* case for application, with two sub-cases, one for crashes and one for closures. In the proof of the two implications there are *four* cases for application: one in case of termination and three for non-terminating applications. The rule duplication in the semantics shows up as rule duplication in the proof.

8. Non-determinism

The compiler correctness statement used above is sometimes too restrictive (Leroy 2009). For instance, evaluation order may be left up to the compiler. This section illustrates how this kind of situation can be handled by defining a non-deterministic language, and implementing a compiler that implements one out of many possible semantics for this language.

The syntax of the language defined in Section 3 is extended with a term-former for non-deterministic choice:

$$\_|\_ \;:\; \mathit{Tm}\ n \to \mathit{Tm}\ n \to \mathit{Tm}\ n$$

The semantic domain is now the maybe monad transformer applied to the partiality monad transformer ($\lambda M\ A.\ \nu X.\ M\ (A \uplus X)$ for strictly positive monads M) applied to a non-determinism monad ($\lambda A.\ \mu X.\ A \uplus X \times X$; Moggi (1990)), implemented monolithically as follows:

$$\begin{array}{l}
\textbf{data}\ D\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad \mathsf{fail} \;:\; D\ A \\
\quad \mathsf{return} \;:\; A \to D\ A \\
\quad _|_ \;:\; D\ A \to D\ A \to D\ A \\
\quad \mathsf{later} \;:\; \infty\ (D\ A) \to D\ A \\
_ \gg\!= _ \;:\; D\ A \to (A \to D\ B) \to D\ B \\
\mathsf{fail} \gg\!= f = \mathsf{fail} \\
\mathsf{return}\ x \gg\!= f = f\ x \\
(x_1 \mid x_2) \gg\!= f = (x_1 \gg\!= f) \mid (x_2 \gg\!= f) \\
\mathsf{later}\ x \gg\!= f = \mathsf{later}\ (\sharp\ (\flat\ x \gg\!= f))
\end{array}$$

As before the monad laws hold up to strong bisimilarity, which can be defined as follows:

$$\begin{array}{l}
\textbf{data}\ _\cong_ \;:\; D\ A \to D\ A \to \mathit{Set}\ \textbf{where} \\
\quad \mathsf{fail} \;:\; \mathsf{fail} \cong \mathsf{fail} \\
\quad \mathsf{return} \;:\; \mathsf{return}\ x \cong \mathsf{return}\ x \\
\quad _|_ \;:\; x_1 \cong y_1 \to x_2 \cong y_2 \to x_1 \mid x_2 \cong y_1 \mid y_2 \\
\quad \mathsf{later} \;:\; \infty\ (\flat\ x \cong \flat\ y) \to \mathsf{later}\ x \cong \mathsf{later}\ y
\end{array}$$

Finally we can extend the semantics by adding a clause for choice (note that $\_|\_$ is overloaded):

$$[\![\ t_1 \mid t_2\ ]\!]\ \rho = [\![\ t_1\ ]\!]\ \rho \mid [\![\ t_2\ ]\!]\ \rho$$

It may be worth pointing out that now the semantics is no longer deterministic, despite being defined as a function.

As an example we can define a call-by-value fixpoint combinator ($Z = \lambda f.\ (\lambda g.\ f\ (\lambda x.\ g\ g\ x))\ (\lambda g.\ f\ (\lambda x.\ g\ g\ x))$) and a non-deterministic non-terminating term ($t = Z\ (\lambda f\ x.\ f\ x \mid f\ x)\ 0$):

$$\begin{array}{l}
Z : \mathit{Tm}\ 0 \\
Z = \mathsf{lam}\ (h \cdot h) \\
\quad \textbf{where}\ h = \mathsf{lam}\ (\mathsf{var}\ 1 \cdot \mathsf{lam}\ (\mathsf{var}\ 1 \cdot \mathsf{var}\ 1 \cdot \mathsf{var}\ 0)) \\
t : \mathit{Tm}\ 0 \\
t = Z \cdot \mathsf{lam}\ (\mathsf{lam}\ (\mathsf{var}\ 1 \cdot \mathsf{var}\ 0 \mid \mathsf{var}\ 1 \cdot \mathsf{var}\ 0)) \cdot \mathsf{con}\ 0
\end{array}$$

The semantics of t, $[\![\ t\ ]\!]\ [\,]$, is strongly bisimilar to *t-sem*:

$$\begin{array}{l}
\mathit{t\text{-}sem} : D\ \mathit{Value} \\
\mathit{t\text{-}sem} = \mathsf{later}\ (\sharp\ \mathsf{later}\ (\sharp\ \mathsf{later}\ (\sharp\ \mathsf{later}\ (\sharp\ (\mathit{t\text{-}sem} \mid \mathit{t\text{-}sem})))))
\end{array}$$

The virtual machine is unchanged, so the compiler correctness statement will relate deterministic and non-deterministic computations. To do this we can use the following variant of weak bisimilarity:

$$\begin{array}{l}
\textbf{data}\ _\approx^{\in}_ \;:\; (\mathit{Maybe}\ A)_\perp \to D\ A \to \mathit{Set}\ \textbf{where} \\
\quad \mathsf{fail} \;:\; \mathsf{now}\ \mathsf{nothing} \approx^{\in} \mathsf{fail} \\
\quad \mathsf{return} \;:\; \mathsf{now}\ (\mathsf{just}\ x) \approx^{\in} \mathsf{return}\ x \\
\quad |^{\mathrm{l}} \;:\; x \approx^{\in} y_1 \to x \approx^{\in} y_1 \mid y_2 \\
\quad |^{\mathrm{r}} \;:\; x \approx^{\in} y_2 \to x \approx^{\in} y_1 \mid y_2 \\
\quad \mathsf{later} \;:\; \infty\ (\flat\ x \approx^{\in} \flat\ y) \to \mathsf{later}\ x \approx^{\in} \mathsf{later}\ y \\
\quad \mathsf{later}^{\mathrm{l}} \;:\; \flat\ x \approx^{\in} y \to \mathsf{later}\ x \approx^{\in} y \\
\quad \mathsf{later}^{\mathrm{r}} \;:\; x \approx^{\in} \flat\ y \to x \approx^{\in} \mathsf{later}\ y
\end{array}$$

You can read $x \approx^{\in} y$ as "x implements one of the allowed semantics of y".

Compiler correctness can now be stated as follows:

$$\begin{array}{l}
\mathit{correct} \;:\; (t : \mathit{Tm}\ 0) \to \\
\qquad \mathit{exec}\ \langle\ \mathit{comp}\ t\ [\,], [\,], [\,]\ \rangle \approx^{\in} \\
\qquad\quad [\![\ t\ ]\!]\ [\,] \gg\!= \lambda\ v \to \mathsf{return}\ (\mathit{comp}_{\mathrm{v}}\ v)
\end{array}$$

If we extend the compiler in the following way, then we can prove that it is correct using an argument which is very similar to that in Section 7:

$$\mathit{comp}\ (t_1 \mid t_2)\ c = \mathit{comp}\ t_1\ c$$

We can also prove type soundness for the non-deterministic language, using the type system from Section 4 extended with the following rule:

$$\_|\_ \;:\; \Gamma \vdash t_1 \in \sigma \;\to\; \Gamma \vdash t_2 \in \sigma \;\to\; \Gamma \vdash t_1 \mid t_2 \in \sigma$$

Type soundness can be stated using $\_\approx^{\in}\_$. Type-correct terms should not crash, no matter how the non-determinism is resolved:

$$\begin{array}{l}
\mathit{type\text{-}soundness} \;:\; [\,] \vdash t \in \sigma \;\to \\
\qquad\qquad \neg\ (\mathsf{now}\ \mathsf{nothing} \approx^{\in} [\![\ t\ ]\!]\ [\,])
\end{array}$$

It is easy to prove this statement by adapting the proof from Section 4. All it takes is to extend the *Lift* type with the constructor

$$\_|\_ \;:\; \mathit{Lift}\ P\ x \;\to\; \mathit{Lift}\ P\ y \;\to\; \mathit{Lift}\ P\ (x \mid y),$$

and then propagating this change through the rest of the proof. Note that the new definition of *Lift* uses induction nested inside coinduction (as do D and $\_\approx^{\in}\_$).

9. Term Equivalences

Let us now return to the deterministic language from Section 3. Weak bisimilarity as defined in Section 2 is, despite its name, a very strong notion of equality for the semantic domain $(\mathit{Maybe}\ \mathit{Value})_\perp$. We can lift this equality to closed terms in the following way:

$$\begin{array}{l}
\equiv \;:\; \mathit{Tm}\ 0 \to \mathit{Tm}\ 0 \to \mathit{Set} \\
t_1 \equiv t_2 = [\![\ t_1\ ]\!]\ [\,] \approx [\![\ t_2\ ]\!]\ [\,]
\end{array}$$

This is a very syntactic equality, which distinguishes the observationally equivalent terms $t_1 = \mathsf{lam}\ (\mathsf{lam}\ (\mathsf{var}\ 0)) \cdot \mathsf{con}\ 0$ and $t_2 = \mathsf{lam}\ (\mathsf{var}\ 0)$, because

$$\begin{array}{ll}
[\![\ t_1\ ]\!]\ [\,] & \approx \\
\mathit{return}\ (\mathsf{lam}\ (\mathsf{var}\ 0)\ (\mathsf{con}\ 0 :: [\,])) & \not\approx \\
\mathit{return}\ (\mathsf{lam}\ (\mathsf{var}\ 0)\quad [\,]) & \approx \\
[\![\ t_2\ ]\!]\ [\,]. &
\end{array}$$

The relational big-step semantics from Section 5 is no different: $[\,] \vdash t_1 \Downarrow v$ does not imply that we have $[\,] \vdash t_2 \Downarrow v$.

This section defines some less syntactical term equivalences. Discussion of the finer points of these equivalences is out of scope for this paper; the main point is that they can be defined without too much fuss.

Let us start by defining a notion of applicative bisimilarity (Abramsky 1990). Computations are equivalent ($\_\approx_\bot\_$) if they are weakly bisimilar, with equivalent (rather than equal) possibly exceptional values; possibly exceptional values are equivalent ($\_\approx_{\mathrm{MV}}\_$) if they are of the same kind and, in the case of success, contain equivalent values; and values are equivalent ($\_\approx_{\mathrm{V}}\_$) if they are either equal constants, or closures which are equivalent when evaluated with the free variables bound to an arbitrary value:[7]

```
mutual
  data _≈⊥_ :
         (Maybe Value)⊥ → (Maybe Value)⊥ → Set where
    now    :       u ≈MV v → now u ≈⊥ now v
    later  : ∞ (♭ x ≈⊥ ♭ y) → later x ≈⊥ later y
    later¹ :       ♭ x ≈⊥ y → later x ≈⊥ y
    later² :       x ≈⊥ ♭ y → x ≈⊥ later y

  data _≈MV_ : Maybe Value → Maybe Value → Set where
    just    : u ≈V v → just u ≈MV just v
    nothing :          nothing ≈MV nothing

  data _≈V_ : Value → Value → Set where
    con : con i ≈V con i
    lam : (∀ v → ∞ (⟦ t₁ ⟧ (v :: ρ₁) ≈⊥ ⟦ t₂ ⟧ (v :: ρ₂))) →
          lam t₁ ρ₁ ≈V lam t₂ ρ₂
```

This is yet again a definition which uses induction nested inside coinduction. Note that the lam constructor is coinductive. If this constructor were inductive, then the relations would not be reflexive: lam (var zero) [ ] would be provably distinct from itself.

Using the relations above we can define applicative bisimilarity by stating that terms are equivalent if they are equivalent when evaluated in an arbitrary context:

```
_≈T_ : Tm n → Tm n → Set
t₁ ≈T t₂ = ∀ ρ → ⟦ t₁ ⟧ ρ ≈⊥ ⟦ t₂ ⟧ ρ
```

The definition of $\_\approx_\bot\_$ is very similar to the definition of weak bisimilarity in Section 2. It is possible to define a single notion of weak bisimilarity, parametrised by a relation to use for values. The accompanying code uses such a definition.

Let us now turn to contextual equivalence. Contexts with zero or more holes can be defined as follows:

```
data Context (m : ℕ) : ℕ → Set where
  hole :                          Context m m
  con  : ℕ                      → Context m n
  var  : Fin n                  → Context m n
  lam  : Context m (1 + n)      → Context m n
  _·_  : Context m n → Context m n → Context m n
```

The type *Context m n* contains contexts whose holes expect terms of type *Tm m*. If we fill the holes, then we get a term of type *Tm n*:

```
_[_] : Context m n → Tm m → Tm n
hole    [t] = t
con i   [t] = con i
var x   [t] = var x
lam C   [t] = lam (C [t])
(C₁ · C₂) [t] = C₁ [t] · C₂ [t]
```

Contextual equivalence can be defined in two equivalent ways. The usual one states that t_1 and t_2 are contextually equivalent if $C\,[t_1\,]$ terminates iff $C\,[t_2\,]$ terminates, for any closing context C:

```
_⇓ : A⊥ → Set
x ⇓ = ∃ λ v → x ≈ now v
```

```
_≈C_ : Tm n → Tm n → Set
t₁ ≈C t₂ = ∀ C → ⟦ C [t₁] ⟧ [] ⇓ ⇔ ⟦ C [t₂] ⟧ [] ⇓
```

However, we can also define contextual equivalence using weak bisimilarity:

```
_≈'C_ : Tm n → Tm n → Set
t₁ ≈'C t₂ = ∀ C → ⟦ C [t₁] ⟧ [] ≈° ⟦ C [t₂] ⟧ []
```

Here $\_\approx^\circ\_$ is a notion of weak bisimilarity which identifies all terminating computations:

```
now : now u ≈° now v
```

It is easy to prove that these two notions of contextual equivalence are equivalent.

As an aside one can note that the contextual equivalences above are a bit strange, because there is no context which distinguishes con 0 from con 1. This could be fixed by extending the language with suitable constructions for observing the difference between distinct constants.

10. Conclusions

When writing down a semantics I think one of the main priorities should be to make it easy to understand. Sometimes a more complicated definition may be more convenient for certain tasks, but in that case one can define two semantics and prove that they are equivalent.

I hope I have convinced you that functional operational semantics defined using the partiality monad are easy to understand. I have also used two such semantics to state a compiler correctness result, and I find this statement to be easier to understand than a corresponding statement phrased using relational semantics (see Section 7).

The semantics also seem to be useful when it comes to proving typical meta-theoretic properties, at least for the simple languages discussed in this paper. I have proved type soundness and compiler correctness directly for the semantics given above. The type soundness proof in Section 4 is given in relatively complete, formal detail, yet it is short and should be easy to follow. Furthermore, as mentioned in Section 7, the compiler correctness proof avoids some duplication which is present in a corresponding proof for relational semantics.

As discussed above the support for total corecursion in languages like Agda and Coq is somewhat limited: definitions like $[\![\_]\!]$ are often rejected. However, my experience with sized types in MiniAgda (see Section 2) is encouraging. I suspect that a more polished implementation of sized types could be quite satisfying to work with.

Finally I want to mention a drawback of this kind of semantics: proofs which proceed by induction on the structure of $\_\vdash\_\Downarrow\_$ when a relational big-step semantics is used can become somewhat awkward when transferred to this setting, as illustrated by the proof in Section 5 showing that $[\![\, t \,]\!]\,\rho \approx \mathit{return}\, v$ implies $\rho \vdash t \Downarrow v$. However, it is unclear to me how often this is actually a problem. For instance, neither the type soundness proofs nor the compiler correctness proofs in this paper are affected by this drawback.

Acknowledgements

I want to thank Thorsten Altenkirch for encouraging this line of work, Peter Dybjer for useful feedback on a draft of the paper, and Tarmo Uustalu for pointing out some related work. I would also like to thank the anonymous reviewers for lots of useful feedback.

Large parts of this work were done when I was working at the University of Nottingham, with financial support from EPSRC (grant code: EP/E04350X/1). I have also received support from the

[7] $\forall\, v \to \ldots$ means the same as $(v : \_) \to \ldots$; Agda tries to infer the value of the underscore automatically.

ERC: “The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° 247219.”

References

Andreas Abel. MiniAgda: Integrating sized and dependent types. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR 2010)*, volume 43 of *EPTCS*, 2010. doi:10.4204/EPTCS.43.2.

Samson Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*. Addison-Wesley, 1990.

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2012.

Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. Short note supporting a talk given at the Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR 2010), 2010.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *LNCS*, pages 50–65, 2005. doi:10.1007/11541868_4.

Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP’09, Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming*, pages 97–107, 2009. doi:10.1145/1596550.1596567.

Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 115–130, 2009. doi:10.1007/978-3-642-03359-9_10.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–28, 2005. doi:10.2168/LMCS-1(2:1)2005.

The Coq Development Team. *The Coq Proof Assistant, Reference Manual, Version 8.3pl3*, 2011.

Thierry Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, International Workshop TYPES ’93*, volume 806 of *LNCS*, pages 62–78, 1994. doi:10.1007/3-540-58085-9_72.

Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In *POPL ’92, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–94, 1992. doi:10.1145/143165.143184.

Patrick Cousot and Radhia Cousot. Bi-inductive structural semantics. *Information and Computation*, 207(2):258–283, 2009. doi:10.1016/j.ic.2008.03.025.

Nils Anders Danielsson. Beating the productivity checker using embedded languages. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR 2010)*, volume 43 of *EPTCS*, pages 29–48, 2010. doi:10.4204/EPTCS.43.3.

Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively: An exercise in mixed induction and coinduction. In *Mathematics of Program Construction, 10th International Conference, MPC 2010*, volume 6120 of *LNCS*, pages 100–118, 2010. doi:10.1007/978-3-642-13321-3_8.

Neil Ghani and Tarmo Uustalu. Monad combinators, non-determinism and probabilistic choice. Extended abstract distributed at the workshop on Categorical Methods in Concurrency, Interaction and Mobility (CMCIM 2004), 2004.

Sergey Goncharov and Lutz Schröder. A coinductive calculus for asynchronous side-effecting processes. In *Fundamentals of Computation Theory, 18th International Symposium, FCT 2011*, volume 6914 of *LNCS*, pages 276–287, 2011. doi:10.1007/978-3-642-22953-4_24.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52:107–115, 2009. doi:10.1145/1538788.1538814.

Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.

Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991. doi:10.1016/0304-3975(91)90033-X.

Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Lab. for Found. of Comp. Sci., University of Edinburgh, 1990.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While: Big-step and small-step, relational and functional styles. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 375–390, 2009. doi:10.1007/978-3-642-03359-9_26.

Keiko Nakata and Tarmo Uustalu. Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: An exercise in mixed induction-coinduction. In *Proceedings Seventh Workshop on Structural Operational Semantics (SOS 2010)*, volume 32 of *EPTCS*, pages 57–75, 2010. doi:10.4204/EPTCS.32.5.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 383–413. Cambridge University Press, 2009.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM ’72, Proceedings of the ACM annual conference*, volume 2, pages 717–740, 1972. doi:10.1145/800194.805852.

J.J.M.M. Rutten. A note on coinduction and weak bisimilarity for while programs. *Theoretical Informatics and Applications*, 33:393–400, 1999. doi:10.1051/ita:1999125.

Davide Sangiorgi and Robin Milner. The problem of “weak bisimulation up to”. In *CONCUR ’92, Third International Conference on Concurrency Theory*, volume 630 of *LNCS*, pages 32–46, 1992. doi:10.1007/BFb0084781.

Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990. doi:10.1016/0890-5401(90)90018-D.

Philip Wadler. The essence of functional programming. In *POPL ’92, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992. doi:10.1145/143165.143169.