Programming Language Semantics

Gavin Mendel-Gleason

Data Chemist Ltd.

gavin@datachemist.com

Abstract

This paper comprises a set of notes for a short course on programming language semantics using Agda [?]. We cover the basics of operational and denotational semantics, with some example proofs and proof exercises.

Most other proof assistants / programming languages would work as a replacement, and the diligent student should be able to translate them into their language of choice with little difficulty, aside from some subtleties with coinduction in a very limited number of proofs.

2012 ACM Subject Classification Program semantics

Keywords and phrases denotational semantics, operational semantics, type theory

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Programming language semantics is the study of the meaning of programs. A programming language can be thought of as some syntactic means of describing what computation we would like to perform, coupled with a mechanism for performing the computation according to what this language describes.

We are going to take a quick tour of programming language semantics, making use of the Agda proof assistant as a method of both implementing, and proving properties of programming languages.

The advantages in clarity of using a formal proof assistant which is itself a programming language as the approach to describing semantics are many. It's often much clearer what precisely is being described in language semantics when we can actually see the datastructures, proofs and evaluation itself described in a well structured metalanguage. We can steal the computation of our host system to clarify meaning as well as give succinct descriptions of what precisely we are trying to prove by means of *types*.

Prior familiarity with type theory is not required to understand this course, though it would be useful if the student is somewhat fluent with both functional programming in a language such as haskell, ML or similar, as well as some familiarity with imperative programming paradigms.

Type theory is a discipline in mathematics which gives a formal approach to the statement of conjectures and evidence for their satisfaction. It is also, however, possible using something known as the Curry-Howard correspondence, to view the evidence of satisfaction, or proofs, as a functional programming language. This is exploited to great effect in so-called 'Dependently typed languages' such as Agda.

We will see how Agda can be used to describe programming language semantics using "Dependent types" for programming languages which we implement in agda.

If all of this is clear as mud: fear not. The detail may in fact dispel the devil.

2 Preliminaries

Instead of thinking of type theory through a very general mathematical lens, we're going to take a more prosaic approach. We will treat it as merely a sophisticated functional programming language.

We start from the ground up, building some basic machinery ourselves in order to understand how it works. Later we will use the analogous machinery from the Agda standard library.

Our exploration begins by defining a number of *inductive types*. Inductive types are the primary manner in which we describe our data structures for our semantics. They will also prove useful for defining properties and relations about the terms in the programming languages we will define. Inductive types are essentially abstract data types which are potentially recursive.

Inductive types come equiped automatically with mechanisms to introduce elements of them, known as *constructors* and ways of eliminating them, using pattern matching in a manner which should be familiar users of ML and Haskell.

The simplest inductive type is the empty type, which we call "bottom" or ⊥.

```
data ⊥ : Set where

explosion : ∀ {A : Set} → ⊥ → A
explosion ()

¬_ : ∀ (A : Set) → Set
¬ A = A → ⊥
```

This type has very little to say. You can't produce one so it has no constructors, and if you find one, you can't pattern match on it because it could not have been introduced in the first place.

We can see how this is used with our term `explosion` which is the proof of the *principle of explosion*. First, the brackets, { } around the `A` say that the argument is *implicit* and we are asking Agda to try and infer it from context. This means that the first argument we pattern match on is the terms of type ⊥. However, since there are none to match, we don't even have to finish our description of what to do if we find an occurence and so need not find a way to produce an element of the type A. From false premises, anything follows.

In addition we can describe a kind of negation with `¬_` which allows us to say that our premise can not hold, because otherwise we would be able to produce a datatype with no elements. This will prove a powerful, if somewhat non-intuitive way of describe that certain properties do not hold.

The next simplest type is the type called *top* or ⊤. Sometimes this type is called *one* (and the type ⊥ is called *zero*) since it only has the one constructor: `tt`.

```
data ⊤ : Set where
  tt : ⊤
```

Next we see the more familar type of booleans. The boolean type has two constructors, `true` and `false`.

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

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

_and_ : Bool → Bool → Bool
true and true = true
true and false = false
false and b = false

_or_ : Bool → Bool → Bool
true or b = true
false or true = true
false or false = false

if_then_else_ : ∀ {C : Set} → Bool → C → C → C
if true then x else y = x
if false then x else y = y
```

With the bool type defined, we can define the not function, which maps booleans to booleans, but flips the constructor.

Similarly we can define the usual _and_ and _or_ which allow us to combine booleans in the usual way. Here the underscores tell Agda that we mean for the function to be infix, so its first argument is on the left, and its second is on the right.

However, we can also describe mixfix functions in Agda, as we see with the if_then_else_ function, which takes a boolean as its first argument, and then takes two branches of type C which the boolean is used to select from. We can then recover the quite familiar form of conditional branching as a function.

For a more in-depth description of the Agda programming language of dependent types and its use as a proof-assistant, see Ulf Norell's work *Dependently Typed Programming in Agda*[?] and Aaron Stump's *Verified Functional Programming in Agda*[?]. We will press on hoping that the reader can pick things up as we go along, and, if not, looks to these additional resources in the event of confusion.

3 Operational Semantics

Now that we've seen some basic uses of inductive types and functions which manipulate them, we are ready to do some heavier lifting in the service of programming language semantics.

operational semantics is an approach to semantics which looks at how, operationally, our programming terms *compute*. we'll get into the nitty-gritty of evaluation, and learn how to state general properties of terms by seeing how evaluation progresses.

```
module operationalsemantics where
  open import Data.Nat

  data Exp : Set where
    num : ℕ → Exp
    _⊕_ : Exp → Exp → Exp
```

We define a new *module* which we name OperationalSemantics. In Agda, a module is essentially a record that allows us to define parameters, control our namespace and produce a collection of associated terms which refer to eachother. In this case we have a very simple module with no parameters. Directly below the introduction of our module, and within its scope, we import another module, Data.Nat which contains the definitions for the natural numbers, and various functions to manipulate them and facts about them.

We can then define the syntax of our first language: Exp. This language is composed of numbers, and addition. The syntax for introducting a number uses the constructor num on a natural number, the definition of which we've imported from Data.Nat. We can then add two expressions using the infix constructor _⊗_.

So what does an expression look like? The expression twelve shows how to write the addition of 3 numbers in our syntax.

```
twelve : Exp
twelve = num 3 ⊕ (num 4 ⊕ num 5)
```

Our syntax is, however, merely a data-structure. It doesn't *do* anything. We must add *dynamics* to our syntax in order to get something worthy of being called a language.

We can do this by defining what it would mean to *evaluate* an expression. We will do this by defining an *evaluation relation*.

3.1 Big-Step Operational Semantics

The first evaluation relation we will define, is something called a *big-step evaluation relation*. It's called a big-step relation because we define a relation between a term in our syntax, with the final stage of evaluation is when something has reached a *value*. In this case our values are numbers.

This provides a contrast with *small-step evaluation relations* that instead tell you what the next step of a computation is. We will look at small-step evaluation relations later, and then explore how they relate.

```
infix 10 _⇓_
data _⇓_ : Exp → ℕ → Set where
  n⇓n : ∀ {n} →

    ------------------
    num n ⇓ n

  E⊕E : ∀ {E₁ E₂ n₁ n₂} →

    E₁ ⇓ n₁ → E₂ ⇓ n₂ →
    ------------------------------
      E₁ ⊕ E₂ ⇓ (n₁ + n₂)
```

We introduce a new, infix, inductive type which we name _⇓_. We will call this a relation, because it has two parameters, an expression, drawn from Exp and a natural number drawn from ℕ.

All other types we've seen were defined to be themself of type Set directly. We can think of this as a family of types, with each member of the family drawn by chosing elements of Exp and ℕ, or as is natural for our current problem, as a *relation* between Exp and ℕ.

Our relation has two constructors. The first is called n⇓n. It basically says, that the big step evaluation relation relates an expression formed of a natural number, to that same natural number. In other words, when you evaluate a natural number, nothing happens.

We use a number of dashes, -- to simulate the horizontal bar sometimes used in proofs, to separate our premises, from our conclusions. In agda more than two dashes is simply a comment, so this is just a visual aid and has no impact on the terms we are expressing.

In our first case, we have no premises (aside from the uninteresting implicit n).

The second constructor is E⊕E. We can read this as stating that, given we know the number to which two expressions E_1 and E_2 evaluate, we can then determine that the expressions conjoined with _⊕_ will evaluate to the sum of the numbers to which each expression evaluates, respectively.

```
evalBig : ∀ E → Σ[ n ∈ ℕ ] E ⇓ n
evalBig (num x) = x , n⇓n
evalBig (e ⊕ e₁) with evalBig e | evalBig e₁
evalBig (e ⊕ e₁) | n , proof_n | m , proof_m = n + m , E⊕E proof_n proof_m
```

With this bigstep evaluation relation in hand, we can write an evaluation function, evalBig, which demonstrates that given any expression in our syntax defined above, we can obtain a natural number which relates the expression to its final evaluated form.

In Agda, we do this using the Σ record. This is a pair datastructure, comprising an element, and a proof of the type $E \Downarrow n$ which depends on both n and E. In this case the element is some natural number ℕ and the proof we must construct is the full demonstration that E evaluates to that number under the big step evaluation relations.

In English we might read the type of evalBig as stating that:

> "Given any expression, evalBig will give us the number this expression evaluates to, and a proof that this is a value which is related by the big-step evaluation relation."

So how does evalBig work? It is essentially a simple proof by induction on the structure of the syntax. With the syntax of our language we have two cases. If we are the base case, namely (num x) we can use the fact that numbers evaluate to themselves. This expression has no sub-expressions, and so we are done, which is what makes it a base case.

If we are a sum of two expressions, combined with the _⊕_ constructor, we first evaluate each sub expression independently, and then sum their values, using the E⊕E constructor to combine the proofs obtained in both branches to produce the *next step* of the proof.

With evalBig in hand, we can look back at twelve and see the effect of evaluating this expression.

```
example⇓ : num 3 ⊕ (num 4 ⊕ num 5) ⇓ 12
example⇓ = proj₂ (evalBig (num 3 ⊕ (num 4 ⊕ num 5)))
```

We apply evalBig to our expression, and then use proj_2 to project out the *proof* from our pair of the number 12 and the proof which it is coupled with.

This example provides a sort of template for the way in which we can build up operational semantics using big-step evaluation. Using this general approach as a guide, we can look at more interesting languages, and ask more sophisticated questions and get proofs about our semantics.

However, for now we will continue on with this simple language, and look at another approach to operational semantics.

3.2 Small-step operational semantics

In our previous example we demonstrated our reduction relation by relating expressions to the values which they must evaluate to. There is another option, namely, that we describe our reduction in terms of *the next step*.

We can define a new type, _⟶_, which relates two expressions when the expression on the left can transition to the expression on the right in *one step*.

```
infix 8 _⟶_
data _⟶_ : Exp → Exp → Set where
  +⟶ : ∀ {n m} →

                 -----------------------------
                 num n ⊕ num m ⟶ num (n + m)

  ⊕₁⟶ : ∀ {E₁ E₁' E₂} →

                 E₁ ⟶ E₁' →
                 ----------------------
                 E₁ ⊕ E₂ ⟶ E₁' ⊕ E₂

  ⊕₂⟶ : ∀ {n E₂ E₂'} →

                 E₂ ⟶ E₂' →
                 --------------------------
                 num n ⊕ E₂ ⟶ num n ⊕ E₂'
```

This time we have three rules with one base case and a left and right rule. The base case, +⟶ takes two values combined with _⊕_ and evaluates to a value by summing the two values.

The left rule, $\oplus_1$⟶, relates an expression with a sum in which the left-hand summand can be further related by a small step, but whose right-hand summand remains unchanged.

The right rule, $\oplus_2$⟶, relates an expression with a sum in which the left-hand is already fully evaluated to a number, but whose right-hand can still be related by a small step.

To get a feel for whats going on, we can look at how we can relate expressions concretely using this relation.

```
example⟶₁ : (num 3 ⊕ num 7) ⊕ (num 8 ⊕ num 1) ⟶ num 10 ⊕ (num 8 ⊕ num 1)
example⟶₁ = ⊕₁⟶ +⟶
example⟶₂ : (num 10) ⊕ (num 8 ⊕ num 1) ⟶ num 10 ⊕ num 9
example⟶₂ = ⊕₂⟶ +⟶
```

In example⟶₁ the left hand side has summed exactly one pair of numbers. This is done by descending into the left branch and then summing the pair of summands.

In example⟶₂ we have already reduced the left-hand side to a number, we can then use the ⊕₂⟶ to decend into the right branch where we find a pair of numbers which can be summed.

The reader may have noticed that these rules are written very carefully such that there is no choice in the application of rules. You have to evaluate all left expressions first, before one is allowed to proceed with the right branch.

This choice is not necessary, but describes a *deterministic* relation. Instead of this we could choose an alternative approach which allows a choice of summand in which to descend.

```
infix 8 _⟶ch_
data _⟶ch_ : Exp → Exp → Set where
  +⟶ch : ∀ {n m} →

            -------------------------------
            num n ⊕ num m ⟶ch num (n + m)

  ⊕₁⟶ch : ∀ {E₁ E₁' E₂} →

            E₁ ⟶ch E₁' →
            ----------------------
            E₁ ⊕ E₂ ⟶ch E₁' ⊕ E₂

  ⊕₂⟶ch : ∀ {E₁ E₂ E₂'} →

            E₂ ⟶ch E₂' →
            ----------------------
            E₁ ⊕ E₂ ⟶ch E₁ ⊕ E₂'
```

We have a new relation, _⟶ch_, where the *ch* is short for 'choice'. Here we have the same approach to taking two summands which are themselves values and combining it to a single value. However for the left and right rules, we are not constrained in which branch we choose. We can choose to make our step in either the right branch or the left branch.

```
example⟶ch₁ : (num 3 ⊕ num 7) ⊕ (num 8 ⊕ num 1) ⟶ch num 10 ⊕ (num 8 ⊕ num 1)
example⟶ch₁ = ⊕₁⟶ch +⟶ch
example⟶ch₂ : (num 3 ⊕ num 7) ⊕ (num 8 ⊕ num 1) ⟶ch (num 3 ⊕ num 7) ⊕ num 9
example⟶ch₂ = ⊕₂⟶ch +⟶ch
```

We can now see two different examples where we make a choice of branch in which to step. In example⟶ch₁ we chose the left branch for evaluation, summing 3 and 7 and producing 10. In example⟶ch₂ we take the right branch, summing 8 and 1 to produce 9.

Now that we have defined these two different approaches, we might ask, are we performing the same evaluation?

```
⟶⇒⟶ch : ∀ {E₁ E₂} → (E₁ ⟶ E₂) → (E₁ ⟶ch E₂)
⟶⇒⟶ch +⟶ = +⟶ch
⟶⇒⟶ch (⊕₁⟶ d) = ⊕₁⟶ch (⟶⇒⟶ch d)
⟶⇒⟶ch (⊕₂⟶ d) = ⊕₂⟶ch (⟶⇒⟶ch d)
```

We can see from the theorem ⟶⇒⟶ch that we can always mimic the deterministic small step evaluation relation _⟶_ with the _⟶ch_ by simply always choosing the semetric proof rule.

However, the reverse theorem which transforms a choice small step to the deterministic one is not true. To prove this, we make use of Agda's Data.Empty and the Relation.Nullary libraries, rather than our own negation which we described in the introduction.

```
open import Data.Empty
open import Relation.Nullary

¬⟶ch⇒⟶ : ¬ (∀ E₁ E₂ → (E₁ ⟶ch E₂) → (E₁ ⟶ E₂))
¬⟶ch⇒⟶ f with f ((num 0 ⊕ num 0) ⊕ (num 0 ⊕ num 0)) ((num 0 ⊕ num 0) ⊕ num 0)
                    (⊕₂⟶ch +⟶ch)
¬⟶ch⇒⟶ f | ()
```

To prove that we can not always perform the same small step reduction we need merely to find a single example for which does not hold.

Since ¬_ is essentially short hand for a function which maps from a type to the empty type, our first argument, f has the type of the theorem

$$\forall E_1 E_2 \rightarrow (E_1 \longrightarrow\!\mathsf{ch}\ E_2) \rightarrow (E_1 \longrightarrow E_2)$$

... which we know can not be true.

With our proof in hand, we can apply it to a patently impossible reduction. Agda cleverly detects that we have no cases in which this can hold and we are able to eliminate the possibility of our argument, which eliminates the need to supply an impossible proof of ⊥.

We have claimed earlier that our small step evaluation realtion is deterministic, but we have not yet proved it. In order to prove determinism, it is convenient first to have a notion of equivalence. In Agda we can do this by means of a propositional equality type $\_\equiv_p\_$.

We will define our own here in order to get a flavour. Later we will use Agda's libraries to handle this machinery for us.

```
infix 4 _≡p_
data _≡p_ {A : Set} (x : A) : A → Set where
  instance reflp : x ≡p x
```

The equality type can be described as follows. Given any type and a value of that type, we can show that the value is equivalent to itself, using the refl_p constructor. This might seem quite useless, as we can only show things are equal to themselves! However, it is more flexible than it first appears, because Agda is able to perform computations. So in order to show two things are equal, we will be able to make the host system perform some computation and we can build up non-trivial equivalences in this way.

Now that we have the basic idea of equivalence, we can switch to Agda's built in propositional equality _≡_ and its single constructor refl.

```
open import Relation.Binary.PropositionalEquality

⟶deterministic : ∀ {E E₁ E₂} → E ⟶ E₁ → E ⟶ E₂ → E₁ ≡ E₂
⟶deterministic +⟶ +⟶ = refl
⟶deterministic +⟶ (⊕₁⟶ ())
⟶deterministic +⟶ (⊕₂⟶ ())
⟶deterministic (⊕₁⟶ ()) +⟶
⟶deterministic (⊕₁⟶ p) (⊕₁⟶ q) = cong₂ _⊕_ (⟶deterministic p q) refl
⟶deterministic (⊕₁⟶ ()) (⊕₂⟶ q)
⟶deterministic (⊕₂⟶ ()) +⟶
⟶deterministic (⊕₂⟶ p) (⊕₁⟶ ())
⟶deterministic (⊕₂⟶ p) (⊕₂⟶ q) = cong₂ _⊕_ refl ((⟶deterministic p q))
```

That our small step evalution relation is deterministic is described by the type of ⟶deterministic. The theorem we are trying to prove, is that given any term E in our language, and a proof that this is related by a single step to a term E_1, then if it is also related by a single step to some other term E_2 then E_1 and E_2 must actually be the same term.

The proof of the theorem proceeds by induction on the proof of both reduction relations using a simultaneous case match. The base case of both is trivial, giving us that the sum of the numbers n and m which are implicit variables of $+\!\longrightarrow$ are equal to themselves which is easily proved with refl. Though we can not see these numbers in the proof as written here, replacing refl with a *hole* by compiling Agda with a '?' will allow you to see the implicit variables in the context.

Six more of the cases are eliminated by Agda automatically and can be replaced with a vacuous match. As an example, in the first case in which we have a vacuous match, the argument to $\oplus_1\!\longrightarrow$ would have to have type: num n $\longrightarrow E_1$'. Case analysis on this yields no ways in which to form this type as num n can not be the left-hand side of any reduction.

In order to deal with the recursive case, where we have like constructors, we can make use of conf_2. This function merely uses the fact that if two arguments are equal, then functions of those arguments are also equal.