Handout A: Introduction to Natural Semantics

Christoph Reichenbach

November 6, 2019

Formal semantics allow us to describe the meaning of programs precisely and concisely. In this course we only discuss *Natural Semantics*, also known as *Big-Step Operational Semantics*. These semantics are closely related to the concept of programming language implementation by *interpretation*.

Natural semantics assume that we know the syntax of our language. In the following, we assume that we have a BNF specification of the grammar, such as the one in Figure 1. This language, DivAdd, is syntactically ambiguous (e.g., there are two different ways to parse `1 + 2 div 3`). This syntactic ambiguity is an important concern for the design of the grammar, but for the definition of semantics (i.e., for this handout), we assume that any ambiguities has been resolved (e.g., by assigning precedence to the operators). We don't discuss the relevant syntactic mechanisms here; instead we will use parentheses to disambiguate as needed, e.g., `1 + (2 div 3)` (even though the BNF may not make such parentheses explicit).

1 A Meta-Language for Describing Semantics

Natural semantics is a meta-language (cf. Sebesta, Section 3.3.1.3) for describing the behaviour of an object language. The object language here is the programming language that we want to describe (e.g., DivAdd).

We will now use the English language as a meta-language to describe the behaviour of Natural Semantics, as an object language, so there are three layers to our discussion.

Natural semantics relates a program to the result that the program computes. For simplicity, we begin with programs that compute numbers; later we will study programs that update variables and sketch how we can read input and print output.

The key idea behind natural semantics is the evaluation relation ($\Downarrow$), which states the relation between a program p and the program's result v:

$$p \Downarrow v$$

We read this as 'p evaluates to v'.

$$\begin{array}{rcl} \langle expr \rangle & \longrightarrow & nat \\ & | & \langle expr \rangle \texttt{+} \langle expr \rangle \\ & | & \langle expr \rangle \texttt{div} \langle expr \rangle \end{array}$$

Figure 1: Syntax of our language DivAdd. Here *nat* is a terminal symbol that can be a textual representation of any natural number.

Here, p and v are *Metavariables*, that is, variables in the meta-language. p is a variable that can contain any input program, and v is a variable that can contain any result that the language might compute.

Let us begin with an (exceedingly simple) programming language:

$$\langle W \rangle \longrightarrow \texttt{ett} \mid \texttt{två} \mid \texttt{tre}$$

This language allows only three programs, each of which consists of a single word in the Swedish language. We can now define its semantics by the following rules:

$$\texttt{ett} \Downarrow 1 \qquad \texttt{två} \Downarrow 2 \qquad \texttt{tre} \Downarrow 3$$

What we have now done is to list all the different programs in the language and explain what this program should mean, connecting program and output by the $\Downarrow$ symbol.

We see that the evaluation relation maps input programs in the object language W to natural numbers in $\mathbb{N}$.

In general, the terms to the left of $\Downarrow$ are terms in the object language, i.e., the language we are trying to describe. The terms on the right-hand side of $\Downarrow$ are in some 'result language' that our semantics computes. These can be any mathematical objects. In some cases the object language and the output language can even overlap. For example, in a pocket calculator language, the output might be in the set of rational numbers, but rational numbers might also be part of the input language. That is, both the object language and the meta-language may make use of a shared concept (such as the rational numbers).

1.1 Ambiguity

Note that $\Downarrow$ is, in general, a *relation* and not a function. We can see what that means in the following language:

$$\langle Q \rangle \longrightarrow \texttt{eins} \mid \texttt{zwei} \mid \texttt{?}$$

With the semantics given below:

$$\texttt{eins} \Downarrow 1 \qquad \texttt{zwei} \Downarrow 2 \qquad \texttt{?} \Downarrow 0 \qquad \texttt{?} \Downarrow 3$$

Here, we have two specifications for the `?` symbol: one that evaluates it to 0 and one that evaluates it to 3. This means that the semantics is *ambiguous*.

While we could use ambiguous semantics to model nondeterminism, we generally want to avoid such ambiguity. When formally defining programming languages, language designers take great care to avoid such ambiguity or sometimes develop proofs to show that the language 'is confluent' (i.e., different hypothetical paths of computation 'flow together' at the end). In this course we require that semantics are not ambiguous, and we can usually ensure this by carefully checking that left-hand side of one $\Downarrow$ in our semantics description doesn't overlap with the left-hand side of another $\Downarrow$ — or, if it does, that these cases will yield the same right-hand side.

1.2 Conditional Rules

Programming languages with a finite number of input programs aren't usually very interesting. Let us instead consider a language that allows an infinite number of input programs, such as DivAdd, given in Figure 1.

As in our previous examples, let us try to define the semantics of each of the different branches of our grammar separately.

First, consider input programs that have the form *nat*. That is, they are a string of digits of a form that represents a natural number, e.g., `0` or `1` or `65536`.

If we were writing a language implementation, we would now have to translate this character string into a natural number. For instance, we can imagine a function `asNat` that takes in a character string and translates it into a natural number (similar to `atoi()` in C or `int()` in Python or `Integer.parseInt()` in Java):

$$n \Downarrow \mathsf{asNat}(n) \qquad \text{but only if } n \in \mathit{nat}$$

Here again n is a *Metavariable*, i.e., a variable in our meta-language, Natural Semantics, as opposed to a program variable in DivAdd. We don't have any program variables in DivAdd, so there is no ambiguity as to what kind of variable it is.

Since we are only describing the case where n is a string of digits (and not anything that contains a `+` or `div` symbol), our rule above adds the requirement that $n \in \mathit{nat}$, in other words, that n must be an element of the set of strings of digits that we've called *nat*.

These kinds of preconditions are very common, so we have a simplified notation for them:

$$\frac{n \in \mathit{nat}}{n \Downarrow \mathsf{asNat}(n)}$$

The meaning of this rule is exactly the same as for our previous rule. This 'inference rule' notation allow us to compactly capture preconditions (or *premises*) on a bar above our semantics specification.

Often, semantics specifications try to avoid excessive formal correctness in favour of simplicity. In these specifications, the writer might omit the `asNat` function by arguing that 'separating between natural numbers and their textual

representation is overly pedantic, as long as there is no ambiguity'. Such an author might write the above rule as follows:

$$\frac{n \in \mathit{nat}}{n \Downarrow n}$$

1.3 Recursion

The next construct in DivAdd is ⟨*expr*⟩`+`⟨*expr*⟩. Intuitively, we want this construct to represent addition. As a first attempt, we might try the following rule[1]:

$$\frac{n_1 \in \mathit{nat} \quad n_2 \in \mathit{nat}}{n_1\texttt{+}n_2 \Downarrow \mathsf{asNat}(n_1) + \mathsf{asNat}(n_2)}$$

This rule allows us to add up two numbers. However, it doesn't capture all the cases that ⟨*expr*⟩`+`⟨*expr*⟩ describes, only the case *nat*`+`*nat*. For example, if we try to evaluate `(1 + 2)+(3 + 4)`, our rule doesn't match, since it will set $n_1 =$ `(1 + 2)`, which will then fail the check $n_1 \in \mathit{nat}$.

Instead, what we would like is to first compute the *result* of `(1 + 2)`, and then use that result, i.e., to *recurse*. Fortunately, nothing prevents us from having the definition of $\Downarrow$ recursively depend on itself:

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1\texttt{+}e_2 \Downarrow n_1 + n_2}$$

Note that we now have four metavariables– e_1 and e_2, which can bind to any ⟨*expr*⟩, and n_1 and n_2, which bind to evaluation results (natural numbers).

1.4 Completeness

Finally, let us define ⟨*expr*⟩ `div` ⟨*expr*⟩, which we intend as division:

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 \texttt{ div } e_2 \Downarrow \lfloor \frac{n_1}{n_2} \rfloor}$$

Here, we borrow the notation $\lfloor \frac{n_1}{n_2} \rfloor$ from arithmetic to mean 'n_1 divided by n_2, rounded down'.

While this rule seems superficially satisfactory, it has one omission: the meaning of the input program `1 div 0` $\Downarrow \ldots$ is undefined, because $\frac{x}{0}$ is undefined in arithmetic.

Such implicit omissions (or more explicit omissions, such as forgetting to write down a rule) can leave the definition of a language to be incomplete.

Some languages (like C and C++) deliberately include incompleteness in their language definitions, arguing that it gives more flexibility to language

[1] Here, '`+`' is a symbol in our object language, whereas '+' is addition at the meta-language level.

$$\frac{n \in nat}{n \Downarrow \mathsf{asNat}(n)}\ (nat) \qquad \frac{e_1 \Downarrow n_1 \quad e_1 \neq \mathsf{Err} \quad e_2 \Downarrow n_2 \quad e_1 \neq \mathsf{Err}}{e_1 \texttt{+} e_2 \Downarrow n_1 + n_2}\ (add')$$

$$\frac{e_1 \Downarrow \mathsf{Err}}{e_1 \texttt{+} e_2 \Downarrow \mathsf{Err}}\ (add\text{-}l\text{-}E) \qquad \frac{e_2 \Downarrow \mathsf{Err}}{e_1 \texttt{+} e_2 \Downarrow \mathsf{Err}}\ (add\text{-}r\text{-}E)$$

$$\frac{e_1 \Downarrow n_1 \quad n_1 \neq \mathsf{Err} \quad e_2 \Downarrow n_2 \quad n_2 \neq 0 \quad n_2 \neq \mathsf{Err}}{e_1 \ \texttt{div}\ e_2 \Downarrow \lfloor \frac{n_1}{n_2} \rfloor}\ (div')$$

$$\frac{e_1 \Downarrow \mathsf{Err}}{e_1 \ \texttt{div}\ e_2 \Downarrow \mathsf{Err}}\ (div\text{-}l\text{-}E) \qquad \frac{e_2 \Downarrow \mathsf{Err}}{e_1 \ \texttt{div}\ e_2 \Downarrow \mathsf{Err}}\ (div\text{-}r\text{-}E) \qquad \frac{e_2 \Downarrow 0}{e_1 \ \texttt{div}\ e_2 \Downarrow \mathsf{Err}}\ (div\text{-}0\text{-}E)$$

Figure 2: Semantics for DivAdd with explicit error handling

implementations, though this style of language definition is no longer popular today.

Most languages instead aim for a *complete* language definition. There are a number of options available to us for DivAdd, the two most popular of which are:

1. **Adding a Meta-Rule**: We can add an informal rule that states that if the semantics of an operation is not defined, then execution aborts with an error. This is a pragmatic escape hatch that allows us to keep specifications compact, but it risks the language designer 'sprinkling' behaviour over corner cases in the language that they hadn't thought about. Moreover, it does not allow for the semantics to describe any forms of error recovery.

   Figure 6 in the appendix gives the full semantics for our language with such a meta-rule.

2. **Error Values**: Extend the evaluation relation $\Downarrow$ so that in addition to the intended result, it can also return *error values*. This approach requires adding potentially many additional rules to the language, as we can see in Figure 2, where 4 of the 8 rules only describe how to pass errors around.

The Appendix (Section A.1, optional reading) describes how we can directly take such semantics and turn them into an interpreter for our language.

2 Describing a Language with Variables

Programming languages derive most of their utility from their ability to abstract and reuse, with variables being the most central abstraction feature in practice. A formalism that allows us to describe semantics must thus also allow us to handle object languages that allow programmers to use variables. For example,

$$
\begin{array}{rcl}
\langle expr \rangle & \longrightarrow & nat \\
& | & id \\
& | & \langle expr \rangle \texttt{+} \langle expr \rangle \\
& | & \texttt{let } id \texttt{ = } \langle expr \rangle \texttt{ in } \langle expr \rangle
\end{array}
$$

Figure 3: Language LetAdd, where *nat* is a natural number and *id* is a variable identifier.

consider the language in Figure 3: this language allows us to write expressions such as

```
let a = 1 + 2 + 3 in a + a
```

with the intent that we can assign a value to `a` and then reuse that value by simply using the name '`a`'. Thus, we may want the above expression to compute the number 12.

Most languages also give us variables whose contents can be rebound or updated; we will discuss this feature at a later stage.

We will now define the evaluation relation $\Downarrow$ for LetAdd. That is, when we talk about $\Downarrow$ in this section, we mean a different evaluation relation than in the previous section (since we are discussing a different language).

Our challenge now is to give a meaning to variables. For instance, in

$$\texttt{let a = 1 in 2 + a} \Downarrow n$$

we want n to be 3, but this means that we need to carry the information that we have bound `a` to 1 into the expression `2 + a`. That is, we need to know that in this context,

$$\texttt{a} \Downarrow 1$$

In a different context, such as `let a = 2 in 2 + a` $\Downarrow n$, we may want `a` $\Downarrow 2$ instead. Clearly we need to distinguish these cases, and we do so by passing an extra parameter to our evaluation relation, the *environment*.

2.1 Environments

An *environment* is a mathematical object that can tell us the bindings of variables (cf. Sebesta, Section 5.4). We will use the following notation:

$$E_2 = \{\texttt{a} \mapsto 1, \texttt{b} \mapsto 7\}$$

to describe an environment E_2 that knows that `a` is bound to 1 and `b` is bound to 7. To look up the binding of a variable x in an environment E, we write $E(x)$, e.g., $E_2(\texttt{a}) = 1$.

Observe here that `a` is a variable in the object language (LetAdd), whereas x is a variable in the meta-language (Natural semantics), such that x can bind to any object-language variable (including `a`).

We write

$$E_\emptyset = \{\}$$

for the empty environment. If we want to update an environment E with a new binding from x to v, we write $E[x \mapsto v]$. For example,

$$E_2[\texttt{c} \mapsto 42] = \{\texttt{a} \mapsto 1, \texttt{b} \mapsto 7, \texttt{c} \mapsto 42\}$$

or

$$E_2[\texttt{a} \mapsto 0] = \{\texttt{a} \mapsto 0, \texttt{b} \mapsto 7\}$$

We define this as follows:

$$E[x \mapsto v](y) = \begin{cases} v & \iff \quad y = x \\ E(y) & \quad\quad\; \textit{otherwise} \end{cases}$$

2.2 Defining Semantics with Environments

These environments E now become parameters to our evaluation relation $\Downarrow$. Borrowing notation from mathematical logic, we write this extra parameter in a somewhat unusual style (at least from the perspective of programmers):

$$E \vdash p \Downarrow n$$

Here, the turnstile ($\vdash$) separates the evaluation context (our environment E) from the description of the evaluation relation.

We can now use E in the definition of our evaluation rules:

$$E \vdash \texttt{a} \Downarrow E(\texttt{a})$$

Figure 4 shows the evaluation rules for our full language. Note how the extra parameter E is now threaded through all rules. Rules nat_l and add_l, for instance, are almost identical to the rules *nat* and *add* from the language DivAdd, except for E. Rule var_l shows how we look up variables in the environment.

Rule let_l describes how we update our environment: when we encounter an expression

$$\texttt{let } v \texttt{ = } e_1 \texttt{ in } e_2$$

we first evaluate e_1 to n_1 in the environment E (since e_1 might also contain variables), then we build a new environment $E[v \mapsto n_1]$, which we then use as the environment for evaluating e_2.

Figure 5 shows a full example of how we can now formally derive the semantics of a small program.

Once again our semantics is not entirely complete. For instance, the program `a` only has a meaning if `a` is in the environment. If the environment contains no binding for `a`, this should (in practice) be an error; we can handle this situation analogously to the division by zero for language DivAdd[2].

[2] A key practical difference is that we cannot predict division by zero at compile time, whereas for many languages we can at compile time predict whether there is a reference to an undefined name. Thus, we can already do this error handling as part of the static semantics, rather than as part of the dynamic semantics.

$$\frac{n \in nat}{E \vdash n \Downarrow \mathsf{asNat}(n)}\ (nat_l) \qquad \frac{E \vdash e_1 \Downarrow n_1 \quad E \vdash e_2 \Downarrow n_2}{E \vdash e_1\texttt{+}e_2 \Downarrow n_1 + n_2}\ (add_l)$$

$$\frac{E \vdash e_1 \Downarrow n_1 \quad E[v \mapsto n_1] \vdash e_2 \Downarrow n_2}{E \vdash \texttt{let } v \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow n_2}\ (let_l) \qquad \frac{v \in id}{E \vdash v \Downarrow E(v)}\ (var_l)$$

Figure 4: Semantics of LetAdd

$$\dfrac{\dfrac{\dfrac{\texttt{1} \in nat}{E_\emptyset \vdash \texttt{1} \Downarrow 1}\ (nat_l) \quad \dfrac{\texttt{2} \in nat}{E_\emptyset \vdash \texttt{2} \Downarrow 2}\ (nat_l)}{E_\emptyset \vdash \texttt{1 + 2} \Downarrow 3}\ (add_l) \quad \dfrac{\dfrac{\texttt{a} \in id}{E_\emptyset[\texttt{a} \mapsto 3] \vdash \texttt{a} \Downarrow 3}\ (var_l) \quad \dfrac{\texttt{a} \in id}{E_\emptyset[\texttt{a} \mapsto 3] \vdash \texttt{a} \Downarrow 3}\ (var_l)}{E_\emptyset[\texttt{a} \mapsto 3] \vdash \texttt{a + a} \Downarrow 6}\ (add_l)}{E_\emptyset \vdash \texttt{let a = 1 + 2 in a + a} \Downarrow 6}\ (let_l)$$

Figure 5: An example derivation of the semantics of an expression in LetAdd, using our natural semantics rules from Figure 4.

A Translations to Interpreters

The following listings (all optional reading) show how to translate the semantics for our languages into *interpreters* in different programming languages. This is purely for illustration; you are not expected to understand the code of languages that you are not already familiar with.

I recommend that you check translations of the semantics to languages that you are familiar with until you can see the connection between natural semantics and interpretation (unless you already do, of course). If you don't find a language that you know, please let us know!

A.1 Translations of DivAdd

Figure 2 presented the semantics of Language DivAdd with explicit error handling rules. In the following, we use a version of the semantics with implicit error handling rules (Figure 6).

Our implementations assume that someone else has already taken care of the translation of strings-of-digits to numbers (the asNat function).

A.1.1 DivAdd in Java

The Java implementation encodes all parts of the BNF in a tree with an interface Expr. Here, we implement the $\Downarrow$ relation as a method eval of the interface Expr, and each implementation of Expr implements its own part of the evaluation rule handling.

```
public interface Expr {
```

$$\frac{n \in nat}{n \Downarrow \mathsf{asNat}(n)}\ (nat) \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 \texttt{+} e_2 \Downarrow n_1 + n_2}\ (add)$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n_2 \neq 0}{e_1\ \texttt{div}\ e_2 \Downarrow \lfloor \frac{n_1}{n_2} \rfloor}\ (div)$$

Figure 6: Semantics for DivAdd without error handling. Instead we use the implicit rule that 'all other cases produce an Err value'.

```
    public int
    eval();
}

public class Num implements Expr {
    private int n;
    public Num(int n) {
        this.n = n;
    }

    public int
    eval() {
        return this.n;
    }
}

public class Add implements Expr {
    private Expr e1, e2;
    public Add(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    public int
    eval() {
        // no need to create variables n1, n2
        return e1.eval() + e2.eval();
    }
}

public class Div implements Expr {
    private Expr e1, e2;
    public Div(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
```

```
    public int
    eval() {
        int n2 = e2.eval();
        if (n2 == 0) {
            throw new RuntimeException("Err");
        }
        return e1.eval() / n2;
    }
}
```

A.1.2 DivAdd in Haskell

The translation into Haskell is very direct, representing $\Downarrow$ as the function eval:

```
-- Language syntax
data Expr = Num Int
          | Add Expr Expr
          | Div Expr Expr

-- Encoding the result as 'num' or 'Err'.
data Result = Value Int    -- normal result
            | Err          -- error result
            deriving Show -- to print results

-- (add)
eval (Num n) = Value n
-- (nat), using Pattern Guards
eval (Add e1 e2)
  | Value n1 <- eval e1
  , Value n2 <- eval e2
  = Value (n1 + n2)
-- (div)
eval (Div e1 e2)
  | Value n1 <- eval e1
  , Value n2 <- eval e2
  , n2 /= 0
  = Value (n1 `div` n2)
-- Catch-all for errors
eval _ = Err
```

A.2 Translations of LetAdd

A.2.1 LetAdd in Java

The Java implementation adds the environment parameter explicitly to the eval function.

```
public interface Expr {
    public int
    eval(Env E);
}

public class Num implements Expr {
    private int n;
    public Num(int n) {
        this.n = n;
    }

    public int
    eval(Env E) {
        return this.n;
    }
}

public class Add implements Expr {
    private Expr e1, e2;
    public Add(Expr e1, Expr e2) {
        this.e1 = e1;
        this.e2 = e2;
    }

    public int
    eval(Env E) {
        // no need to create variables n1, n2
        return e1.eval(E) + e2.eval(E);
    }
}

public class Var implements Expr {
    private String id;
    public Var(String id) {
        this.id = id;
    }

    public int
    eval(Env E) {
```

```
        return E.lookup(this.id);
    }
}

public class Let implements Expr {
    private String id;
    private Expr e1, e2;

    public Let(String id, Expr e1, Expr e2) {
        this.id = id;
        this.e1 = e1;
        this.e2 = e2;
    }

    public int
    eval(Env E) {
        int n1 = this.e1.eval(E);
        return this.e2.eval(E.add(this.id, n1));
    }
}
```

For the environment, we use the following helper class:

```
public class Env {
    // linked list-based implementation
    private Env parent;
    private String id;
    private int value;

    public Env() {
    }

    public Env(Env parent, String id, int value) {
        this.parent = parent;
        this.id = id;
        this.value = value;
    }

    public Env
    add(String id, int binding) {
        return new Env(this, id, binding);
    }

    public int
    lookup(String id) {
        Env e = this;
```

```
        while (e != null) {
            if (id.equals(e.id)) {
                return this.value;
            }
            e = e.parent;
        }
        throw new RuntimeException("Name not found: " + id);
    }
}
```