Semantics and Types

sk and dbtucker

2002-11-01

1 Semantics

We have been writing interpreters in Scheme in order to understand various features of programming languages. What if we want to explain our interpreter to someone else? If that person doesn’t know Scheme, we can’t communicate how our interpreter works. It would be convenient to have some common language for explaining interpreters. We already have one: math!

Let’s try some simple examples. If our program is a number $n$, it just evaluates to some mathematical representation of $n$. We’ll use a $\hat{n}$ to represent this number, whereas $n$ itself will hold the numeral. For instance, the numeral 5 is represented by the number $\hat{5}$ (note the subtle differences in typesetting!). In other words, we will write

$$n \Rightarrow \hat{n}$$

where we read $\Rightarrow$ as “reduces to”. Numbers are already values, so they don’t need further reduction.

How about addition? We might be tempted to write

$$\{+\ l\ r\} \Rightarrow \hat{l} + \hat{r}$$

In particular, the addition to the left of the $\Rightarrow$ is in the programming language, while the one on the right happens in mathematics and results in a number. That is, the addition symbol on the left is syntactic. It could map to any operation—a particularly perverse language might map it to multiplication! It is the expression on the right that gives it meaning, and in this case it assigns the meaning we would expect.

That said, this definition is unsatisfactory. Mathematical addition only works on numbers, but $l$ and $r$ might each be complex expressions in need of reduction to a value (in particular, a number) so they can be added together. We denote this as follows:

$$l \Rightarrow \hat{l}_v \quad r \Rightarrow \hat{r}_v$$

$$\{+ l r\} \Rightarrow \hat{l}_v + \hat{r}_v$$

The terms above the bar are called the antecedents, and those below are the consequents. This rule is just a convenient way of writing an “if . . . then” expression: it says that if the conditions in the antecedent hold, then those in the consequent hold. So if $l$ reduces to $\hat{l}_v$, and if $r$ reduces to $\hat{r}_v$, then adding the respective expressions results in the sum of their values. (In particular, it makes sense to add $l_v$ and $r_v$, since each is now a number.) A rule of this form is called a judgment, because based on the truth of the conditions in the antecedent, it issues a judgment in the consequent (in this case, that the sum will be a particular value).

These rules are slightly subtle, and there is a different way to understand them. The subtlety here is that they are actually binding names to values. That is, a different way of reading the rule is not as an “if . . . then” but rather as an imperative: it says “reduce $l$, call the result $\hat{l}_v$; reduce $r$, call its result $\hat{r}_v$; if these two succeed, then add $\hat{l}_v$ and $\hat{r}_v$, and declare the sum the result for the entire expression”. Seen this way, $l$ and $r$ are bound in the consequent to the sub-expressions of the addition term, while $\hat{l}_v$ and $\hat{r}_v$ are bound in the antecedent to the results of evaluation (or reduction). Seen this way, they truly are the abstract representation of our interpreter.

Let’s turn our attention to functions. We want them to evaluate to closures, which consist of a name, a body and an environment. How do we represent a structure in mathematics? A structure is simply a tuple, in this case a triple. (If we had multiple kinds of tuples, we might use tags to distinguish between them, but for now that won’t be necessary.) We would like to write

$$\{\text{fun}\ \{i\}\ b\} \Rightarrow \{i, b, ??\}$$
but the problem is we don’t have a value for the environment to store in the closure. So we’ll have to make the environment explicit. From now on, $\Rightarrow$ will always have a term and an environment on the left, and a value on the right. We first rewrite our two existing reduction rules:

\[
\begin{align*}
n, \mathcal{E} & \Rightarrow \hat{n} \\
l, \mathcal{E} & \Rightarrow \hat{l} \\
r, \mathcal{E} & \Rightarrow \hat{r} \\
\{+ l r\}, \mathcal{E} & \Rightarrow l + r
\end{align*}
\]

Now we can define a reduction rule for functions:

\[
\begin{align*}
\{ \text{fun} \ (i \ b) \}, \mathcal{E} & \Rightarrow \langle i, b, \mathcal{E} \rangle
\end{align*}
\]

Given an environment, we can also look up the value of identifiers:

\[
\begin{align*}
i, \mathcal{E} & \Rightarrow \mathcal{E}(i)
\end{align*}
\]

All that remains is application. As with addition, application must first evaluate its subexpressions, so the general form of an application must be as follows:

\[
\begin{align*}
f, \mathcal{E} & \Rightarrow ??? \\
a, \mathcal{E} & \Rightarrow ???
\end{align*}
\]

What kind of value must $f$ reduce to? A closure, naturally:

\[
\begin{align*}
f, \mathcal{E} & \Rightarrow \langle i, b, \mathcal{E}' \rangle \\
a, \mathcal{E} & \Rightarrow ???
\end{align*}
\]

(We’ll use $\mathcal{E}'$ to represent to closure environment to make clear that it may be different from $\mathcal{E}$.) We don’t particularly care what kind of value $a$ reduces to; we’re just going to substitute it:

\[
\begin{align*}
f, \mathcal{E} & \Rightarrow \langle i, b, \mathcal{E}' \rangle \\
a, \mathcal{E} & \Rightarrow a_v
\end{align*}
\]

But what do we write below? We have to evaluate the body, $b$, in the extended environment; whatever value it returns is the value of the application. So the evaluation of $b$ also moves into the antecedent:

\[
\begin{align*}
f, \mathcal{E} & \Rightarrow \langle i, b, \mathcal{E}' \rangle \\
a, \mathcal{E} & \Rightarrow a_v \\
b, ??? & \Rightarrow b_v
\end{align*}
\]

In what environment do we reduce the body? It has to be the environment in the closure; if we use $\mathcal{E}$ instead of $\mathcal{E}'$, we introduce dynamic rather than static scoping! But additionally, we must extend $\mathcal{E}'$ with a binding for the identifier named by $i$; in particular, it must be bound to the value of the argument. We can write all this concisely as:

\[
\begin{align*}
f, \mathcal{E} & \Rightarrow \langle i, b, \mathcal{E}' \rangle \\
a, \mathcal{E} & \Rightarrow a_v \\
b, \mathcal{E}'[i \leftarrow a_v] & \Rightarrow b_v
\end{align*}
\]

where $\mathcal{E}'[i \leftarrow a_v]$ means “the environment $\mathcal{E}'$ extended with the identifier $i$ bound to the value $a_v$”. If $\mathcal{E}'$ already has a binding for $i$, this extension shadows that binding.

The judicious use of names conveys information here. We’re demanding that the value used to extend the environment must be the same as that resulting from evaluating $a$: the use of $a_v$ in both places indicates that. It also places an ordering on operations: clearly the environment can’t be extended until $a_v$ is available, so the argument must evaluate before application can proceed with the function’s body. The choice of two different names for environments—$\mathcal{E}$ and $\mathcal{E}'$—therefore denotes that the two environments need not be the same, and the names help us keep them distinct.

We call this a big-step operational semantics. It’s a semantics because it ascribes meanings to programs. (We saw how a small change could result in dynamic instead of static scope, and more mundanely, that the meaning of $+$ was given to be addition, not some other binary operation.) It’s operational because evaluation largely proceeds in a mechanical fashion; we aren’t compiling the entire program into a mathematical object and using fancy math to reduce it to an answer. Finally, it’s big-step because $\Rightarrow$ reduces expressions down to irreducible answers. In contrast, a small-step semantics performs one reduction at a time.
Exercises
Write reduction rules for conditionals and recursion.

2 Types

2.1 Motivation

Until now, we’ve partially ignored the problem of program errors. We haven’t done so entirely: if a programmer writes

```
{fun {x}}
```

we do reject this program, because it is not syntactically legal—every function must have a body. But what if, instead, he were to write

```
{+ 3
 {fun {x} x}}
```

Right now, our interpreter might produce an error such as

```
umv-n: not a number
```

What’s happening here? A check deep in the bowels of our interpreter is detecting the use of a non-numeric value in a position expecting a number.

At this point, we can make the same distinction between the syntactic and meta levels about errors as we did about representations. The error above is an error at the syntactic level,¹ because the interpreter is checking for the correct use of its internal representation. If we had division in the interpreted language, however, and if the corresponding `numV/` procedure failed to check that the denominator was non-zero, the error would come directly from Scheme’s division procedure. At that point, we would be relying fully on the meta implementation. If it caught errors, our interpreter would halt, whereas if the implementation did not adequately catch errors, our interpreter would have to either catch them explicitly or provide an unfaithful implementation to the user.

Of course, this discussion about the source of error messages somewhat misses the point: we really ought to reject this program without ever executing it. But the act of rejecting it has become harder, because this program is legitimate from the perspective of the parser. It’s only illegal from the semantic viewpoint, because the meaning of `+` is an operator that does not accept functions as arguments. Therefore, we clearly need a more sophisticated layer that checks for the validity of programs.

How hard is this? Rejecting the example above seems pretty trivial: indeed, it’s so easy, we could almost build this into the parser (to not accept programs that have syntactic functions as arguments to arithmetic primitives). But obviously, the problem is more difficult in general. Consider this program:

```
{+ 3
 {f 5}}
```

Is this program valid? Clearly, it depends on whether or not `f`, when applied to 5, evaluates to a number. You could reasonably say that if we were given the value of `f`, we could determine this easily. That is indeed true sometimes: for instance,

```
{with {f {fun {x} (+ x 1)}}
 {+ 3
 {f 5}}}
```

is clearly legal, whereas

```
{with {f {fun {x}
 {fun {y} (+ x y)}}}
 {+ 3
 {f 5}}}
```

¹Not to be confused with a syntax error!
is not. Here, simply substituting \( f \) in the body seems to be enough. The problem does not quite reduce to the parsing problem that we had earlier—a function application is necessary to determine the program’s validity. But we would like to determine when a program will lead to an error without having to run it! This is the province of types.

Starting with this lecture, we will study type systems (a term we will make more formal over time; for now, we can think of it as the collection of types in the language) in considerable detail. First, we need to build an intuition for the problems that types can address, and the obstacles that they face. Consider the following program:

\[
\{ + 3 \\
 \text{if}0 \mystery \ \\
 5 \\
\{ \text{fun} \ (x) \ x \} \}
\]

This program executes successfully (and evaluates to 8) if \( \mystery \) is bound to 0, otherwise it results in an error. No parser can possibly detect that! The value of \( \mystery \) might arise from any number of sources. For instance, it may be bound to 0 only if some mathematical statement, such as the Collatz conjecture, is true.\(^2\) In fact, we don’t even need to explore such a specific terrain: our program may simply be

\[
\{ + 3 \\
 \text{if}0 \ \{ \text{read-number} \} \\
 5 \\
\{ \text{fun} \ (x) \ x \} \}
\]

Unless we can read the user’s mind, we have no way of knowing whether this program will execute without error. In general, even without involving the mystery of mathematical conjectures or the vicissitudes of users, we cannot statically determine whether a program will halt with an error, because of the Halting Problem.

This highlights an important moral:

Type systems are always prey to the Halting Problem. Consequently, a type system for a general-purpose language must always either over- or under-approximate: either it must reject programs that might have run without an error, or it must accept programs that will error when executed.

While this is a problem in theory, what impact does this have on practice? Quite a bit, it turns out. In languages like Java, programmers think they have the benefit of a type system, but in fact many common programming patterns force programmers to employ casts instead. Casts intentionally subvert the type system, leaving most of the validity checking for execution time. At present, it appears that avoiding casts entirely greatly hobbles the set of Java programs a user might construct. This clearly indicates that Java’s evolution is far from complete. In contrast, most of the type problems of Java are not manifest in a language like ML, but its type systems still holds a few (subtler) lurking problems. In short, there is still much to do before we can consider type system design a solved problem.

### 2.2 What Are Types?

Formally, a type is any property of a program that we can establish without executing the program. In particular, types capture the intuition above that we would like to predict a program’s behavior without executing it. Of course, given a general-purpose programming language, we cannot predict its behavior entirely without execution (think of the user input example, for instance). So any static prediction of behavior must necessarily be an approximation of what happens. People conventionally use the term type to refer not just to any approximation, but one that is an abstraction of the set of values.

A type labels every expression in the language, recording what kind of value evaluating that expression will yield.\(^3\) That is, types describe invariants that hold for all executions of a program. It approximates in that it records only what kind of value the expression yields, not the precise value itself. For instance, types for the language we have seen so far might include number and function. The operator + consumes only values of type number, thereby rejecting a program of the form

\(^2\)Consider the function \( f(n) \) defined as follows: If \( n \) is even, divide \( n \) by 2; if odd, compute \( 3n + 1 \). The Collatz conjecture posits that, for every positive integer \( n \), there exists some \( k \) such that \( f^k(n) = 1 \). (The sequences demonstrating convergence to 1 are often quite long, even for small numbers! For instance: \( 7 \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 \).)

\(^3\)An expression may well yield different kinds of values on different executions within the same program. We’ll see much more about this in subsequent lectures! For now, let’s assume it’s not a problem.
To reject this program, we did not need to know precisely which function was the second argument to +, be it \((\text{fun } \{x\} \ x)\) or \((\text{fun } \{x\} \ (\text{fun } \{y\} \ (+ \ y \ x)))\). Since we can easily infer that 3 has type \texttt{number} and \(\text{fun } \{x\} \ x\) has type \texttt{function}, we have all the information we need to reject the program without executing it!

Note that we are careful to refer to \textit{valid} programs, but never \textit{correct} ones. Types do not ensure the correctness of a program. They only guarantee that the program does not make certain kinds of errors. Many errors lie beyond the ambit of a type system, however, and are therefore not caught by it. Most type systems will not, for instance, distinguish between a program that sorts values in ascending order from one that sorts them in descending order, yet in many domains the difference between those two can have extremely telling consequences!

### 2.3 Typing Rules

First, we must agree on a language of types. Recall that types need to abstract over sets of values; earlier, we suggested two possible types, \texttt{number} and \texttt{function}. Since those are the only kinds of values we have for now, we could just go with those as our types.

We present a type system as a collection of \textit{typing rules}, which describe how to determine the type of an expression. There must be at least one type rule for every kind of syntactic construct so that, given a program, at least one type rule applies always to every sub-term. Usually, typing rules are recursive, and determine an expression’s type from the types of its parts.

The type of any numeral is \texttt{number}:

\[
\text{n : number}
\]

(read this as saying “any numeral \(n\) has type \texttt{number}”) and of any function is \texttt{function}:

\[
\{\text{fun } \{i\} \ b\} : \text{function}
\]

but what is the type of an identifier? Clearly, we need a type \textit{environment} (a mapping from identifiers to \textit{types}). It’s conventional to use \(\Gamma\) for the type environment. As with the value environment, the type environment must appear on the left of every type judgment. All type judgments will have the following form:

\[
\Gamma \vdash e : t
\]

where \(e\) is an expression and \(t\) is a type, which we read as ‘\(\Gamma\) proves that \(e\) has type \(t\)’. Thus,

\[
\begin{align*}
\Gamma \vdash n : \text{number} \\
\Gamma \vdash \{\text{fun } \{i\} \ b\} : \text{function} \\
\Gamma \vdash i : \Gamma(i)
\end{align*}
\]

The last rule simply says that the the type of identifier \(i\) is whatever type it is bound to in the environment.

This leaves only addition and application. Addition is quite easy:

\[
\Gamma \vdash i : \text{number} \quad \Gamma \vdash r : \text{number} \\
\Gamma \vdash \{+ \ l \ r\} : \text{number}
\]

All this leaves is the rule for application. We know it must have roughly the following form:

\[
\Gamma \vdash f : \text{function} \quad \Gamma \vdash a : \tau_a \quad \cdots \\
\Gamma \vdash \{f \ a\} : \text{???}
\]

where \(\tau_a\) is the type of the expression \(a\) (we will often use \(\tau\) to name an unknown type).

What’s missing? Compare this against the semantic rule for applications. There, the representation of a function held an environment to ensure we implemented static scoping. Do we need to do something similar here?

For now, we’ll take a much simpler route. We’ll demand that the programmer \textit{annotate} each function with the type it consumes and the type it returns. This will become part of a modified function syntax. That is, a programmer might write

\[
\{+ \ 3 \\
\{\text{fun } \{x\} \ x\}\}
\]
where the two type annotations are now required: the one immediately after the argument dictates what type of value the function consumes, while that after the argument but before the body dictates what type it returns. We must change our type grammar accordingly; to represent such types, we conventionally use an arrow (→), where the type before the arrow represents the argument and that after the arrow represents the function's return value:

type ::= number
   | (type → type)

(noticed that we have dropped the overly naïve type function from our type language). Thus, the type of the function above would be (number→number). The type of the outer function below

(fun {x : number} : (number → number)
   {fun {y : number} : number
    (+ x y)})

in (number→(number→number)), while the inner function has type (number→number). In type judgments, we will often leave out the outer parentheses around the types.

Equipped with these types, the problem of checking applications becomes easy:

\[ \frac{\Gamma \vdash f : (\tau_1 \to \tau_2) \quad \Gamma \vdash a : \tau_1}{\Gamma \vdash \{f \ a\} : \tau_2} \]

That is, if you provide an argument of the type the function is expecting, it will provide a value of the type it promises. Notice how the judicious use of the same type name accurately captures the sharing constraints we desire.

There is one final bit to the introductory type puzzle: how can we be sure the programmer will not lie? That is, a programmer might annotate a function with a type that is completely wrong (or even malicious). (A different way to look at this is, having rid ourselves of the type function, we must revisit the typing rule for a function declaration.) Fortunately, we can guard against cheating and mistakes quite easily: instead of blindly accepting the programmer's type annotation, we check it:

\[ \frac{\Gamma \vdash i : \tau_1}{\Gamma \vdash \{i : \tau_1\} \ b : \tau_2} \]

This rule says that we will believe the programmer's annotation if the body has type \( \tau_2 \) when we extend the environment with \( i \) bound to \( \tau_1 \).

Notice a few things about the types for functions and applications:

- When typing the function declaration, we assume the argument will have the right type and guarantee that the body, or result, will have the promised type.
- When typing a function application, we guarantee the argument has the type the function demands, and assume the result will have the type the function promises.

This interplay between assumptions and guarantees is quite crucial to typing functions. Notice that the two “sides” are carefully balanced against each other to avoid fallacious reasoning about program behavior. In addition,

- Just as number does not specify which number will be used, a function type does not limit which of many functions will be used. If, for instance, the type of an argument to a function is number→number→number, the argument could perform either addition or subtraction.\(^4\) The type checker is able to reject misuse of any function that has this type without needing to know which actual function the programmer used.

\(^4\)We're assuming the Curried form of the operators.
Puzzles

- It’s possible to elide the return type annotation on a function declaration, leaving only the argument type annotation. Do you see how?
- Because functions can be nested within each other, a function body may not be closed at the type of checking it. But we don’t seem to capture the definition environment for types the way we did for procedures. So how does such a function definition type check? For instance, how does the second example of a typed procedure above pass this type system?

3 Type System Design Forces

Designing a type system involves finding a careful balance between two competing forces:

1. Having more information makes it possible to draw richer conclusions about a program’s behavior, thereby rejecting fewer valid programs or permitting fewer buggy ones.

2. Acquiring more information is difficult:
   - It may place unacceptable restrictions on the programming language.
   - It may require greater computational expense.
   - It may force the user to annotate parts of a program. Many programmers (sometimes unfairly) balk at writing anything beyond executable code, and may thus view the annotations as onerous.
   - It may ultimately hit the limits of computability, an unsurpassable barrier. (Often, designers can surpass this barrier by changing the problem slightly, though this usually moves the task into one of the three categories above.)

We will see instances of this tension in this course, but a fuller, deeper appreciation of these issues is the subject of an entire course (or more) to itself!

4 Why Types?

Type systems are not easy to design, and are sometimes more trouble than they are worth. This is, however, only rarely true. In general, types form a very valuable first line of defense against program errors. Of course, a poorly-designed type system can be quite frustrating: Java programming sometimes has this flavor. A powerful type system such as that of ML, however, is a pleasure to use. Programmers who are familiar with the language and type system report that programs that type correctly often work correctly within very few development iterations!

Even in a language like Java, types (especially when we are not forced to subvert them with casts) perform several valuable roles:

- Naturally, when they detect legitimate program errors, they help reduce the time spent debugging.
- Type systems catch errors in code that is not executed by the programmer. This matters because if a programmer constructs a weak test suite, many parts of the system may receive no testing. The system may thus fail after deployment rather than during the testing stage. (Dually, however, passing a type checker makes many programmers construct poorer test suites—a most undesirable and unfortunate consequence!)
- They help document the program. As we discussed above, a type is an abstraction of the values that an expression will hold. Explicit type declarations therefore provide an approximate description of a method’s behavior.
- Compilers can exploit types to make programs execute faster, consume less space, spend less time in garbage collection, and so on.
- While no language can eliminate arbitrarily ugly code, a type system imposes a baseline of order that prevents at least a few truly impenetrable programs—or, at least, prohibits certain kinds of terrible coding styles (such as making lists whose elements aren’t all of the same type).