Chapter 23

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 $\widehat{n}$ to represent this *number*, whereas *n* itself will hold the *numeral*. For instance, the numeral 5 is represented by the number $\widehat{5}$ (note the subtle differences in typesetting!). In other words, we will write

$$n \Rightarrow \widehat{n}$$

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

$$\{+ \;\; l \;\; r\} \Rightarrow \widehat{l + r}$$

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 mathematical operation. A particularly perverse language might map it to multiplication, but more realistically, it is likely to map to addition modulo some some base to reflect fixed-precision arithmetic. It is the expression on the right that gives it meaning, and in this case it assigns the meaning we would expect (corresponding, say, to DrScheme's use of unlimited-precision numbers for integers and rationals).

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:

$$\frac{l \Rightarrow \widehat{l_v} \qquad r \Rightarrow \widehat{r_v}}{\{+\; l\; r\} \Rightarrow \widehat{l_v + r_v}}$$

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. If there are multiple conditions in the antecedent, they must all hold for

the rule to hold. So we read the rule above as: *if* l reduces to l_v, *and if* r reduces to 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 subtly also *bind* 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 l_v; reduce r, call its result r_v; if these two succeed, then add l_v and 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 l_v and r_v are bound in the antecedent to the results of evaluation (or reduction). This representation truly is an abstract description of the 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

$$\texttt{\{fun \{}i\texttt{\}}\ b\texttt{\}} \Rightarrow \langle i, b, ??? \rangle$$

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:

$$n, \mathscr{E} \Rightarrow \widehat{n}$$

$$\frac{l, \mathscr{E} \Rightarrow \widehat{l_v} \qquad r, \mathscr{E} \Rightarrow \widehat{r_v}}{\{\texttt{+}\ l\ r\}, \mathscr{E} \Rightarrow \widehat{l_v + r_v}}$$

Now we can define a reduction rule for functions:

$$\texttt{\{fun \{}i\texttt{\}}\ b\texttt{\}}, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E} \rangle$$

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

$$i, \mathscr{E} \Rightarrow \mathscr{E}(i)$$

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:

$$\frac{f, \mathscr{E} \Rightarrow ??? \qquad a, \mathscr{E} \Rightarrow ???}{\{f\ a\}, \mathscr{E} \Rightarrow ???}$$

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

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow ???}{\{f\ a\}, \mathscr{E} \Rightarrow ???}$$

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

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow a_v}{\{f\ a\}, \mathscr{E} \Rightarrow ???}$$

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:

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow a_v \qquad b, ??? \Rightarrow b_v}{\{f\ a\}, \mathscr{E} \Rightarrow b_v}$$

In what environment do we reduce the body? It has to be the environment in the closure; if we use $\mathscr{E}$ instead of $\mathscr{E}'$, we introduce dynamic rather than static scoping! But additionally, we must extend $\mathscr{E}'$ 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

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow a_v \qquad b, \mathscr{E}'[i \leftarrow a_v] \Rightarrow b_v}{\{f\ a\}, \mathscr{E} \Rightarrow b_v}$$

where $\mathscr{E}'[i \leftarrow a_v]$ means "the environment $\mathscr{E}'$ extended with the identifier i bound to the value a_v". If $\mathscr{E}'$ 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—$\mathscr{E}$ and $\mathscr{E}'$—denotes that the two environments need not be the same.

We call this a *big-step operational semantics*. It's a *semantics* because it ascribes meanings to programs. (We can see how a small change can result in dynamic instead of static scope and, more mundanely, that the meaning of + is 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 atomic reduction at a time, rather like a stepper in a programming environment.

Exercise 23.0.2 *Extend the semantics to capture conditionals.*

Exercise 23.0.3 *Extend the semantics to capture lists.*
Hint: *You may want to consider tagging tuples.*

Exercise 23.0.4 *Extend the semantics to capture recursion.*

Exercise 23.0.5 *Alter the semantics to reflect lazy evaluation instead.*

Chapter 25

Type Judgments

25.1 What They Are

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, `number` and `function`. Since those are the only kinds of values we have for now, let's use those as our types.

We present a type system as a collection of rules, known formally as *type judgments*, which describe how to determine the type of an expression.[1] 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 to every sub-term. Judgments are often recursive, determining an expression's type from the types of its parts.

The type of any numeral is `number`:

$$n : \texttt{number}$$

(read this as saying "any numeral n has type `number`") and of any function is `function`:

$$\{\texttt{fun } \{i\} \ b\} : \texttt{function}$$

but what is the type of an identifier? Clearly, we need a *type environment* (a mapping from identifiers to *types*). It's conventional to use Γ (the upper-case Greek "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 "Γ proves that e has type t". Thus,

$$\Gamma \vdash n : \texttt{number}$$

$$\{\texttt{fun } \{i\} \ b\} : \texttt{function}$$

$$\Gamma \vdash i : \Gamma(i)$$

[1] A *type system* for us is really a collection of types, the corresponding judgments that ascribe types to expressions, *and* an an algorithm for perform this ascription. For many languages a simple algorithm suffices, but as languages get more sophisticated, devising this algorithm can become quite difficult, as we will see in Section 30.

The last rule simply says that 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:

$$\frac{\Gamma \vdash l : \texttt{number} \qquad \Gamma \vdash r : \texttt{number}}{\Gamma \vdash \{\texttt{+ } l \ r\} : \texttt{number}}$$

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

$$\frac{\Gamma \vdash f : \texttt{function} \qquad \Gamma \vdash a : \tau_a \qquad \cdots}{\Gamma \vdash \{f \ a\} : ???}$$

where τ_a is the type of the expression a (we will often use τ 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 *annotate* each function with the type it consumes and the type it returns. This will become part of a modified function syntax. That is, the language becomes

```
<TFWAE> ::= ...
          | {fun {<id> : <type>} : <type> <TFWAE>}
```

(the "T", naturally, stands for "typed") 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. An example of a function definition in this language is

```
{fun {x : number} : number
  {+ x x}}
```

We must also change our type grammar; to represent function types we conventionally use an arrow, where the type at the tail of the arrow represents the type of the argument and that at the arrow's head represents the type of the function's return value:

```
<type> ::= number
         | (<type> -> <type>)
```

(notice that we have dropped the overly naive 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}}}
```

is `(number -> (number -> number))`, while the inner function has type `(number -> number)`.

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

$$\frac{\Gamma \vdash f : (\tau_1 \rightarrow \tau_2) \qquad \Gamma \vdash a : \tau_1}{\Gamma \vdash \{f \ a\} : \tau_2}$$

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 τ_1 and τ_2 accurately captures the sharing 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:

$$\frac{\Gamma[i \leftarrow \tau_1] \vdash b : \tau_2}{\Gamma \vdash \texttt{fun } \{i \texttt{ : } \tau_1\} \texttt{ : } \tau_2 \ b\} : (\tau_1 \rightarrow \tau_2)}$$

This rule says that we will believe the programmer's annotation if the body has type τ_2 when we extend the environment with i bound to τ_1.

There is an important relationship between the type judgments for function declaration and for application:

- 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. 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 a function is `(number -> number)`, the function could be either increment or decrement (or a lot else, besides). The type checker is able to reject misuse of *any* function that has this type without needing to know which actual function the programmer will use.

By the way, it would help to understand the status of terms like *i* and *b* and *n* in these judgments. They are "variable" in the sense that they will be replaced by some program term: for instance, $\{\texttt{fun } \{i \texttt{ : } \tau_1\} \texttt{ : } \tau_2 \ b\}$ may be instantiated to `{fun {x : number} : number x}`, with *i* replaced by `x`, and so forth. But they are not program variables; rather, they are variables that stand for program text (including program variables). They are therefore called *metavariables*.

Exercise 25.1.1 *It's possible to elide the return type annotation on a function declaration, leaving only the argument type annotation. Do you see how?*

Exercise 25.1.2 *Because functions can be nested within each other, a function body may not be closed at the time of type-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?*

25.2 How Type Judgments Work

Let's see how the set of type judgments described above accepts and rejects programs.

1. Let's take a simple program,

```
{+ 2
   {+ 5 7}}
```

   We stack type judgments for this term as follows:

$$\dfrac{\emptyset \vdash \texttt{2} : \texttt{number} \qquad \dfrac{\emptyset \vdash \texttt{5} : \texttt{number} \qquad \emptyset \vdash \texttt{7} : \texttt{number}}{\emptyset \vdash \texttt{\{+ 5 7\}} : \texttt{number}}}{\emptyset \vdash \texttt{\{+ 2 \{+ 5 7\}\}} : \texttt{number}}$$

   This is a *type judgment tree*.[2] Each node in the tree uses one of the type judgments to determine the type of an expression. At the leaves (the "tops") are, obviously, the judgments that do not have an antecedent (technically known as the *axioms*); in this program, we only use the axiom that judges numbers. The other two nodes in the tree both use the judgment on addition. The metavariables in the judgments (such as l and r for addition) are replaced here by actual expressions (such as `2`, `5`, `7` and `{+ 5 7}`): we can employ a judgment only when the pattern matches consistently. Just as we begin evaluation in the empty environment, we begin type checking in the empty *type* environment; hence we have $\emptyset$ in place of the generic Γ.

   Observe that at the end, the result is the type `number`, not the value 14.

2. Now let's examine a program that contains a function:

```
{{fun {x : number} : number
   {+ x 3}}
 5}
```

   The type judgment tree looks as follows:

$$\dfrac{\dfrac{\dfrac{[\texttt{x}\leftarrow\texttt{number}] \vdash \texttt{x} : \texttt{number} \qquad [\texttt{x}\leftarrow\texttt{number}] \vdash \texttt{3} : \texttt{number}}{[\texttt{x}\leftarrow\texttt{number}] \vdash \texttt{\{+ x 3\}} : \texttt{number}}}{\emptyset \vdash \texttt{\{fun \{x : number\} : number \{+ x 3\}\}} : (\texttt{number}\rightarrow\texttt{number})} \qquad \emptyset \vdash \texttt{5} : \texttt{number}}{\emptyset \vdash \texttt{\{\{fun \{x : number\} : number \{+ x 3\}\} 5\}} : \texttt{number}}$$

   When matching the sub-tree at the top-left, where we have just Γ in the type judgment, we have the extended environment in the actual derivation tree. We must use the same (extended) environment consistently, otherwise the type judgment for addition cannot be applied. The set of judgments used

[2] If it doesn't look like a tree to you, it's because you've been in computer science too long and have forgotten that real trees grow upward, not downward. Botanically, however, most of these "trees" are really shrubs.

to assign this type is quite different from the set of judgments we would use to evaluate the program: in particular, we type "under the `fun`", i.e., we go into the body of the `fun` even if the function is never applied. In contrast, we would never evaluate the body of a function unless and until the function was applied to an actual parameter.

3. Finally, let's see what the type judgments do with a program that we know to contain a type error:

```
{+ 3
   {fun {x : number} : number
     x}}
```

The type judgment tree begins as follows:

$$\frac{???}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

We don't yet know what type (if any) we will be able to ascribe to the program, but let's forge on: hopefully it'll become clear soon. Since the expression is an addition, we should discharge the obligation that each sub-expression must have numeric type. First for the left child:

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad ???}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

Now for the right sub-expression. First let's write out the sub-expression, then determine its type:

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad \emptyset \vdash \texttt{\{fun \{x : number\} : number x\}} : ???}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

As per the judgments we have defined, any function expression must have an arrow type:

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad \emptyset \vdash \texttt{\{fun \{x : number\} : number x\}} : (??? \rightarrow ???)}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

This does the type checker no good, however, because arrow types are distinct from numeric types, so the resulting tree above does not match the form of the addition judgment (no matter what goes in place of the two ???'s). To match the addition judgment the tree must have the form

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad \emptyset \vdash \texttt{\{fun \{x : number\} : number x\}} : \texttt{number}}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

Unfortunately, we do not have any judgments that let us conclude that a syntactic function term can have a numeric type. So this doesn't work either.

In short, we cannot construct a legal type derivation tree for the original term. Notice that this is not the same as saying that the tree directly identifies an error: it does not. A type error occurs when we are *unable to construct a type judgment tree*.

This is subtle enough to bear repeating: To flag a program as erroneous, we must *prove* that no type derivation tree can possibly exist for that term. But perhaps some sequence of judgments that we haven't thought of exists that (a) is legal and (b) correctly ascribes a type to the term! To avoid this we may need to employ quite a sophisticated proof technique, even human knowledge. (In the third example above, for instance, we say, "we do not have any judgments that let us conclude that a syntactic function term can have a numeric type". But how do we know this is true? We can only conclude this by carefully studying the structure of the judgments. A computer program might not be so lucky, and in fact may get stuck endlessly trying judgments!)

This is why a set of type judgments alone does not suffice: what we're really interested in is a type system that includes an algorithm for type-checking. For the set of judgments we've written here, and indeed for the ones we'll study initially, a simple top-down, syntax-directed algorithm suffices for (a) determining the type of each expression, and (b) concluding that some expressions manifest type errors. As our type judgments get more sophisticated, we will need to develop more complex algorithms to continue producing tractable and useful type systems.