Semantics and Types

Shriram Krishnamurthi

2003-10-27

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 $\widehat{n}$ 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 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:

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

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). 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

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

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:

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

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

Now we can define a reduction rule for functions:

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

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

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

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:

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

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

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

(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:

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

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:

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

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

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

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 $\mathcal{E}'$—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.

Exercises

Write reduction rules for conditionals and recursion.

2 Types

2.1 Motivation

Until now, we've largely 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 isn't 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

```
numV-n: not a number
```

A check deep in the bowels of our interpreter is flagging 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 interpreter's behavior would be that of Scheme's on division-by-zero. If we had expected an error and Scheme did not flag one (or vice versa), then the interpreter would be unfaithful to the intent of the interpreted language.

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. Sometimes it does not seem much harder: 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}}}
```

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 consider this program:

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

[1] Not to be confused with a syntax error!

Is this program valid? Clearly, it depends on whether or not `f`, when applied to `5`, evaluates to a number. Since this expression may be used in many different contexts, we cannot know whether or not this is legal without examining each application, which in turn may depend on other substitutions, and so on. In short, it appears that we will need to run the program just to determine whether `f` is always bound to a function, and one that can accept numbers—but running the program is precisely what we're trying to avoid!

We now commence the study of *types* and *type systems*, which are designed to identify the abuse of types before executing a program. 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
   {if0 mystery
        5
        {fun {x} x}}}
```

This program executes successfully (and evaluates to `8`) if `mystery` is bound to `0`, otherwise it results in an error. 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 something quite so exotic: our program may simply be

```
{+ 3
   {if0 {read-number}
        5
        {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. This 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?

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. They approximate this information in that they typically record 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: $7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.)

[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.

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

To reject this program, we did not need to know precisely which function was the second argument to +, be it `{fun {x} x}` or `{fun {x} {fun {y} {+ x y}}}`. Since we can easily infer that `3` has type number and `{fun {x} x}` has type function, we have all the information we need to reject the program without executing it.

Note that we are careful to refer to *valid* programs, but never *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, number and 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 *typing rules*, which describe how to determine the type of an expression.[4] 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 number:

$$n : \mathsf{number}$$

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

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

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 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,

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

$$\Gamma \vdash \{fun\ \{i\}\ b\} : \mathsf{function}$$

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

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:

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

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

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

where τ_a is the type of the expression a (we will often use τ to name an unknown type).

[4] A *type system* for us is really a collection of types, the corresponding typing rules that ascribe types to expressions, *and* an an algorithm for perform this ascription. For many languages a simple, natural algorithm suffices, but as languages get more sophisticated, devising this algorithm can become quite difficult. We will in coming weeks see an instance of an algorithm that, though regarded quite standard now, was very sophisticated when it was first introduced.

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, a programmer might write

```
{fun {x : number} : number
  {+ 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 ($\rightarrow$), 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* $\rightarrow$ *type*)

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

```
{fun {x : number} : (number -> number)
  {fun {y : number} : number
    {+ x y}}}
```

is (number$\rightarrow$(number$\rightarrow$number)), while the inner function has type number$\rightarrow$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 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:

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

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$\rightarrow$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.

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 Judgments at Work

Let's see how the set of type judgments described above accept and reject programs.

1. Let's take a simple program,

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

   We stack type judgments for this term as follows:

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

   This is a *type judgment tree*.[5] 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 expression variables 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:

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

   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 to assign this type is quite different from the set of rules 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.

[5] 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.

3. Finally, let's see what the type judgments do with a buggy program:

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

The type judgment tree begins as follows:

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

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:

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

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

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

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

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

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

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

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 quite subtle: To mark 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 useful type systems.

4 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 incur 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!

5 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:

- When type systems 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!)
- Types 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 code'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.